И. А. БАРКОВ

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебник

САНКТ-ПЕТЕРБУРГ
МОСКВА
КРАСНОДАР
2019

УДК 004.04
ББК 32.973.26-018.1я73
Б 25

Барков И. А.

Б 25 Объектно-ориентированное программирование: Учебник. — СПб.: Издательство «Лань», 2019. — 700 с.: ил. — (Учебники для вузов. Специальная литература).

ISBN 978-5-8114-3586-9

Является учебником по одноименной дисциплине. В первой части кратко рассмотрены: природа классов и объектов, объектно-ориентированный анализ и проектирование программных систем, взаимодействие с операционной системой Windows, абстрактные типы данных.

Во второй части подробно обсуждаются базовые основы объектно-ориентированного программирования: наследование, инкапсуляция, полиморфизм, событийное взаимодействие. Графические модели структур классов и структур объектов используются для наглядного представления принципов объектно-ориентированного программирования, что позволяет глубже понять происходящие в объектно-ориентированной программе процессы.

Изложение сопровождается программными примерами с использованием инструментальной среды программирования RAD Studio и языка C++Builder. Принципиальные инвариантные положения могут быть полезны при изучении других языков и систем программирования.

Учебник предназначен для преподавателей, студентов, магистрантов, аспирантов, программистов.

УДК 004.04
ББК 32.973.26-018.1я73

Обложка
Е. А. ВЛАСОВА

© Издательство «Лань», 2019
© И. А. Барков, 2019
© Издательство «Лань»,
художественное оформление, 2019

ПРЕДИСЛОВИЕ

Первая половина трудностей освоения программирования заключается в изучении идей, заложенных в язык программирования, вторая — в развитии навыков выражать свои идеи с помощью языка программирования.

Автор неизвестен

История развития программирования прошла три этапа, характеризующихся разными уровнями сложности освоения программирования как профессиональной деятельности.

Первый этап характерен очень высоким уровнем сложности. Программирование осуществляется на языке кодов компьютера, позже — на языке ассемблера. От программиста требуются глубокие знания архитектуры компьютера: системы команд, методов управления памятью, вычислительным процессом и т. д. Появившаяся в это время профессия «программист» относилась к элитным, загадочным.

С появлением языков программирования высокого уровня начался второй этап, характеризующийся резким снижением уровня начальной профессиональной подготовки. Идеи языка программирования стали доступны широкому кругу различных специалистов, и, как следствие, программистская деятельность стала составной частью многих профессий. В этот период было создано огромное число достаточно простых (по современным меркам) программ. Потребность в автоматизации разного рода задач была удовлетворена. Однако к концу XX века стали резко усложняться передаваемые компьютеру задачи. На этом фоне усилились предъявляемые к программам требования: правильности, надежности, корректности, совместимости, универсальности, проверяемости, дружественного к пользователю интерфейса, простоты совершенствования и сопровождения. На рубеже 1980-х годов появилось понятие «кризис программирования». По экзотическим прогнозам тех лет, к 2005–2010 годам для удовлетворения потребности в автоматизации труда все человечество должно было переключиться на программирование. В ответ на возникшие проблемы наука программирования выдвинула множество новых идей и пересмотрела старые. Была изучена и развита концепция типов данных языков программирования. Появилось понятие «инкапсуляция», позволившее упорядочить вычислительные структуры. Использование полиморфизма и правил именования идентификаторов придало программам свойство модифицируемости. Переход к технологии персональных компьютеров, сделавший этот инструмент массовым, способствовал созданию специальных средств управления вычислительным процессом, получивших общее название «дружественный интерфейс». В практику программирования были введены концепции событийного программирования, защищенных от ошибок программных кодов, «экранных художников» и пр. Наконец, сформировалась идея объектно-ориентированного программирования.

С появлением объектно-ориентированного программирования разговоры о «кризисе программирования» постепенно прекратились. Начался новый, тре-

тий этап в сложности освоения базовых знаний для занятия программистской деятельностью. Теперь весьма поверхностное освоение приемов программирования позволяет создавать программы. Легкость получения работающей программы создает у программистов иллюзию достаточной подготовленности. Однако профессионал, как правило, найдет в таких программах множества недостатков. Чаще всего качество таких программ не удовлетворяет современным требованиям надежности, модифицируемости, защищенности от несанкционированного использования. Для создания, отвечающего всем современным требованиям программного изделия, необходимо глубокое понимание идей, принципов объектно-ориентированного программирования, реализации их в системе программирования, знание устройства объектно-ориентированной программы, свободное владение объектно-ориентированной технологией создания программных систем. Таким образом, программирование стало одновременно простым и сложным.

Очередной этап развития программирования, по-видимому, будет характеризоваться постепенным переходом к способу создания программ путем манипуляции виртуальными объектами с применением средств искусственного интеллекта. Способ «кодирования программ» станет предметом изучения историков.

Приведенный в начале предисловия эпиграф характеризует две стороны освоения программирования. В настоящее время в учебно-методической литературе по программированию преобладает акцент на преодоление второй стороны трудностей освоения программирования: выражать свои идеи с помощью языка программирования. Предлагаемая книга ориентирована, в первую очередь, на изучение идей и принципов объектно-ориентированного программирования, без знания которых невозможно создавать качественные программы.

ЛАНЬ®

ВВЕДЕНИЕ

Программист должен обладать способностью первоклассного математика к абстракции и логическому мышлению в сочетании с эдисоновским талантом сооружать все, что угодно, из нуля и единиц. Он должен сочетать аккуратность бухгалтера с проницательностью разведчика, фантазию автора детективных романов — с трезвой практичностью экономиста. А кроме того, программист должен иметь вкус к коллективной работе, понимать интересы пользователя и многое другое...

Ершов Андрей Петрович

В современном программировании объектно-ориентированный подход является одним из популярнейших. Он превратился в стандарт во многих фирмах и программистских сообществах. Средства для поддержки объектно-ориентированного программирования вошли практически во все профессиональные системы программирования. Этот стиль вместе с организационными нововведениями резко повысил качество программ, производительность труда отдельного программиста, эффективность коллективной работы программистов.

В одной из первых систем объектно-ориентированного программирования Smalltalk объекты называются актерами, а их деятельность — ролями. В этом случае на плечи режиссера ложится обязанность подбора актеров, распределения между ними ролей, организации спектакля во времени. Указанная метафора очень хорошо подходит для описания взаимодействия человека с программой, программных компонентов между собой, программы с аппаратурой. При внесении изменений в исходную задачу чаще всего сообщество актеров остается неизменным, меняются их роли.

Объектно-ориентированное программирование базируется на интенсивном использовании сложных структур — объектов, которые соединяют внутри себя как данные, так и операции над данными. С использованием инкапсуляции и абстрагирования объекты организуются таким образом, что к данным прямого доступа нет и к ним можно обратиться только посредством операций. Этот прием позволяет безболезненно заменять внутреннее представление данных, что исключительно важно для переделки сложных программных систем. Для этой же цели в объектно-ориентированном программировании используются механизмы наследования и полиморфизма. Можно построить на базе уже имеющегося новый тип данных и впоследствии переопределить его данные и операции. Можно создавать полиморфные операции для обработки разных типов данных. Объектно-ориентированный подход открыл программистам новые возможности моделирования реального мира, тем самым смог поднять планку сложности программных систем. Справедливости ради заметим, что существенный выигрыш достигается только в крупных программных системах.

В качестве теоретической основы в объектно-ориентированном программировании используются абстрактные типы данных. Целью данной теории яв-

ляется обоснование автоматизированной процедуры формирования понятий, изменения понятий в процессе наследования, использования для определения одних понятий других понятий.

Очень важной стадией развития и свидетельством зрелости объектно-ориентированного программирования является объектно-ориентированная технология проектирования программных систем, которая предоставила средства и приемы предварительного описания и документирования сложных задач автоматизации, обогатила содержание этапа проектирования программных изделий, создала условия для автоматизации создания программных изделий.

Однако наличие развитых инструментов не гарантирует безусловное получение качественных программ, необходимо в совершенстве овладеть как самими инструментами, так и приемами их совместного использования.

Очевидно, что целью учебного процесса изучения объектно-ориентированного программирования является, с одной стороны, получение базовых знаний, позволяющих отчетливо представлять заложенные в технологию возможности создания качественных программ, с другой стороны предостережение от возможных ошибок неправильного и неполного использования содержащегося в инструменте потенциала. Поэтому нужна учебно-методическая литература, в которой основной акцент ставится на углубленное изучение идей и принципов объектно-ориентированного программирования.

Цель и особенности книги. Цель этой книги заключается в изложении идей и принципов объектно-ориентированного программирования. Как было отмечено в предисловии, для овладения профессиональным уровнем программирования идеи и принципы являются первичными, первичными они являются в настоящей книге. Для придания целостности изложению в книгу включены материалы смежных дисциплин. При этом используется методический прием: от метафоры объектного программирования — к принципам анализа и проектирования программ, от принципов проектирования программ к математической теории типов данных. Завершается цепочка рассмотрением внутреннего устройства программ с целью демонстрации реализации идей объектно-ориентированного программирования в системе программирования. Параллельно идет обсуждение практических приемов объектно-ориентированного программирования.

Стремление достичь цели и придать монографическую целостность рассмотренным в книге вопросам привело к тому, что пришлось опустить некоторые детали. Поэтому изучение визуальных компонентов осуществляется только на уровне принципов построения всей системы визуальных компонентов. Справочная информация по визуальным компонентам приводится ограниченно. Этому посвящена обширная учебная и профессиональная литература, в которой, однако, за множеством мелких деталей описания компонентов часто скрывается суть объектно-ориентированного программирования.

Указанные методические приемы раскрытия и изложения темы в течение ряда лет были апробированы в курсе лекций по объектно-ориентированному программированию и показали свою состоятельность.

Содержание книги. Книга состоит из разделенных на две части двенадцати глав и приложений.

В первой части книги рассмотрены необходимые для наших рассуждений основные идеи и принципы объектно-ориентированного анализа и проектирования программ, язык общения специалистов по разработке и документированию программных изделий, введение в абстрактные типы данных, основы построения операционной системы Windows.

В первой главе рассматривается метафора объектно-ориентированного программирования. Под *метафорой* понимается объяснение сложных явлений простым, общедоступным языком. Вводятся основные понятия и обсуждаются этапы создания программных изделий, рассматриваются трудности современного программирования и способы их преодоления путем использования объектно-ориентированного подхода.

Во второй главе приведен обзор объектно-ориентированной технологии проектирования программных систем. Изложение не претендует на полноту. В главу включены основные понятия указанной технологии, без знания которых изучение объектно-ориентированного программирования становится неполноценным.

В третьей главе приведено в сокращенном виде описание языка UML (Unified Modeling Language) — унифицированного языка моделирования, который ориентирован на решение задач первых двух этапов жизненного цикла программ и признается многими специалистами как язык системотехнического мышления и стандарт документирования результатов анализа и проектирования программных систем.

Четвертая глава содержит описание теории абстрактных типов данных, являющейся математическим обоснованием структурирования данных и операций в программировании. Знание этой теории поможет глубже понять принципы объектно-ориентированного программирования и приемы реализации этих принципов в системах программирования.

В пятой главе представлены основные сведения об операционной системе Windows, а также рассмотрены принципы взаимодействия операционной системы и прикладной программы.

Во второй части книги рассматривается устройство объектно-ориентированной программы и системы программирования. Основной акцент ставится на реализацию и использование принципов объектно-ориентированного программирования, знание которых позволяет программисту осознанно создавать объектно-ориентированные программы, используя при этом всю мощь применяемого инструмента. Обсуждаются базовые инструменты объектно-ориентированного программирования: наследование, инкапсуляция, полиморфизм. Рассмотрены принципы построения библиотеки визуальных компонентов с использованием объектно-ориентированного подхода. Включены также темы событийного программирования и защиты программы от ошибок.

В шестой главе вводятся абстрактные модели структуры класса и структуры объекта. В дальнейшем указанные модели активно используются для наглядного представления происходящих в программе процессов, демонстрации

принципов объектно-ориентированного программирования. Рассматриваются компоненты класса и объекта, методы обработки классов и объектов.

В седьмой главе обсуждаются правила наследования полей, статического и динамического наследования методов.

В восьмую главу включено рассмотрение инкапсуляции, являющейся основополагающим принципом современного программирования.

Девятая глава посвящена описанию технологии реализации полиморфизма в создаваемых программах.

В десятой главе приведена методика событийного программирования, рассмотрены приемы обработки в программе событий и сообщений.

В одиннадцатой главе рассматриваются объектно-ориентированные методы обработки ошибок в программе.

Двенадцатая глава содержит сведения о структуре, принципах построения и приемах использования библиотеки визуальных компонентов.

В приложении приведено:

1. Историческая справка по объектно-ориентированному программированию.
2. Правила именования идентификаторов, позволяющие облегчить чтение программы.
3. Пример объектно-ориентированной программы.

В целом изложение развивается от основных идей и принципов объектно-ориентированной технологии проектирования программных систем и теории абстрактных типов данных, которые постепенно конкретизируются изложением возможных приемов реализации указанных идей и принципов. Для освоения практического программирования нужно решить задачи, приведенные в конце глав, соблюдая при этом рекомендации объектно-ориентированного программирования.

Предполагаемые читатели. Книга адресована нескольким категориям читателей: бакалаврам, магистрантам и аспирантам, а также программистам, пытающимся прочно овладеть идеями и принципами объектно-ориентированного программирования, применению указанных идей и принципов в практической деятельности по созданию программных изделий; преподавателям. Овладение приведенными методическими приемами позволит выработать навыки создания качественных программ.

Методические рекомендации по работе с книгой. Для чтения книги необходимо освоение начального курса «Программирование», включая обработку динамических структур данных. Изложение ориентировано на язык программирования C++, рассматриваемые примеры — на C++Builder, однако принципиальные инвариантные положения могут быть полезны в других системах программирования.

В программировании традиционно рассматриваются логические и физические модели сущностей и явлений. Этот прием также использован в книге.

В первой части книги представлена «логическая» модель объектно-ориентированного программирования, представляющая собой взгляд человека, внешнюю сторону объектных моделей. Изложение носит ознакомительный характер, однако

предполагается, что при изучении второй части читатель достаточно свободно владеет изложенными в первой части понятиями и принципами.

Во второй части книги рассматривается «физическая» модель объектно-ориентированного программирования. Интенсивно используются понятия «структура класса» и «структура объекта» с использованием графической интерпретации. С помощью этих средств программные примеры сопровождаются диаграммами, в которых наглядно изображены происходящие в программе процессы. Рекомендуется тщательно изучить диаграммы и комментарии к ним. Большинство приведенных в книге примеров прошло практическую проверку с помощью разработанных автором и под его руководством визуальных, лингвистических и программных средств.

В мире программирования нет единого представления о правилах именования идентификаторов, многие фирмы разрабатывают такие правила для внутреннего пользования. По этой причине в настоящей книге разработаны правила именования идентификаторов (приведены в Приложении 2), которые использовались в течение ряда лет в учебных и реальных программных разработках и по мнению автора оказались удачными. Рекомендуется познакомиться с Приложением 2 как только в тексте встретится первый пример программного кода.

Приведенный в конце список литературы содержит только те источники, которые были использованы при написании книги. Формулы, рисунки, алгоритмы нумеруются последовательно внутри каждой главы. Например, ссылка вида А.В на рисунок означает, что рисунок с номером В находится в главе А.

В конце каждой главы приведены «Контрольные вопросы». С помощью контрольных вопросов выделяются ключевые моменты в содержании главы. При изучении каждой главы рекомендуется ответить на заданные вопросы или, в крайнем случае, вернуться к тексту главы и отыскать ответ на каждый вопрос.

В конце каждой главы приведены формулировки задач. Задачи могут быть использованы как при самостоятельном изучении, так и в аудиторных занятиях. Задачи главы 12 могут служить лабораторным практикумом по объектно-ориентированному программированию. Задания сформулированы как этапы разработки достаточно крупной прикладной программы, что соответствует назначению объектно-ориентированного программирования. Лабораторные работы были неоднократно апробированы в учебном процессе и выполняются студентами с интересом.

ЛАНЬ®

ЛАНЬ®

ЧАСТЬ I
ПРИНЦИПЫ И ТЕХНОЛОГИЯ СОЗДАНИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ

1. МЕТАФОРА ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Подобно тому, как взявший в руки молоток начинает видеть во всем окружающем только гвозди, проектировщик с объектно-ориентированным мышлением начинает воспринимать весь мир в виде объектов.

Гради Буч

Обращение к метафоре будет использоваться неоднократно. Поэтому дадим определение этого понятия.

Метафора в программировании означает соответствие между логическими компонентами языка программирования и привычными человеку понятиями.

И инженер, и художник должны хорошо чувствовать материал, с которым они работают. В объектно-ориентированной методологии анализа и проектирования сложных программных систем основными строительными блоками являются классы и объекты, которые служат в качестве абстракции реальной действительности. Мы будем интенсивно использовать понятие «абстракция», поэтому уточним, что под этим понимается. Обратимся к энциклопедическому определению.

Абстракция (от *лат*. abstraction — отвлечение):

1) метод научного познания, основанный на том, что при изучении некоторого явления, процесса не учитываются его несущественные стороны и признаки; это помогает упрощать картину изучаемого явления и рассматривать его как бы «в чистом виде»;

2) продукт познания (понятие, описание, закон, модель, идеальный объект и т. п.), рассмотренный в сопоставлении с конкретной эмпирической действительностью, которая не фиксируется в этом продукте во всем богатстве своих свойств и связей.

В этой главе мы рассмотрим природу классов и объектов, взаимоотношения между ними, а также сообщим полезные правила проектирования хороших абстракций.

1.1. Объекты

1.1.1. Понятие объекта

Термин «объект» в программном обеспечении впервые был введен в языке имитационного моделирования Simula и применялся для моделирования реальности. Интуитивно *объект можно определить как осязаемую реальность, обладающую индивидуальным поведением*. Об объектах можно думать как о полезных существах, которые «живут» в нашей программе и коллективно решают некоторую прикладную задачу. Наша обязанность заключается в том, чтобы создать эти существа, распределить между ними обязанности, регламентировать сценарий их поведения и взаимодействия при решении поставленной задачи.

В общем случае каждый объект «помнит» необходимую информацию, «умеет» выполнять некоторый набор действий и характеризуется набором свойств. То, что объект «помнит», хранится в его полях. То, что объект «умеет делать», реализуется в виде его внутренних функций, называемых *методами*. *Свойства* объектов аналогичны свойствам, которые мы наблюдаем у обычных предметов. Значения свойств можно устанавливать и читать. Программно свойства реализуются через поля и методы.

Рассмотрим расположенный на экранной форме объект «кнопка». Объект «кнопка» имеет свойство «цвет». Значение цвета «кнопка» запоминает в одном из своих полей. При изменении значения свойства «цвет» автоматически вызывается метод, который перерисовывает кнопку на экране. Этот пример позволяет сделать важный вывод: свойства имеют первостепенное значение для программиста, использующего объект. Чтобы понять суть и назначение незнакомого объекта, мы обязательно должны знать его свойства и методы, очень редко — поля (объект сам знает, что с ними делать).

Объект моделирует абстракцию окружающей действительности и таким образом существует во времени и пространстве. С точки зрения восприятия человеком объектом может быть:

— осязаемый и (или) видимый предмет (например, уже упоминавшаяся кнопка);

— нечто, воспринимаемое мышлением (например, учебный процесс);

— нечто, на что направлена мысль или наше действие (например, прогноз погоды).

Объектами реального мира не исчерпываются типы объектов, интересные при проектировании программных систем. Другие важные типы объектов вводятся на этапе проектирования, и их взаимодействие друг с другом служит механизмом отображения поведения более высокого уровня. Это приводит нас к более четкому определению: *объект представляет собой конкретный опознаваемый предмет, единицу или сущность* (*реальную или абстрактную*)*, имеющую четко определенное функциональное назначение в данной предметной области. В еще более общем плане объект может быть определен как нечто, имеющее четко очерченные границы*.

Представим себе завод, на котором создаются композитные материалы для таких различных изделий, как, скажем, велосипедные рамы и крылья самолетов. Заводы часто разделяются на цеха: механический, химический, электрический и т. д. Цеха подразделяются на участки, на каждом из которых установлено несколько единиц оборудования: штампы, прессы, станки. На производственных линиях можно увидеть множество емкостей с исходными материалами, из которых с помощью химических процессов создаются блоки композитных материалов. Затем из них делается конечный продукт — рамы или крылья. Каждый осязаемый предмет может рассматриваться как объект. Токарный станок имеет четко очерченные границы, которые отделяют его от обрабатываемого на этом станке композитного блока; рама велосипеда, в свою очередь, имеет четкие границы по отношению как к велосипеду, так и к участку с оборудованием.

Существуют такие объекты, для которых определены явные концептуальные границы, но сами объекты представляют собой неосязаемые события или процессы. Например, химический процесс на заводе можно трактовать как объект, так как он имеет четкую концептуальную границу, взаимодействует с другими объектами посредством упорядоченного и распределенного во времени набора операций и проявляет хорошо определенное поведение. Рассмотрим систему автоматизированного проектирования. Два тела, например сфера и куб, имеют, как правило, нерегулярное пересечение. Хотя эта линия пересечения не существует отдельно от сферы и куба, она все же является самостоятельным объектом с четко определенными концептуальными границами.

Объекты могут быть осязаемыми, но иметь размытые физические границы: реки, туман или толпы людей. Это верно только на достаточно высоком уровне абстракции. Для человека, идущего через полосу тумана, бессмысленно отличать «мой туман» от «твоего тумана». Однако рассмотрим карту погоды: полосы тумана в Москве и Казани представляют собой совершенно разные объекты.

Разумеется, такой взгляд несколько упрощен, так как существуют понятия, явно не являющиеся объектами. К их числу относятся атрибуты, такие как возраст, красота, цвет, эмоции (например, любовь или гнев). Однако потенциально все перечисленное — это свойства, присущие объектам. Можно также утверждать, что некоторый мужчина (объект) любит свою жену (другой объект) или что конкретный кот (еще один объект) — серый.

Объект имеет свойства, характеризуется состоянием, обладает некоторым хорошо определенным поведением и уникальной идентичностью.

Полезно понимать, что объект — это нечто, имеющее четко определенные границы, но этого недостаточно, чтобы отделить один объект от другого или дать оценку качества абстракции. На основе имеющегося опыта можно дать следующее определение:

Совокупность схожих по структуре и поведению объектов образует класс; термины «экземпляр класса» и «объект» взаимозаменяемы.

Контрольные вопросы

1. Сформулируйте своими словами понятие «объект».
2. Приведите примеры объектов из различных прикладных областей.
3. Что является объектом с точки зрения восприятия человеком?
4. Какие объекты образуют класс?
5. Какое альтернативное название применяется для объекта?

1.1.2. Свойства и состояние объекта

Рассмотрим торговый автомат, продающий напитки. Поведение такого объекта состоит в том, что после опускания в него монеты и нажатия кнопки автомат выдает выбранный напиток. Что произойдет, если сначала будет нажата кнопка выбора напитка, а потом уже опущена монета? Большинство автоматов при этом просто ничего не сделает, так как пользователь нарушил их основные правила. Другими словами, автомат играл роль (ожидание монеты), которую

пользователь игнорировал, нажав сначала кнопку. Или предположим, что пользователь автомата не обратил внимание на предупреждающий сигнал: «Бросьте столько мелочи, сколько стоит напиток» — и опустил в автомат лишнюю монету. В большинстве случаев автоматы не дружественны к пользователю и радостно заглатывают все деньги.

В каждой такой ситуации мы видим, что поведение объекта определяется его историей: важна последовательность совершаемых над объектом действий. Такая зависимость поведения от событий и от времени объясняется тем, что у объекта есть внутренние состояния. Для торгового автомата, например, состояние определяется суммой денег, опущенных до нажатия кнопки выбора. Другая важная информация — это набор воспринимаемых монет и запас напитков.

На основе рассмотренных примеров можно сказать: *Объект характеризуется перечнем* (*обычно статическим*) *всех свойств данного объекта и состоянием* (*обычно динамическим*) *— текущим набором значений этих свойств.*

Одним из свойств торгового автомата является способность принимать монеты. Это статическое (фиксированное) свойство, в том смысле, что оно есть существенная характеристика торгового автомата. С другой стороны, этому свойству соответствует динамическое значение, характеризующее количество принятых монет: сумма увеличивается по мере опускания монет в автомат. Значение свойства объекта, в свою очередь, может быть статическим (например, заводской номер автомата) и динамическим (количество и стоимость принятых монет).

Свойством объекта называется характеристика, которая может принимать некоторые значения.

К числу свойств объекта относятся присущие ему или приобретаемые им характеристики, черты, качества или способности, делающие данный объект самим собой. Например, для лифта характерным является то, что он сконструирован для поездок вверх и вниз, а не горизонтально. Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу объекта. Мы говорим «как правило», потому что в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения. Робот первоначально может рассматривать некоторое препятствие как статическое, а затем обнаруживает, что это дверь, которую можно открыть. В такой ситуации, по мере получения новых знаний, изменяется создаваемая роботом концептуальная модель мира.

Другим примером является экранная форма и уже упоминавшаяся кнопка. Экранная форма может появиться на экране без кнопки, однако по логике решения задачи может возникнуть потребность в использовании кнопки, и она появится. В этом случае кнопка будет характеризовать динамическое свойство экранной формы. Заметим, что в рассматриваемом примере кнопка может быть и статическим свойством формы. В этом случае кнопка создается вместе с формой, но можно сначала ее сделать невидимой, а затем — видимой. В последнем случае кнопка должна обладать дополнительным свойством «видимость», которое может принимать значения: «кнопка видима» и «кнопка невидима».

Все свойства имеют некоторые значения. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект. Состояние лифта может описываться числом 3, означающим номер этажа, на котором лифт в данный момент находится. Состояние торгового автомата описывается в терминах других объектов, например имеющихся в наличии напитков. Конкретные напитки — это самостоятельные объекты, отличные от торгового автомата: их можно пить (а автомат — нет) и совершать с ними иные действия.

Таким образом, мы установили различие между объектами и простыми величинами: простые количественные характеристики (например, число 3) являются «постоянными, неизменными и непреходящими», тогда как объекты существуют во времени, изменяются, имеют свойства и внутреннее состояние; преходящи и могут создаваться, уничтожаться и разделяться.

Тот факт, что всякий объект обладает свойствами, имеет состояние, означает, что всякий объект занимает определенное пространство (физически или в памяти компьютера).

Примеры. Предположим, что на языке C++ нам нужно создать регистрационные записи о сотрудниках. Можно сделать это следующим образом:

```
struct TEmployes [1]
{
   int  intIdentifier;
   char strName[intNameSize];
   char  strDepartment[intDepartmentSize];
   double  dubSalary;
};.
```

ЛАНЬ®

Каждый компонент в приведенной структуре обозначает конкретное свойство нашей абстракции регистрационной записи о сотруднике: **intIdentifier** — регистрационный номер сотрудника, **strName** — имя сотрудника, **strDepartment** — место работы сотрудника, **dubSalary** — получаемое сотрудником за труд вознаграждение. Описание определяет не объект, а класс объектов, поскольку оно не вводит какой-либо конкретный экземпляр (точнее, это описание определяет запись, семантика которой соответствует классу, у которого все поля открыты). Для того чтобы создать объекты данного класса, необходимо написать следующее:

```
TEmployes  objEmployee1, objEmployee2, objEmployee3;
```

В данном случае объявлено три различных объекта. Хотя свойства объектов являются общими (их состояние представляется единообразно), каждый из них живет самостоятельной жизнью, и в каждый момент времени обладает определенным состоянием — набором значений свойств. Пользователи могут считывать и произвольно изменять состояние каждого объекта.

Изменим введенное определение класса следующим образом:

```
class TEmployes
     {
       public:
```

ЛАНЬ®

[1] Правила именования идентификаторов приведены в Приложении 2.

```
        int EmployeeIdentifier();
        char* EmployeeName();
        char* EmployeeDepartment();
        double EmployeeSalary();
    protected:
        int  intIdentifier;
        char strName[intNameSize];
        char  strDepartment[intDepartmentSize];
        double dubSalary;
};
```

Новое определение несколько сложнее предыдущего, но по ряду соображений предпочтительнее. В новом определении свойства класса скрыты (**protected**) от доступа произвольных объектов. Мы даем пользователям право (**public**) только узнать состояние объекта путем вызова соответствующих функций. Изменить состояние объекта могут теперь только доверенные лица, в нашем случае (**protected**) — потомки класса **TEmployes** (в примере не показаны). Таким образом, у каждого объекта (экземпляра класса **TEmployes**) появляется определенное поведение: информировать о текущем состоянии все объекты и разрешать избранным объектам менять текущее состояние. Другое достоинство последнего определения связано с возможностью его повторного использования. Мы увидим, что механизм наследования позволяет повторно использовать абстракцию, а затем уточнить и многими способами специализировать ее. Прием специализации будет заключаться в следующем: взять уже известного сотрудника, снабдить его дополнительными свойствами, например, сведениями о владении акциями родного предприятия, и получить в результате новую абстракцию: сотрудника-акционера.

В заключение скажем, что каждый объект в системе инкапсулирует (изолирует) некоторое состояние и все состояние автоматизированной системы инкапсулировано в объекты. Однако инкапсуляция состояния объекта — это только начало, которого недостаточно, чтобы мы могли охватить полный смысл абстракций, которые мы вводим при разработке программ. По этой причине нам нужно разобраться в том, как объекты функционируют.

Контрольные вопросы

1. Что является свойством объекта?
2. Приведите примеры объектов и их свойств различных прикладных областей.
3. Что является состоянием объекта?
4. Приведите примеры различных состояний выбранного объекта.
5. Чем отличаются объекты со статическим и динамическим набором свойств?
6. Приведите пример состояния некоторого объекта, характеризующегося статическим набором значений свойств.
7. Приведите пример состояния некоторого объекта, характеризующегося динамическим набором значений свойств.
8. Какой механизм позволяет повторно использовать абстракцию с целью ее уточнения и специализации?

1.1.3. Поведение объекта

Наличие внутреннего состояния объектов означает, что порядок выполнения операций имеет существенное значение. Это наводит на мысль представить объект в качестве маленькой независимой машины. Действительно, для ряда объектов такой временной порядок настолько существен, что наилучшим способом их формального описания будет конечный автомат.

В процессе функционирования объекты общаются друг с другом путем посылки и получения сообщений. Каждое сообщение может представлять собой прямую просьбу выполнить то или иное действие, например сделать окно на экране видимым. Другой вид сообщения представляет собой уведомление о произошедшем в вычислительной системе событии, например нажатии кнопки клавиатуры. Получение уведомления, как правило, влечет за собой выполнение заранее запланированного ответного действия. Например, нажатие при определенных условиях клавиши «Enter» вызовет посылку сообщения, в ответ на полученное сообщение-уведомление может быть запущена на выполнение другая программа.

Продолжая аналогию с машинами, можно сказать, что объекты могут быть активными и пассивными. Активный объект имеет свои средства управления, а пассивный — нет. Это означает, что активный объект может отправлять сообщения другим объектам, а пассивный объект обладает возможностью только получать сообщения. Активный объект в общем случае автономен, т. е. он может проявлять свое поведение без воздействия со стороны других объектов. Пассивный объект, напротив, может изменять свое состояние только под воздействием других объектов. Таким образом, активные объекты системы-источники управляющих воздействий. Если система имеет несколько потоков управления (самостоятельных последовательностей команд, процессов), то и активных объектов может быть несколько. В последовательных системах обычно в каждый момент времени существует только один активный объект, например, главное окно, диспетчер которого ловит и обрабатывает все сообщения. В этом случае остальные объекты пассивны: их поведение проявляется, когда к ним обращается активный объект. В других видах последовательных архитектур нет явного центра активности и управление распределено среди объектов системы, которые могут посылать и принимать сообщения.

Таким образом, объекты не существуют изолированно, а подвергаются воздействию и (или) сами воздействуют на другие объекты.

Поведение — это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений.

Иными словами, поведение объекта — это его наблюдаемая и проверяемая извне деятельность.

Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Например, клиент может активизировать операции append() и pop() для того, чтобы управлять объектом-очередью (добавить или изъять элемент). Существует также операция length(), которая позволяет определить размер очереди, но не может изменить это значение (состояние объекта). В основном понятие *сообщение* совпадает с понятие-

ем *операции* над объектами, хотя механизмы передачи сообщения и вызова операции различны. Пока эти два термина будут использоваться как синонимы.

Передача сообщений — это одна часть уравнения, задающего поведение. Из нашего определения следует, что состояние объекта также влияет на его поведение. Вернемся к торговому автомату продажи напитков. Поведение автомата будет зависеть от его состояния. Если мы опустили в него недостаточную сумму, то останемся без напитка. Если же денег достаточно, автомат выдаст нам желаемое и тем самым изменит свое состояние. Итак, поведение объекта определяется выполняемыми над ним операциями и его состоянием, причем некоторые операции имеют побочное действие: они изменяют состояние. Концепция побочного действия позволяет уточнить наше определение состояния:

Состояние объекта представляет суммарный результат его поведения.

Наиболее интересны те объекты, состояние которых не статично: их состояние изменяется и запрашивается операциями.

Пример. Опишем класс **TQueues** (очереди):

```
class TQueues
        {
            public:
               TQueues();
               virtual  ~ TQueues();
               virtual void Clear();
               virtual  void Append(TObject*  objAItem);
               virtual  TObject* pop();
               virtual  void Remove(Integer intAItemNumber);
               virtual  TObject* First();
               virtual  int Length();
               virtual  boolean IsEmpty();
               virtual  void View();
            private
                        …
          end;
        };
```

Операция **TQueues()** называется конструктором и позволяет создавать объекты (экземпляры класса **TQueues**), **~ TQueues()** — деструктор, напротив, уничтожает в динамической памяти объекты.

Операция **Clear()** предназначена для опустошения очереди; операции **Append()** и **Pop()**, соответственно, осуществляют пополнение и укорачивание очереди; операция **Remove()** удаляет **intAItemNumber-**го очередника; операция **First()** показывает первого очередника; функция **Length()** возвращает длину очереди (количество участников); функция **IsEmpty()** информирует о состоянии очереди (пустая/непустая); наконец, операция **View()** отвечает за просмотр содержимого очереди.

Использование служебного слова **virtual** — виртуальный означает, что мы предоставляем программисту-пользователю нашего класса **TQueues** возможность переопределения указанных операций по своему усмотрению с уче-

том приведенных заголовков операций. Об использовании виртуальных методов речь пойдет в последующих главах.

В определении класса используется обычный для объектного программирования прием ссылки на объекты неопределенного класса с помощью **TObject** (вместо типа **TObject** можно взять тип любого его потомка), благодаря чему в очередь можно вставлять объекты разных классов. Эта техника небезопасна — клиент должен ясно понимать, с каким (какого класса) объектом он имеет дело. Кроме того, сама очередь не «владеет» объектами, которые в нее помещены. Деструктор ~ **TQueues()** уничтожает очередь, но не ее участников. В языках программирования используются параметризованные типы, которые помогают справляться с такими проблемами.

Так как определение **TQueues** задает класс, а не объект, мы должны объявить экземпляры класса, с которыми могут работать клиенты:

TQueues *objQueueA, *objQueueB, *objQueueC;

Добавим моделирующие участников очереди объекты (сотрудники):

TEmployes *objEmploye1, *objEmploye2, *objEmploye3, *objEmploye4;

Теперь, после вызова конструкторов и создания соответствующих объектов:

```
objQueueA=new TQueues();
objQueueB=new TQueues();
objEmploye1=new TEmployes();
objEmploye2=new TEmployes();
objEmploye3=new TEmployes();
```

мы можем выполнять операции над объектами:

```
objQueueA–>Append(objEmployee1);
objQueueA–>Append(objEmployee2);
objQueueA–>Append (objEmployee3);
objQueueB–>Append(objEmployee1);
objQueueB–>Append(objEmployee2);
objQueueB–>Append (objEmployee3);
objEmploye4=TEmployes(objQueueA–>Pop());.
```

В результате очередь **objQueueA** содержит двух сотрудников (первым стоит **objEmployee2**), а очередь **objQueueB** — трех (первым стоит **objEmployee1**). Таким образом, очереди имеют определенное состояние, которое влияет на их будущее поведение, например, первую очередь можно безопасно продвинуть (pop) еще два раза, а вторую — три. Заметим, что в последней строке рассмотренного примера использовано приведение объявленного в операции **Pop**() типа **TObject** к типу **TEmployes** фактического участника очереди **objQueueA**. Обратим также внимание на то, все объекты объявляются как ссылки на фактическую их реализацию, а при использовании объектов применяется операция **–>**, которая вызывает действие, суть которого можно выразить понятиями «разыменование», «переход по адресу», «переход по ссылке».

Операции объекта. Операция — это услуга, которую объект может предоставить своим клиентам. В объектно-ориентированных языках операции,

входящие в определение класса объекта и выполняемые над данным объектом, называются *методами*.

На практике типичный клиент совершает над объектами операции пяти видов.

Две операции являются универсальными; они обеспечивают инфраструктуру, необходимую для создания и уничтожения экземпляров класса:

— конструктор — операция создания объекта и его инициализации;

— деструктор — операция, освобождающая состояние объекта и разрушающая сам объект.

Кроме того, имеются следующие разновидности операций:

— модификатор — операция, которая изменяет состояние объекта;

— селектор — операция, считывающая состояние объекта, но не меняющая состояния;

— итератор — операция, позволяющая организовать доступ ко всем частям объекта в строго определенной последовательности.

Поскольку логика этих операций различна, полезно выработать такой стиль программирования, который учитывает эти различия в коде программы. В нашей спецификации класса **TQueues** мы вначале указали основополагающие операции: конструктор и деструктор (**TQueues()**, **~ TQueues()**), затем перечислили все модификаторы (**Clear**, **Append**, **Pop**, **Remove**), а потом все селекторы (**First, Length**, **IsEmpty**), наконец, приведена операция-итератор (**View**), отвечающая за последовательный просмотр всего содержимого очереди. Рекомендуем взять на вооружение указанный прием, результатом будет явно обозначенная структура класса, которую легче понимать. Заметим, что подобных, основанных на договоренностях рекомендаций по компоновке и оформлению, будет немало. На одну из таких договоренностей уже была ссылка — это правила именования идентификаторов. Все рекомендации появились в результате мирового опыта программирования, на основе многочисленных проб и ошибок. Следование подобным рекомендациям означает, прежде всего, уважение себя и своих коллег.

Роли и ответственность объектов. Совокупность всех методов и свободных процедур, относящихся к конкретному объекту, определяют поведение объекта, охватывающее все его статические и динамические аспекты. В самых нетривиальных абстракциях полезно подразделять общее поведение объекта на частные аспекты поведения, которые мы будет называть ролями. Роль определяет контракт абстракции с ее клиентами в текущий момент времени. Например, торговый автомат может играть роль ожидания монеты.

Объединяя наши определения состояния и поведения объекта, введем понятие ответственности. Ответственность объекта имеет две стороны: знания, которые объект поддерживает, и действия, которые объект может исполнить. Они выражают смысл его предназначения и место в системе. Ответственность понимается как совокупность всех услуг и всех контрактных обязательств объекта. Таким образом, можно сказать: *состояние и поведение объекта определяют исполняемые им роли, а совокупность ролей определяет ответственность данной абстракции*.

Действительно большинство интересных объектов исполняют в своей жизни разные роли, например:

— банковский счет может быть в хорошем или плохом состоянии (две роли), и от текущей роли зависит, что произойдет при попытке снятия с него денег;

— для фондового брокера пакет акций — это товар, который можно покупать или продавать, а для юриста пакет акций — это знак обладания определенными правами;

— в течение дня одна и та же персона может играть роль матери, врача, садовника и критика популярного сериала.

Роли банковского счета являются динамическими и взаимоисключающими. Роли пакета акций слегка перекрываются, но каждая из них зависит от того, что клиент с ними делает. В случае персоны роли динамически изменяются каждую минуту.

Как будет видно далее, мы часто будем начинать наш анализ перечислением разных ролей, которые может играть объект. Во время проектирования автоматизированной системы мы будем выделять эти роли, вводя конкретные операции, выполняющие ответственности каждой роли.

Контрольные вопросы

1. Что считается поведением объекта?
2. В чем заключается активное и пассивное поведение объекта?
3. Что является операцией объекта?
4. Какой термин принято использовать для обозначения операций объекта?
5. Перечислите пять видов операций и сформулируйте назначение каждого вида операций.
6. Что считается ролью объекта?
7. Что считается ответственностью объекта?
8. Приведите пример(ы) объекта(ов), демонстрирующий(ие) понятия свойства, состояния, поведения, роли, ответственности объекта.

1.1.4. Идентичность объекта

Рассмотрим следующее определение: *Идентичность — это такое свойство объекта, которое отличает его от всех других объектов.*

Идентичность объектов обусловлена их динамической природой. Понятие идентичности объекта нельзя отождествлять с понятием имени объекта. Рассмотрим пример:

TAnyClass *Object1, *Object2;

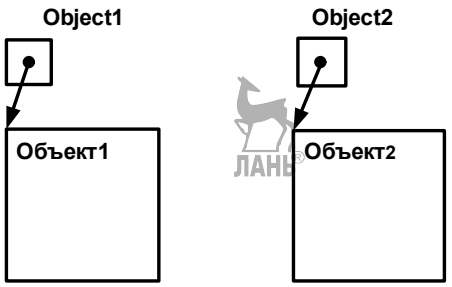
В данном определении введены имена двух объектов. В результате трансляции введенным именам будут сопоставлены ячейки памяти, в которые можно записать ссылку на объект:

Сами объекты могут появиться только с помощью конструктора:

Object1=new TAnyClass(); Object2=new TAnyClass();,

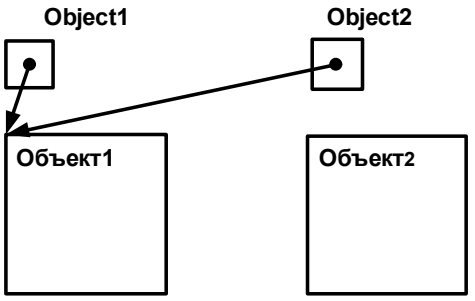
где **TAnyClass()** — конструктор нашего класса **TAnyClass**.

В результате мы получим следующие структуры:



Теперь идентифицировать объект (получить доступ к объекту) можно с помощью имени объекта. Выполним присваивание **Object2=Object1**;.

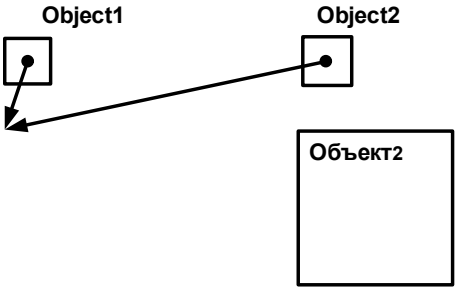
В результате имеем структуры



Теперь «Объект1» идентифицируется двумя именами: **Object1** и **Object2**, а «Объект2» потерял свою идентичность (стал недоступен). К потере идентичности объекта приводит также использование деструктора. Если в последнем случае выполнить операцию

Object2 –> ~TAnyClass ();,

то оба имени (**Object1**, **Object2**) станут недействительными и оба объекта потеряют идентичность. Кроме того, в динамической памяти останется «мусор» в виде недоступного (потерявшего идентичность) объекта «Объект2»:



Замечание. Обращаем внимание, что значения ссылок в ячейках **Object1**, **Object2,** как правило, деструктором не изменяются, но, как видно из последнего рисунка, они становятся недействительными.

Контрольные вопросы

1. Что считается идентичностью объекта?
2. Каким образом осуществляется идентификация объекта?
3. Какие неприятности возможны при потере идентичности объекта?

1.2. Отношения между объектами

1.2.1. Типы отношений

Сами по себе объекты не представляют большого интереса: только в процессе взаимодействия объектов реализуется система. Мы получаем сообщество хорошо воспитанных объектов, которые вежливо просят друг друга об услугах. Самолет, по определению, — «совокупность элементов, каждый из которых по своей природе стремится упасть на землю, но за счет совместных непрерывных усилий они преодолевают эту тенденцию». Самолет летит только благодаря согласованным действиям своих компонентов-объектов. Не будем забывать при этом об участии человека.

Отношения двух любых объектов основываются на знаниях, которыми один обладает относительно другого: об операциях, которые можно выполнять, и об ожидаемом поведении. Особый интерес для объектно-ориентированного анализа и проектирования представляют два типа иерархических отношений объектов:

— отношение связи (взаимодействия);
— отношение агрегации.

Контрольные вопросы

1. Как вы представляете себе понятие «отношение между объектами»?
2. Какие виды отношений вызывают особый интерес?

1.2.2. Отношение связи

Понятие связи означает физическое или концептуальное взаимодействие между объектами. Объект сотрудничает с другими объектами через связи, соединяющие его с ними. Другими словами, связь — это специфическое сопоставление, через которое объект-клиент запрашивает услугу у объекта-сервера или через которое один объект находит путь к другому.

Пример. Во многих промышленных процессах требуется непрерывное изменение температуры. Необходимо поднять температуру до заданного значения, выдержать заданное время и понизить до нормы. Режим изменения температуры у разных процессов разный: зеркало телескопа надо охлаждать очень медленно, а закаляемую сталь очень быстро.

Абстракция охлаждения имеет достаточно четкое поведение, что дает нам право на описание такого класса.

Опишем класс **TTemperatureRamps** (охлаждение), который по смыслу задает функцию времени от температуры:

```
class  TTemperatureRamps
        {
           public:
                      TTemperatureRamps();
                       virtual  ~ TTemperatureRamps ();
                       virtual void Clear();
                       virtual void Bind(double  dubATemperature,  int intAMinute);
                       virtual double TemperatureAt(int intAMinute);
           protected:
                        …
         };
```

Выдерживая наш стиль, мы описали некоторые из операций виртуальными, так как ожидаем, что этот класс будет иметь подклассы, в которых реализация виртуальных операций может быть уточнена.

На самом деле в смысле поведения нам надо нечто большее, чем просто зависимость температуры от времени. Пусть, например, известно, что на 60-й минуте должна быть достигнута температура 150°C, а на 180-й — 90°C. Для ввода данных будем использовать функцию **Bind()**. Теперь нам нужно узнать, какой должна быть температура на 120-й минуте. Это требует линейной интерполяции, так что требуемое от абстракции поведение усложняется. Для получения таких сведений введена функция **TemperatureAt()**.

Вместе с тем, мы не требуем от абстракции охлаждения возможности управления нагревателем, поддерживающим требуемый режим остывания[2]. Мы предпочитаем разделение понятий, при котором нужное поведение достигается взаимодействием трех объектов: режима остывания (экземпляра класса **TTemperatureRamps**), нагревателя (экземпляра класса **THeaters**) и управляющего органа (экземпляра класса **TRampControllers**). Класс **TRampControllers** можно определить так:

```
class  TRampControllers
        {
            public:
            TRampControllers():
             ~ TRampControllers();
             void Process(TTemperatureRamps  objATemperatureRamp);
             int Schedule(TTemperatureRamps objATemperatureRamp);
      private:
                …
};
```

Операция **Process()** обеспечивает основное поведение этой абстракции; ее назначение — передать график изменения температуры нагревателю, установленному в конкретном месте.

[2] Предполагается, что в процессе остывания нужная температура поддерживается периодическим включением нагревателя. Таким образом, мы не даем возможность нашему изделию остывать слишком быстро.

Например, объявим:

```
TTemperatureRamps *  objVTemperatureRamp;
TRampControllers *  objVRampController;.
```

Теперь, построив объекты, задав пару точек, определим режим остывания, загрузим его в контроллер и запустим процесс:

```
objVTemperatureRamp –> Bind(150, 60);
objVTemperatureRamp –> Bind(90, 180);
objVRampController –> Process(objTemperatureRamp);.
```

В этом примере **objVRampController** — некоторый агент, ответственный за выполнение температурного плана, использует объект **objVTemperatureRamp** как специалиста по планированию температурного режима (сервер). Эта связь проявляется в том, что **objVRampController** явно получает **objVTemperatureRamp** в качестве параметра одной из своих операций.

Сделаем одно замечание по поводу нашего стиля. На первый взгляд может показаться, что наши абстракции — лишь объектные оболочки для элементов, полученных в результате обычной функциональной декомпозиции. Пример операции **Schedule()** (расписание) показывает, что это не так. Объекты класса **TRampControllers** имеют достаточно интеллекта, чтобы определять расписание для конкретных режимов остывания, и мы включаем эту операцию как дополнительное поведение нашей абстракции. В некоторых энергоемких технологиях (например, плавка металлов) можно существенно выиграть, если учитывать остывание установки и тепло, остающееся после предыдущей плавки. Поскольку существует операция **Schedule()**, клиент может запросить объект **objVRampController**, чтобы тот рекомендовал оптимальный момент запуска следующего нагрева.

Участвуя в связи, объект может выполнять одну из следующих трех ролей: актер, сервер, агент.

Актер (Actor) — это деятель, исполнитель; а исполнитель ролей — это и есть актер. Объект может воздействовать на другие объекты, но сам никогда не подвергается воздействию других объектов, это соответствует понятию «активный объект».

Сервер. Объект может только подвергаться воздействию со стороны других объектов, но он никогда не выступает в роли воздействующего объекта, соответственно, сервер является пассивным объектом.

Агент — это объект, который может выступать как в активной, так и в пассивной роли; как правило, объект-агент создается для выполнения операций в интересах какого-либо объекта-актера или объекта-агента.

Сервер предоставляет некоторые услуги, в которых нуждаются и которые запрашивают актеры и агенты. Поэтому, в противоположность серверу, актеров и агентов часто называют *клиентами*.

Видимость. Пусть есть два объекта ***A*** и ***B***. Чтобы ***A*** смог установить связь с объектом ***B***, надо, чтобы ***B*** был, в каком-то смысле, видим для ***A***. Мы можем не заботиться об этом на стадии анализа, но когда дело доходит до реализации системы, мы должны обеспечить видимость связанных объектов.

В предыдущем примере объект **objVRampController** видит объект **objVTemperatureRamp**, поскольку оба они объявлены в одной области видимости и потому, что **objVTemperatureRamp** передается объекту **objVRampController** в качестве параметра. В принципе есть следующие четыре способа обеспечить видимость:

— сервер глобален по отношению к клиенту;

— сервер (или указатель на него) передан клиенту в качестве параметра операции;

— сервер является частью клиента;

— сервер локально порождается клиентом в ходе выполнения какой-либо операции.

Какой именно из этих способов выбрать — зависит от тактики проектирования.

Синхронизация. Когда один объект посылает по связи сообщение другому, связанному с ним, они, как говорят, синхронизируются. В строго последовательном приложении синхронизация объектов состоит в запуске метода. Однако в многопоточной (многопроцессорной) системе объекты требуют более изощренной схемы передачи сообщений, чтобы разрешить проблемы взаимного исключения, типичные для параллельных систем. Активные объекты сами по себе выполняются как потоки, поэтому присутствие других активных объектов на них обычно не влияет. Если же активный объект имеет связь с пассивным объектом, возможны следующие три подхода к синхронизации:

— последовательный — функционирование пассивного объекта обеспечивается в присутствии только одного активного процесса, в этом случае синхронизация вырождается;

— защищенный — функционирование пассивного объекта обеспечивается в присутствии многих потоков управления, но активные клиенты сами должны договориться и обеспечить взаимное исключение (например, цикл работы активного объекта с пассивным может быть таким: заблокировать пассивный объект от доступа других активных объектов, получить услугу, разблокировать пассивный объект);

— синхронный — функционирование пассивного объекта обеспечивается в присутствии многих потоков управления; взаимное исключение обеспечивает сам пассивный объект (сервер).

В качестве примера защищенного подхода приведем распространенный прием синхронизации в случае использования коллективного ресурса:

Printer –> Open(); **…** ; **Printer –> Close()**; или
Clipboard –> Open(); **…** ; **Clipboard –> Close()** или
GlobalLock(h); **...** ; **GlobalUnLock(h)**.

В приведенных примерах **Open()**, **GlobalLock(h)** означают «заблокировать объект от одновременного обращения других клиентов». Многоточием обозначено выполнение над объектом (принтером, буфером обмена, памятью) необходимых нашему приложению действий. Наконец, **Close()**, **GlobalUnLock(h)** осуществляют разблокирование коллективного ресурса, который становится дос-

тупным другим приложениям. Обратим внимание на неудачное использование понятий: **Open** (открывать) обозначает блокирование (закрытие) постороннего доступа, **Close** (закрывать) обозначает разблокирование (открытие) постороннего доступа. Использование термина **Lock** (запирать) надо признать более удачным.

Контрольные вопросы

1. Как понимается понятие «отношение связи» между объектами?
2. Приведите пример, иллюстрирующий отношение связи между объектами.
3. Какие роли может исполнять объект, участвуя в связи?
4. Какие существуют способы обеспечения видимости объектов?
5. Как вы понимаете понятие «синхронизация объектов»?
6. Какие существуют подходы к обеспечению синхронизации объектов?

1.2.3. Отношение агрегации

В то время как связи обозначают равноправные или «клиент-серверные» отношения между объектами, агрегация определяет отношения целого и части, приводящие к соответствующей иерархии объектов, причем, идя от целого (агрегата), мы можем прийти к его частям. На рисунке 1.1 объект **objRampController** имеет два отношения: связь с объектом **objTemperatureRamp** и атрибут **h** класса **THeaters** (нагреватель). В данном случае **objRampController** — целое, а **h** — его часть. Другими словами, **h** — часть состояния **objRampController**. Исходя из **objRampController**, можно найти соответствующий нагреватель. Однако по **h** нельзя найти содержащий его объект (называемый также его контейнером), если только сведения о нем не являются случайно частью состояния **h**.

Агрегация может означать физическое вхождение одного объекта в другой, но не обязательно. Самолет состоит из крыльев, двигателей, шасси и прочих частей. С другой стороны, отношения акционера с его акциями — это агрегация, которая не предусматривает физического включения. Акционер монопольно владеет своими акциями, но они в него не входят физически. Это, несомненно, отношение агрегации, но скорее концептуальное, чем физическое по своей природе. Аналогичными примерами являются пары: автомобиль и шофер; самолет и пилот.

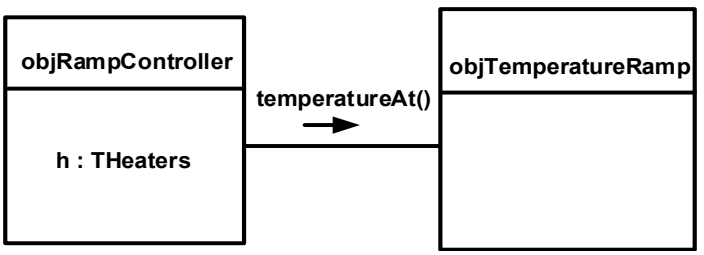


Рис. 1.1
Агрегация

Выбирая одно из двух — связь или агрегацию, — надо иметь в виду следующее. Агрегация иногда предпочтительнее, поскольку позволяет скрыть час-

ти в целом. Иногда, наоборот, предпочтительнее связи, поскольку они слабее и менее ограничительные. Принимая решение, надо взвесить все.

Объект, являющийся атрибутом другого объекта (агрегата), имеет связь со своим агрегатом. Через эту связь агрегат может посылать ему сообщения. При добавлении в спецификацию класса **TRampControllers** описания нагревателя:

h : THeaters;

каждый объект **objRampController** будет иметь свой нагреватель.

Замечание. Мы должны создавать нагреватель при создании нового контроллера, так как сам этот класс не предусматривает вызова конструктора нагревателя по умолчанию. Противоположный подход используется в потомках класса **TComponent**, в котором операции создания и ликвидации подчиненных объектов автоматизированы и выполняются одновременно с соответствующими операциями агрегата.

ЛАНЬ®

Контрольные вопросы

1. Сформулируйте своими словами понятие «отношение агрегации» между двумя объектами.
2. Приведите примеры объектов, между которыми существуют отношения связи и агрегации.
3. В чем заключается смысловое отличие физического и концептуального вхождения одного объекта в другой?

1.3. Классы

1.3.1. Понятие класса

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие этих двух понятий. В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию существенного в объекте. Таким образом, можно говорить о классе «Млекопитающие», который включает характеристики, общие для всех млекопитающих. Для указания на конкретного представителя млекопитающих (кот Тютька) необходимо сказать «кот Тютька — млекопитающее».

Использование термина «класс» в объектном программировании заимствовано, по-видимому, из области «классификация». Напомним, классификация — это осмысленный порядок вещей, явлений, разделение их на разновидности согласно каким-либо важным признакам. Классификация предназначена для постоянного использования в какой-либо науке или области практической деятельности (например, классификация животных и растений). Обычно в качестве основания деления в классификации выбирают признаки, существенные для данных предметов. В этом случае классификация (называемая естественной) выявляет существенные сходства и различия между предметами и имеет познавательное значение. В других случаях, когда цель классификации состоит лишь в систематизации предметов, в качестве основания выбираются признаки, удобные

для целей классификации, но несущественные для самих предметов (например, алфавитные каталоги). Такие классификации называют искусственными.

В общепонятных терминах можно дать следующее определение класса: группа, множество или вид с общими свойствами или общим свойством, разновидностями, отличиями по качеству, возможностями или условиями. В контексте объектно-ориентированного анализа дадим следующее определение класса: *Класс — это множество объектов, имеющих общую структуру и общее поведение.*

Любой конкретный объект является просто экземпляром класса. Что же не является классом? Объекты, не связанные общностью структуры и поведения, нельзя объединить в класс, так как по определению они не связаны между собой ничем, кроме того, что все они объекты[3].

Класс в объектно-ориентированном программировании воплощается как абстрактный тип данных. В главе 3 показано, что объекты реальной жизни мы стараемся объединять в группы со сходными признаками, а группировка объектов и есть классификация. Теория абстрактных типов данных (глава 4) является теоретическим фундаментом построения системы классов. Перечислим основные принципы, заложенные в теорию абстрактных типов данных:

1) вводится понятие основы, которой можно сопоставлять некоторые множества;

2) вводятся операции, которые делятся на два вида — конструкторы и анализаторы; с помощью конструкторов мы можем формально построить элементы сопоставляемого основе множества; с помощью анализаторов мы можем сделать оценку свойств элементов сопоставляемого основе множества;

3) вводится описание свойств сопоставляемого основе множества; с помощью свойств мы выделяем интересующие нас подмножества;

4) вводятся правила построения иерархической системы абстрактных типов данных.

Наконец, теория абстрактных типов данных представляет собой формальную систему, поэтому все перечисленные принципы записываются точно и реализуются однозначно.

Понятия «абстрактный тип данных» и «класс» в данной ситуации мы будем считать тождественными.

Важно отметить, что классы, как их понимают в большинстве существующих языков программирования, необходимы, но недостаточны для декомпозиции сложных систем. Некоторые абстракции так сложны, что не могут быть выражены в терминах простого описания класса. Например, пусть имеются выраженные в виде классов абстракции: графический интерфейс пользователя, база данных и система учета. Можно попытаться интегрировать перечисленные абстракции и построить объединенный класс, но повторной используемости и возможности наследования не получится: маловероятно, что заложенная в такой класс совокуп-

[3] В этой фразе имеется противоречие, так как мы можем создать класс «Объекты». Авторы систем программирования так и поступают: вводят класс **TObject**, в котором сосредотачивают все свойства «быть объектом», а все другие классы являются потомками класса **TObject**.

ность свойств понадобится в другом приложении. Строго говоря, то же самое в нашем примере можно сказать и про отдельные абстракции. Например, иметь универсальный и громоздкий интерфейс — плохая практика, так как большинству клиентов необходима только малая его часть. Более того, изменение одной части этого гигантского интерфейса требует обновления каждого из клиентов, независимо от того, затронуло ли его это изменение по сути. Вложенность классов не устраняет этих проблем, а только откладывает их. Лучше считать их некими совокупностями (кластерами) сотрудничающих классов. Будем называть такие кластеры категориями классов. Определение уровня абстракции, на котором интеграция переходит от классов к категориям классов, является очень нетривиальной задачей проектировщика. В качестве основного критерия можно взять потребность в повторной используемости создаваемого класса.

Примеры определения классов приводились ранее. Подробнее синтаксис и семантика определения класса будут рассмотрены во второй части книги.

Контрольные вопросы

1. Сформулируйте своими словами понятие «класс».
2. Чем принципиально отличаются объект и класс?
3. Вспомните основные принципы, заложенные в теорию абстрактных типов данных.
4. Какой смысл вкладывается в понятие «категория классов»?

1.3.2. Интерфейс и реализация

Рассмотрим идею контрактного программирования: большие задачи надо разделить на много маленьких и перепоручить их мелким субподрядчикам. Нигде эта идея не проявляет себя так ярко, как в проектировании и реализации классов.

По своей природе класс — это генеральный контракт между абстракцией и всеми ее клиентами. Выразителем обязательств класса служит его интерфейс, причем в языках с сильной типизацией потенциальные нарушения контракта можно обнаружить уже на стадии компиляции.

Идея контрактного программирования приводит нас к разграничению внешнего облика (интерфейса) и внутреннего устройства (реализации) класса. Главное в интерфейсе — объявление операций, поддерживаемых экземплярами класса. К нему можно добавить объявления других классов, переменных, констант и исключительных ситуаций, уточняющих абстракцию, которую класс должен выражать. Напротив, реализация класса никому, кроме него самого, не интересна. По большей части реализация состоит в уточнении операций, объявленных в интерфейсе класса.

В основе разделения описания класса на две части используется уже известный прием абстрагирования: существенная для пользователя сторона описания класса находится в интерфейсе, а несущественная — в реализации. При этом интерфейс класса видим пользователям класса, а реализация класса — скрыта. Внутри интерфейса также имеются изобразительные средства, позволяющие более тщательно регулировать способ видимости содержимого класса. Мы можем разбить интерфейс класса на три части:

— открытую (**public**) — видимую всем клиентам без ограничений;

— защищенную (**protected**) — видимую самому классу и его подклассам (доверенным лицам);

— закрытую (**private**) — видимую только самому классу.

Разные языки программирования предусматривают различные комбинации этих частей. Разработчик может задать права доступа к той или иной части класса, определив тем самым зону видимости клиента. В частности, в C++ все три перечисленных уровня доступа определяются и используются явно.

Рассмотрим пример описания и реализации класса.

```
class TAnyClass
    {
      private: int k;              // закрытая переменная k
      public:
      void Set(int new);          // объявление открытой функции Set
    };
void TAnyClass::Set(int new) {if (k > 10) k=new;};
```

В рассмотренном примере интерфейсная часть класса представлена содержимым раздела **public.** Внешние пользователи экземпляра данного класса имеют доступ только к функции **Set**. Реализация класса заключается в реализации функции **Set** и реализации недоступной пользователю переменной **k**. Реализация функции **Set** может располагаться отдельно от описания класса. Чтобы корректно осуществить указанную возможность, необходимо перед именем функции указать имя класса, который является хозяином данной функции. В нашем примере получилась конструкция **TAnyClass::Set**, разделитель **::** можно считать специальной операцией, часто называемой операцией привязки.

Состояние объекта задается в его классе путем определения констант или переменных, помещаемых в его защищенной или закрытой части. Тем самым свойства объекта инкапсулированы (помещены в оболочку), и изменения значений свойств осуществляются клиентом не напрямую, а посредством вызова методов. Инкапсуляция понимается как размещение свойств и методов класса в обособленном месте, в некоторой замкнутой оболочке, в которую одним клиентам разрешается заглянуть, а другим — нет.

В абстрактных типах данных (глава 4) определение типа данных задается модулем с двумя параметрами: описание теории, включающей сигнатуру и свойства, и описание реализации. В языках программирования интерфейс класса содержит только сигнатуру, а свойства типа данных должны обеспечиваться реализацией.

Внимательный читатель может спросить, почему же представление объекта определяется в интерфейсной части класса, а не в его реализации. Причины чисто практические: в противном случае понадобились бы объектно-ориентированные процессоры или очень хитроумные компиляторы. Когда компилятор обрабатывает объявление объекта, например такое:

TDisplayItem objItem;

он должен знать, сколько памяти под него отвести. Если бы эта информация содержалась в реализации класса, нам пришлось бы написать ее полностью, прежде чем мы смогли бы задействовать его клиентов. То есть весь смысл отделения интерфейса от реализации был бы потерян.

Свойства, составляющие представление класса, известны под разными именами. В большинстве языков программирования их называют полями.

Контрольные вопросы

1. Объясните причину деления класса на две части — интерфейс и реализацию.
2. Какие способы видимости содержимого класса используются в языках программирования?
3. Подумайте, какие преимущества и какие недостатки присутствуют в отделении интерфейса класса от его реализации.

1.4. Отношения между классами

1.4.1. Типы отношений

Рассмотрим сходства и различия между следующими классами: цветы, пионы, красные розы, желтые розы, лепестки и божьи коровки. Мы можем заметить следующее:

— пион — цветок;
— роза — (другой) цветок;
— красная и желтая розы — розы;
— лепесток является частью обоих видов цветков;
— божьи коровки питаются вредителями, поражающими некоторые цветы.

Из этого простого примера следует, что классы, как и объекты, не существуют изолированно. В каждой проблемной области ключевые абстракции взаимодействуют многими интересными способами, что мы и должны отразить в программном проекте.

Отношения между классами могут означать одно из двух. Во-первых, у них может быть что-то общее. Например, и пионы, и розы — это разновидности цветов: и те, и другие имеют ярко окрашенные лепестки, испускают аромат и т. д. Во-вторых, может быть какая-то семантическая (смысловая) связь. Например, красные розы больше похожи на желтые розы, чем на пионы. Но между розами и пионами больше общего, чем между цветами и лепестками. Также существует симбиотическая (приносящая взаимную пользу) связь между цветами и божьими коровками: божьи коровки защищают цветы от вредителей, которые, в свою очередь, служат пищей божьим коровкам.

Существует три основных вида отношений между классами. Во-первых, это отношение «обобщение/специализация» (общее и частное, «is-a»). Розы суть цветы, что значит: розы являются специализированным частным случаем, подклассом более общего класса «цветы». Во-вторых, это отношение «целое/часть» («part of»). Так, лепестки являются частью цветов. В-третьих, это семантические, смысловые отношения, ассоциации. Например, божьи коровки

ассоциируются с цветами — хотя, казалось бы, что у них общего. Или вот: розы и свечи — и то, и другое можно использовать для украшения стола. Существует большое разнообразие семантических отношений между классами.

Языки программирования выработали несколько общих подходов к выражению отношений этих трех типов. В частности, большинство объектно-ориентированных языков непосредственно поддерживают разные комбинации следующих видов отношений:

— ассоциация;
— наследование;
— агрегация;
— зависимость;
— реализация (инстанцирование);
— метакласс.

Мы ограничимся здесь только перечислением отношений между классами. Обсуждению отношений посвящены следующие главы.

Контрольные вопросы

1. Какие основные три вида отношений имеются между классами?

1.4.2. Отношение ассоциации

Из шести перечисленных видов отношений наиболее общим и неопределенным является ассоциация. Обычно системный аналитик констатирует наличие ассоциации и, постепенно, уточняя проект, превращает ее в другую, более специализированную связь.

Пример. Желая автоматизировать розничную торговую точку, мы обнаруживаем, кроме прочих, две абстракции — товары и продажи. На рисунке 1.2 показана ассоциация, которую мы при этом усматриваем. Класс **TArticle** — это то, что мы продали в некоторой сделке, а класс **TSale** — сама сделка, в которой продано несколько товаров. Надо полагать, ассоциация работает в обе стороны: задавшись товаром, можно выйти на сделку, в которой он был продан, а пойдя от сделки, найти, что было продано.

Вот две выдержки из объявления соответствующих классов:

```
class  TSale;
class  TArticle
{
    …
    protected:
        TSale *  objLastSale;
}
 class TSale
{
   …
   protected
        TArticle *  objPartSold;
}
```

Это ассоциация вида «один-ко-многим»: каждый экземпляр товара относится только к одной последней продаже, в то время как каждый экземпляр **TSale** может указывать на совокупность проданных товаров.

Рис. 1.2
Ассоциация

Как показывает этот пример, ассоциация — смысловая связь. По умолчанию она не имеет направления (если не оговорено противное, ассоциация, как в данном примере, подразумевает двухстороннюю связь) и не объясняет, как классы общаются друг с другом (мы можем только отметить семантическую зависимость, указав, какие роли классы играют друг для друга). Однако именно это нам требуется на ранней стадии анализа. Итак, мы фиксируем участников, их роли и мощность отношения.

В последнем примере мы имели ассоциацию «один ко многим». Тем самым мы обозначили ее мощность (т. е., грубо говоря, количество участников). На практике важно различать три случая мощности ассоциации:

— один-к-одному;
— один-ко-многим;
— многие-ко-многим.

Отношение «один-к-одному» обозначает очень узкую ассоциацию. Например, в розничной системе продаж примером могла бы быть связь между классом **TSale** (сделка) и классом **TCardTransaction** (операция с карточным счетом): каждая продажа соответствует ровно одному снятию денег с карточного счета. Отношения «многие-ко-многим» тоже нередки. Например, каждый объект класса **TCustomer** (покупатель) может обратиться к нескольким объектам класса **TSalePerson** (продавец), и каждый продавец может взаимодействовать с несколькими покупателями. Все три вида мощности имеют разного рода вариации.

Контрольные вопросы

1. Что представляет собой отношение ассоциации?
2. Какие три вида мощности ассоциации различаются в практике?

1.4.3. Отношение наследования

Наследование является отношением вида «обобщение/специализация».

Понятие наследования рассмотрим на примерах. Находящиеся в полете космические зонды посылают на наземные станции информацию о состоянии своих основных систем (например, источников энергоснабжения и двигателей) и измерения датчиков (таких как датчики радиации, масс-спектрометры, телекамеры, фиксаторы столкновений с микрометеоритами и т. д.). Вся совокупность пере-

даваемой информации называется телеметрическими данными. Как правило, они передаются в виде потока данных, состоящего из заголовка (включающего временные метки и ключи для идентификации последующих данных) и нескольких пакетов данных от подсистем и датчиков. Предположим, что анализ требований к системе обусловил наличие нескольких сотен разновидностей телеметрических данных, включающих параметры в разных контрольных точках системы. Очевидно, что описание такого количества отдельных структур будет избыточным как из-за повторяемости структур, так и из-за наличия общих функций обработки.

Значительно лучше построить иерархию классов, в которой от общих (родительских) классов с помощью наследования образуются более специализированные (дочерние) классы.

Дочерний класс может унаследовать структуру и поведение родительских классов.

Поэтому для описания каждого типа данных телеметрии сами собой напрашиваются структуры:

```
class TTelemetryDatas
    {
        public:
            TTelemetryDatas();
            virtual  ~TTelemetryDatas();
            virtual void Transmit();
             TTime CurrentTime();
        protected:
             int intIdent ;
            TTime timeStamp;
    }.
```

В этом примере введен класс, имеющий конструктор, деструктор (который наследники могут переопределить) и методы **Transmit** (передать) и **CurrentTime** (текущее время), видимые для всех клиентов. Защищенные элементы **intIdent** (код передаваемых данных) и **timeStamp** (показание времени) доступны только классу и его подклассам. Заметим, что функция **CurrentTime** сделана открытой, в которой по договоренности значение **timeStamp** можно читать (но не изменять).

Теперь рассмотрим класс **TElectricalDatas**:

```
class TElectricalDatas : public TTelemetryDatas
            {
               public:
                    TElectricalDatas();
                    ~ TElectricalDatas();
                    virtual void Transmit();
                    double CurrentPower();
               protected:
                    double dubFuelCell1Voltage();
                    double dubFuelCell2Voltage();
                    double dubFuelCell1Amperes();
                    double dubFuelCell2Amperes();
            }.
```

Этот класс — наследник класса **TTelemetryDatas** (об этом говорится в первой строке описания класса с помощью выражения**: public TTelemetryDatas**). Класс **TElectricalDatas** наследует (включает) все содержимое класса-предка **TTelemetryDatas**, но исходная структура дополнена конструктором **TElectrical-Datas**, деструктором ~**TElectricalDatas**, четырьмя новыми элементами, а поведение — переопределено (изменена функция **Transmit**). Кроме того, добавлена функция **CurrentPower** (текущее напряжение).

Таким образом, с учетом содержимого класса-предка **TTelemetryDatas** транслятор преобразует исходное описание класса **TElectricalDatas** к следующему виду:

```
class TElectricalDatas : public TTelemetryDatas
        {
          public:
               TTelemetryDatas();    // конструктор унаследован
               virtual  ~TTelemetryDatas(); //деструктор унаследован
               TTime CurrentTime();        // метод унаследован
          protected:
               int intIdent ;              // поле унаследовано
               TTime timeStamp;            // поле унаследовано
           public:
                TElectricalDatas();        // конструктор введен
                ~ TElectricalDatas();      // деструктор введен
                virtual void Transmit(); // одноименный метод переопределен
                double CurrentPower();           // метод введен
           protected:
                double dubFuelCell1Voltage();    // метод введен
                double dubFuelCell2Voltage();    // метод введен
                double dubFuelCell1Amperes();    // метод введен
                double dubFuelCell2Amperes();    // метод введен
        }.
```

Попросту говоря, наследование — это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов. Класс, структура и поведение которого наследуются, называется суперклассом. Так, **TTelemetryDatas** является суперклассом для **TElectricalDatas**. Производный от суперкласса класс называется подклассом. Это означает, что наследование устанавливает между классами иерархию общего и частного. В этом смысле **TElectricalDatas** является более специализированным классом более общего **TTelemetryDatas**. Мы уже видели, что в подклассе структура и поведение исходного суперкласса дополняются и переопределяются. Обратим внимание на то, что у подкласса объем свойств и операций, как правило, больше объема свойств и операций соответствующего суперкласса.

Наличие механизма наследования является отличительной чертой объектно-ориентированных языков.

Одиночное наследование. Подкласс обычно расширяет или ограничивает существующую структуру и поведение своего суперкласса. Например, подкласс

TGuardedQueues («бдительная» очередь) может добавлять к поведению суперкласса **TQueues** (очередь) операции, которые защищают состояние очереди от одновременного изменения несколькими независимыми потоками. Обратный пример: подкласс **TUnselectableDisplayItems** может ограничить поведение своего суперкласса **TDisplayItems**, запретив выделение объекта на экране. Часто подклассы делают и то, и другое, т. е. расширяют и ограничивают родительский класс одновременно.

Отношения одиночного наследования от суперкласса **TTelemetryDatas** показаны на рисунке 1.3. Стрелки обозначают отношения частного к общему. Например, **TCameraDatas** — это разновидность класса **TSensorDatas**, который в свою очередь является разновидностью класса **TTelemetryDatas**. В главе 3 мы покажем, что правильная организация иерархии абстракций — это вопрос логической классификации.

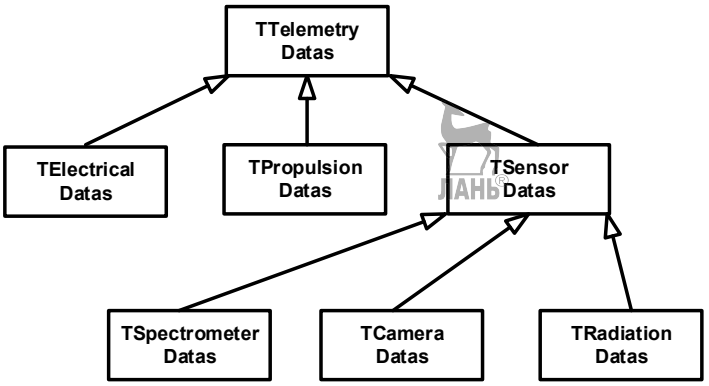


Рис. 1.3
Одиночное наследование

Можно ожидать, что для некоторых классов на рисунке 1.3 будут созданы экземпляры, а для других — нет. Наиболее вероятно образование объектов самых *специализированных классов*, например, **TElectricalDatas** и **TSpectrometerDatas** (такие классы называют конкретными классами, или листьями иерархического дерева). Образование объектов из классов, занимающих промежуточное положение (**TSensorDatas** или тем более **TTelemetryDatas**), менее вероятно. Классы, экземпляры которых не создаются, называются абстрактными. Ожидается, что подклассы абстрактных классов доопределят их до жизнеспособной абстракции, наполняя класс содержанием. В чем же смысл наследования? В повторном использовании! В нашем примере (рис. 1.3) содержимое класса **TTelemetryDatas** повторно используется во всех других классах показанной иерархии. Описание класса **TTelemetryDatas**, появившись однократно, будет скопировано шесть раз с помощью механизма наследования в другие классы, аналогичным образом содержимое класса **TSensorDatas** скопируется три раза. Заметим, что это копирование не механическое: каждая копия предка будет обладать поведением, отвечающим потребностям соответствующего потомка.

Самый общий класс в иерархии классов называется базовым. В большинстве приложений базовых классов бывает несколько, и они отражают наиболее

общие абстракции предметной области. На логическом уровне рассмотрения хорошо сделанная структура классов — это скорее лес из деревьев наследования, чем одна многоэтажная структура наследования с одним корнем. Но в некоторых языках программирования определен базовый класс самого верхнего уровня, который является единым суперклассом для всех остальных классов. В языке C++Builder эту роль играет класс **TObject**, который обеспечивает всех своих потомков свойством «быть классом, с помощью которого можно построить объект». Таким образом, класс **TObject**, являясь суперклассом для всех других классов, обеспечивает все построенные объекты обязательным набором свойств и поведением «быть классом, быть объектом».

У класса обычно бывает два вида пользователей:

— экземпляры класса;

— подклассы.

Часто полезно иметь для них разные интерфейсы. В частности, если мы хотим показать только внешне видимое поведение для экземпляров класса, то нам нужно открыть служебные функции и представления подклассам. Этим объясняется наличие открытой (**public**), защищенной (**protected**) и закрытой (**private**) частей описания класса в языке C++: разработчик может четко разделить, какие элементы класса доступны для экземпляров класса, а какие для подклассов.

Есть серьезные противоречия между потребностями наследования и инкапсуляции. В значительной мере наследование открывает наследующему классу некоторые секреты своего поведения. На практике, чтобы понять, как работает какой-то класс, часто надо изучить все его суперклассы.

Наследование подразумевает, что подклассы повторяют структуры их суперклассов. В предыдущем примере экземпляры класса **TElectricalDatas** содержат элементы структуры суперкласса (**intIdent** и **timeStamp**) и более специализированные элементы (**dubFuelCell1Voltage, dubFuelCell2Voltage, dubFuelCell1Amperes, dubFuelCell2Amperes** — напряжение и ток в двух топливных камерах).

Поведение суперклассов также наследуется. Применительно к объектам класса **TElectricalData** можно использовать операции **CurrentTime** (унаследована от суперкласса), **CurrentPower** (определена в классе) и **Transmit** (переопределена в подклассе). В большинстве языков программирования допускается не только наследование методов суперкласса, но также добавление новых и переопределение существующих методов. В C++ любой метод суперкласса можно переопределить в подклассе.

Метод, объявленный виртуальным (функция **Transmit** в предыдущем примере), может быть в подклассе переопределен, а остальные (**CurrentTime**) — нет[4].

Одиночный полиморфизм. Пусть процедура **Transmit** класса **TTelemetryDatas** реализована следующим образом:

[4] У нас всегда остается возможность написать метод **CurrentTime** заново, для этого случая лучше использовать термин «перекрытие». Отличия переопределения и перекрытия подробно будут рассмотрены во второй части.

```
void TTelemetryDatas::Transmit()
 {
    TransmitIdent();
    TransmitTime();
 }.
```

В классе TElectricalDatas та же процедура переопределена:

```
void TElectricalDatas::Transmit();
 {
    TTelemetryDatas::Transmit();
    TransmitVoltage();
    TransmitAmperes();
 }.
```

Эта процедура сначала вызывает одноименную процедуру суперкласса **TTelemetryDatas::Transmit()**. Та передаст заголовок пакета (**intIdent** и **timeStamp**), после чего в подклассе передаются его собственные данные (напряжение и ток).

Определим экземпляры двух описанных выше классов:

```
TTelemetryDatas * objTelemetry;
TElectricalDatas * objElectrical;
```

и свободную подпрограмму[5]:

```
void EransmitFreshData (TTelemetryDatas * d; const TTime & t)
{
    if (d –> CurrentTime() >= t ) d –> Transmit();
}.
```

Теперь выполним следующие два оператора:

```
TransmitFreshData(objTelemetry, TTime(60));
TransmitFreshData(objElectrical, TTime(120));
```

Что при этом произойдет? В первом операторе будет передан уже известный нам заголовок пакета. Во втором будет передан он же, плюс четыре числа в формате с плавающей точкой, содержащие результаты измерений электрических параметров. Почему это так? Ведь функция **TransmitFreshData** ничего не знает о классе объекта, она просто выполняет **d –> Transmit()**! Это и есть пример полиморфизма. Переменная **d** может обозначать объекты разных классов. У этих классов есть общий суперкласс, и они, хотя и по-разному, могут реагировать на одно и то же сообщение, каждый раз правильно понимая его смысл.

Методы, подобные **TransmitFreshData**, принято называть полиморфными операциями. Традиционные типизированные языки типа Pascal, C основаны на той идее, что функции и процедуры, а следовательно, и операнды должны иметь определенный тип. Это свойство называется мономорфизмом, т. е. каждая переменная и каждое значение относятся к одному определенному типу.

[5] Свободная подпрограмма — это функция, которая определена вне класса. Как увидим далее, методика объектного программирования не рекомендует использовать свободные подпрограммы, однако в языке такая возможность предусмотрена.

В противоположность мономорфизму полиморфизм допускает отнесение значений и переменных к нескольким типам.

При отсутствии полиморфизма код программы вынуждено содержит множество операторов выбора **switch** или **if**. Например, на языке С невозможно образовать иерархию классов телеметрических данных; вместо этого придется определить одну большую запись с вариантами, включающую все разновидности данных. Для выбора варианта нужно проверить метку, определяющую тип записи. На языке С операция **TransmitFreshData** может быть написана следующим образом:

```
const int Electrical=1;
const int Propulsion=2;
const int Spectrometer=3;
void Transmit_Fresh_Data(TDatas Data, TTimes Time)
{
    if (Data.Current_Time >= Time)
        switch (Data.Kind)
            {
                case  Electrical:     Transmit_Electrical_Data(Data); break;
                      case  Propulsion:    Transmit_Propulsion_Data(Data); break;
                case  Spectrometer:  Transmit_ Spectrometer _Data(Data); break;
            };
        }.
```

Чтобы ввести новый тип телеметрических данных, нужно модифицировать эту вариантную запись, добавив новый тип в каждый оператор **case**. В такой ситуации увеличивается вероятность ошибок.

Наследование позволяет различать разновидности абстракций, и монолитные типы становятся не нужными. Полиморфизм наиболее целесообразен в тех случаях, когда несколько классов имеют схожее поведение. Полиморфизм позволяет обойтись без операторов выбора, поскольку объекты сами знают свой тип.

Множественное наследование. Мы рассмотрели вопросы, связанные с одиночным наследованием, т. е. когда подкласс имеет ровно один суперкласс. Однако одиночное наследование при всей своей полезности часто заставляет программиста выбирать между двумя равно привлекательными классами. Это ограничивает возможность повторного использования предопределенных классов и заставляет дублировать уже имеющиеся коды. Например, нельзя унаследовать графический элемент, который был бы одновременно окружностью и картинкой; приходится наследовать что-то одно и добавлять необходимое от второго.

Необходимость множественного наследования в объектно-ориентированном программировании остается предметом горячих споров.

Представьте себе, что нам надо организовать учет различных видов материального и нематериального имущества — банковских счетов, недвижимости, акций и облигаций. Банковские счета бывают текущие и сберегательные. Акции и облигации можно отнести к ценным бумагам, управление ими совершенно отлично от банковских счетов, но и счета и ценные бумаги — это разновидности имущества.

Однако есть много других полезных классификаций тех же видов имущества. В каком-то контексте может потребоваться отличать то, что можно за-

страховать (недвижимость и сберегательные вклады). Другой аспект — способность имущества приносить дивиденды; это общее свойство банковских счетов и ценных бумаг.

Очевидно, одиночное наследование в данном случае не отражает реальности, так что придется прибегнуть к множественному. Если мы составим структуру классов, в которой конечные классы (листья) могут быть сгруппированы во множества по разным ортогональным признакам (как в нашем примере, где такими признаками были способность приносить дивиденды и возможность страховки) и эти множества перекрываются, то это служит признаком невозможности обойтись одной структурой наследования, в которой бы существовали какие-то промежуточные классы с нужным поведением. Мы можем исправить ситуацию, используя множественное наследование, чтобы соединить два нужных поведения там, где это необходимо. Приведем пример на языке C++, получившаяся структура классов показана на рисунке 1.4[6]. На нем класс **Security** (ценные бумаги) наследует одновременно от классов **InterestBearingItem** (источник дивидендов) и **Asset** (имущество). Сходным образом, **BankAccount** (банковский счет) наследует сразу от трех классов: **InsurableItem** (страхуемое) и уже известным **Asset** и **InterestBearingItem**.

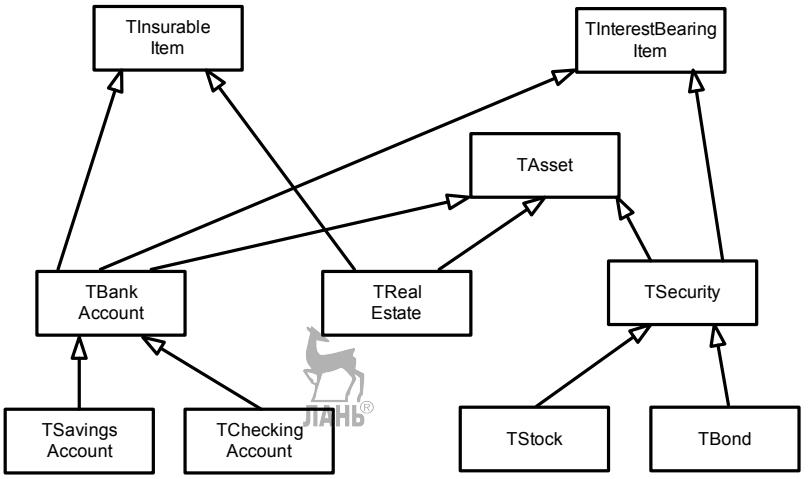


Рис. 1.4
Множественное наследование

Вот как это выражается на C++. Сначала вводится (незаконченное в примере) описание базовых классов:

```
class TAsset ...
class TInsurableItem ...
class TInterestBearingItem ....
```

Теперь промежуточные классы; каждый наследует от нескольких суперклассов:

[6] Язык **C++ — Visual Studio** поддерживает множественное наследование. Язык **C++** — **RAD Studio** не поддерживает множественное наследование.

class TBankAccount: public TAsset, public TInsurableItem, public TInterestBearingItem ...
class TRealEstate: public TAsset, public TInsurableItem ...
class TSecurity: public TAsset, public TInterestBearingItem ...

Наконец, листья иерархии (рис. 1.4):

class TSavingsAccount: public TBankAccount ...
class TCheckingAccount: public TBankAccount ...
class TStock: public TSecurity ...
class TBond: public TSecurity ...,

ЛАНЬ®

где, например, в последней строке примера выражение **public TSecurity** обозначает предка и является признаком наследования. Встречающееся несколько раз подряд выражение **public …** означает множественное наследование.

Проектирование структур классов с множественным наследованием — трудная задача, решаемая путем последовательных приближений. Есть две специфические для множественного наследования проблемы: как разрешить конфликты имен между суперклассами и что делать с повторным наследованием.

Конфликт имен происходит, когда в двух или более суперклассах случайно оказывается элемент (переменная или метод) с одинаковым именем. Представьте себе, что как **TAsset**, так и **TInsurableItem** содержат атрибут **presentValue**, обозначающий текущую стоимость. Так как класс **TRealEstate** (движимое и недвижимое имущество) наследует обоим этим классам, как понимать наследование двух операций с одним и тем же именем? Это, на самом деле, главная беда множественного наследования: конфликт имен может ввести двусмысленность в поведение класса с несколькими предками.

Проблема повторного наследования решается тремя способами. Во-первых, можно его запретить, отслеживая при компиляции. Во-вторых, можно явно развести две копии унаследованного элемента, добавляя к именам префиксы в виде имени класса-источника (это один из подходов, принятых в C++). В-третьих, можно рассматривать множественные ссылки на один и тот же класс, как обозначающие один и тот же класс. Так поступают в C++, где повторяющийся суперкласс определяется как виртуальный базовый класс. Виртуальный базовый класс появляется, когда какой-либо подкласс именует другой класс своим суперклассом и отмечает этот суперкласс как виртуальный, чтобы показать, что это общий (shared) класс.

Контрольные вопросы

1. Как вы понимаете понятие «наследование» в объектно-ориентированном программировании?
2. В чем отличие между одиночным и множественным наследованием?
3. Объясните понятие «одиночный полиморфизм».

1.4.4. Отношение агрегации

Отношение агрегации между классами имеет непосредственное отношение к агрегации между их экземплярами. Пример. Рассмотрим класс **TTemperatureControllers**:

```
class TTemperatureControllers
      {
       public:
              TTemperatureControllers();
              ~TTemperatureControllers();
              void Process(TTemperatureRamps * objTemperatureRamp);
              TMinute Schedule(TTemperatureRamps  * objTemperatureRamp);
       private
              THeaters * objHeater;
      }.
```

Как видно из рисунка 1.5, класс **TTemperatureControllers** — это, несомненно, целое, а экземпляр класса **THeaters** (нагреватель) — одна из его частей.

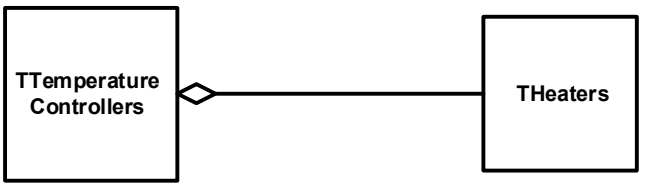


Рис. 1.5
Агрегация

Различают два вида агрегации: физическое и косвенное включение.

В случае класса **TTemperatureControllers** мы имеем агрегацию по значению; эта разновидность физического включения означает, что объект класса **THeaters** не существует отдельно от объемлющего экземпляра класса **TTemperatureControllers**. Агрегация является направленной, как и всякое отношение «целое/часть». Объект **THeaters** входит в объект **TTemperatureControllers**, а не наоборот. Физическое вхождение одного в другое нельзя «зациклить» («часть» не может содержать в себе «целое»).

Агрегация не требует обязательного физического включения. Например, акционер владеет акциями, но они не являются его физической частью. Более того, время жизни этих объектов может быть совершенно различным, хотя концептуально отношение целого и части сохраняется, и каждая акция входит в имущество своего акционера. Поэтому агрегация может быть очень косвенной. Например, объект класса **TShareholders** (акционер) может содержать ключ записи об этом акционере в базе данных акций. Это тоже агрегация без физического включения. «Лакмусовая бумажка» для выявления агрегации такова: если (и только если) налицо отношение «целое/часть» между объектами, их классы должны находиться в отношении агрегации друг с другом.

Решая, с чем мы имеем дело — с наследованием или агрегацией — надо быть осторожным. Если нет уверенности, что налицо отношение общего и частного (is-*a*), вместо наследования лучше применить агрегацию или что-нибудь еще.

Контрольные вопросы

1. Как вы понимаете понятие «агрегация» в объектно-ориентированном программировании?
2. Какие виды включения существуют? Поясните примерами.

1.4.5. Отношение зависимости

Рассмотрим снова пример из главы «Агрегация»:

```
class TTemperatureControllers
    {
     public:
          TTemperatureControllers();
          ~TTemperatureControllers();
         void Process(TTemperatureRamps & objTemperatureRamp);
           TMinute Schedule(TTemperatureRamps  & objTemperatureRamp);
     private
           Theaters * objHeater;
    }
```

На рисунке 1.6 приведен пример, в котором иллюстрируется связь между классами **TemperatureControllers** и **TemperatureRamps**, получившая название отношения зависимости.

Класс **TTemperatureRamps** упомянут как параметр методов **Process** и **Schedule**; это дает нам основания сказать, что класс **TTemperatureControllers** пользуется услугами класса **TTemperatureRamps**.

Клиенты и серверы — это то, во что превращается ассоциация, если оказывается, что одна из ее сторон (клиент) пользуется услугами другой (сервера).

На самом деле, один класс может использовать другой по-разному. В нашем примере это происходит в функции **Schedule**. Можно представить, что **TTemperatureControllers** внутри реализации функции **Schedule** использует, например, экземпляр класса **TPredictor** (предсказатель). Отношения целого и части тут ни при чем, поскольку этот объект не входит в объект **TTemperatureController**, а только используется. В типичном случае такое отношение зависимости проявляет себя, если в реализации какой-либо операции происходит объявление локального объекта используемого класса.

В общем случае предполагается, что изменения в классе **TTemperature-Ramps** могут потребовать изменений в классе **TTemperatureControllers**.

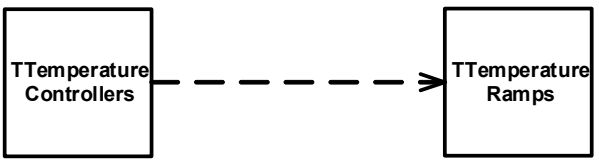


Рис. 1.6
Отношение зависимости

Контрольные вопросы

1. Как вы понимаете понятие «зависимость» в объектно-ориентированном программировании?
2. Какие отношения существуют между клиентом и сервером? Поясните примерами.

1.4.6. Отношение реализации (инстанцирования)

Инстанцирование является аналогом параметризованного абстрактного типа данных.

Наша первая попытка сконструировать класс **TQueue** (очередь) была не особенно успешной, поскольку нам не удалось сделать его безопасным в отношении типов. Мы можем значительно усовершенствовать нашу абстракцию, если прибегнем к конструкции параметризованных классов.

```
template <class TItem>  class TQueue
   {
      public:
         TQueue();
         TQueue(const TQueue<TItem>&);
         virtual ~Queue();
         virtual TQueue<TItem>& operator=(const TQueue<TItem>&);
         virtual int operator == (const TQueue<TItem>&) const;
         int operator != (const TQueue<TItem>&) const;
         virtual void Clear();
         virtual void Append(const TItem&);
         virtual void Pop();
         virtual void Remove(int at);
         virtual int Length() const;
         virtual int IsEmpty() const;
         virtual const TItem& Front() const;
         virtual int Location(const void*);
      protected:
                  ...
   };
```

В этом варианте объекты помещаются в очередь и достаются из нее через класс **TItem**, объявленный как аргумент шаблона.

Параметризованный класс не может иметь экземпляров, пока он не будет инстанцирован. Объявим две конкретных очереди — очередь целых чисел и очередь экранных объектов:

```
TQueue<int> intQueue;
TQueue<TPoint*> pntQueue;
```

Объекты **intQueue** и **pntQueue** — это экземпляры совершенно различных классов, которые даже не имеют общего суперкласса. Тем не менее они получены из одного параметризованного класса **TQueue**.

Инстанцирование безопасно с точки зрения типов. По правилам C++ будет отвергнута любая попытка поместить в очередь или извлечь из нее что-либо кроме, соответственно, целых чисел и значений типа **TPoint**.

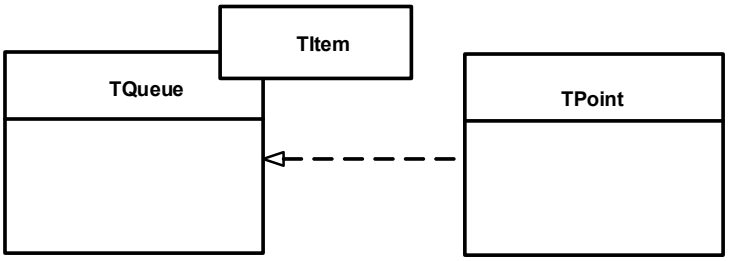


Рис. 1.7
Инстанцирование

Отношение между параметризованным классом **TQueue**, его инстанцированием для класса **TPoint** показано на рисунке 1.7.

Контрольные вопросы

1. Как вы понимаете понятие «инстанцирование» в объектно-ориентированном программировании? Поясните примерами.

1.4.7. Метаклассы

Как было сказано, любой объект является экземпляром какого-либо класса. Что будет, если мы попробуем и с самими классами обращаться как с объектами? Для этого нам надо ответить на вопрос: что же такое класс класса? Ответ — это метакласс. Иными словами, метакласс — это класс, экземпляры которого суть классы. Метаклассы венчают объектную модель в чисто объектно-ориентированных языках. Классы доставляют программисту интерфейс для определения объектов. Если так, то желательно, чтобы и сами классы были объектами, так, чтобы ими можно было манипулировать, как всеми остальными описаниями.

Понятию «метакласс» в абстрактных типах данных соответствует понятие «тип типов» (глава 4).

Хотя в C++Builder метаклассов нет, но есть средства поддержки и переменных класса, и операций метакласса. Конкретно в C++Builder можно вводить специальные методы класса. Главной целью введения методов класса является получение возможности непосредственно работать с классами, а также создавать такие инструменты разработчика, как браузеры классов и объектов.

Контрольные вопросы

1. Сформулируйте своими словами понятие «метакласс».
2. Какие средства используются в C++Builder для реализации метакласса?

1.5. Взаимосвязь классов и объектов

Классы и объекты — это отдельные, но тесно связанные понятия. В частности, каждый объект является экземпляром какого-либо класса; класс может порождать любое число объектов. В большинстве практических случаев классы статичны, т. е. все их особенности и содержание определены в процессе компиляции программы. Из этого следует, что любой созданный объект отно-

сится к строго фиксированному классу. Сами объекты, напротив, в процессе выполнения программы создаются и уничтожаются.

В качестве примера рассмотрим классы и объекты для задачи управления воздушным движением. Наиболее важные абстракции в этой сфере — самолеты, графики полетов, маршрут и коридоры в воздушном пространстве. Трактовка этих классов объектов по самому их определению достаточно статична. Иначе невозможно было бы построить никакого приложения, использующего такие общепонятные факты: самолеты могут взлетать, летать и приземляться, а также что никакие два самолета не должны находиться одновременно в одной и той же точке.

Объекты же этих классов, напротив, динамичны. Набор маршрутов полетов сменяется не очень часто. Существенно быстрее изменяется множество самолетов, находящихся в полете. Частота, с которой самолеты занимают и покидают воздушные коридоры, еще выше.

Контрольные вопросы

1. В чем заключается принципиальное отличие класса и объекта?

В заключение приведем некоторые основополагающие тезисы:

— объект характеризуется свойствами, состоянием, поведением и идентичностью;

— структура и поведение одинаковых объектов описываются в общем для них классе;

— свойства объекта определяют его статические свойства;

— состояние объекта определяет его динамические свойства;

— поведение объекта характеризуется изменением его состояния в процессе взаимодействия (посредством передачи сообщений) с другими объектами;

— идентичность объекта — это его отличия от всех других объектов;

— иерархия объектов может строиться на принципах наследования или агрегации;

— множество объектов с одинаковой структурой и поведением является классом;

— шесть типов иерархий классов включают: ассоциирование, наследование, агрегация, зависимость, инстанцирование и метаклассирование;

— классы и объекты, образующие словарь предметной области, называются ключевыми абстракциями;

— структура, объединяющая множество объектов и обеспечивающая их совместное целенаправленное функционирование, называется механизмом;

— качество абстракций измеряется их зацеплением, связностью, достаточностью, полнотой и примитивностью.

Задачи

В приведенных задачах необходимо:

1. Выделить классы.
2. Определить отношения между классами.
3. Выделить объекты.
4. Определить отношения между объектами.

1. Программное обеспечение микропроцессора диктофона

Требуется разработать модель программного обеспечения, управляющего работой цифрового диктофона. Цифровой диктофон — это бытовое электронное устройство, предназначенное для записи и воспроизведения речи. Звуковые сообщения записываются через встроенный микрофон и сохраняются в памяти устройства. Сообщения воспроизводятся через встроенный громкоговоритель. Диктофон должен быть маленьким, легким, удобным в использовании и работать от батареек.

Основные требования к диктофону.

Устройство должно хранить до 10 сообщений. Длина каждого сообщения ограничена размером свободной памяти. Диктофон должен осуществлять прямой доступ к любому сообщению из памяти.

Интерфейс с пользователем должен осуществляться при помощи экранного меню. На дисплее должны постоянно отображаться текущее время и дата. Дисплей также должен показывать информацию о том, что в данный момент делает устройство.

Дополнительная функция устройства — будильник. Должна быть возможность установки момента времени, при наступлении которого подается сигнал будильника. Сигнал будильника прекращается, если пользователь нажал какую-либо кнопку или по истечении 60 с.

Если диктофон не используется, он должен автоматически перейти в режим малого потребления энергии. Переход из режима малого потребления энергии в обычный режим осуществляется при нажатии пользователем любой кнопки.

Индикатор уровня заряда батарей должен подавать сигнал при разрядке батарей ниже установленного предела. При разряженных батареях диктофон автоматически переходит в режим малого потребления энергии.

2. Программное обеспечение микропроцессора торгового автомата

Требуется разработать модель программного обеспечения встроенного процессора универсального торгового автомата.

Автомат выполняет следующие функции. Он имеет отделения для хранения товаров (до 10 различных наименований). При приеме товара название, цена и количество товара вводятся в память автомата. Автомат должен следить за наличием товара. Если какой-либо товар распродан, автомат должен сообщить об этом на центральную станцию и информировать покупателей (например, зажечь красную лампочку рядом с кнопкой выдачи данного товара).

Автомат должен принимать к оплате бумажные купюры и монеты (установленных образцов). Специальный индикатор должен высвечивать текущую сумму денег, принятых автоматом к оплате. После достижения заданной стоимости клиент должен выбрать товар и нажать кнопку выдачи выбранного товара. Должна быть предусмотрена кнопка «ВОЗВРАТ», при нажатии на которую (но до выдачи покупки) клиенту возвращаются все принятые от него к оплате деньги.

Регулярно автомат сдает выручку инкассаторам.

3. Программное обеспечение микропроцессора холодильника

Требуется разработать модель программного обеспечения встроенного микропроцессора холодильника. Холодильник состоит из нескольких холодильных камер для хранения продуктов. Доступ в каждую камеру осуществляется через дверцу. При помещении продуктов в холодильник и при извлечении их из холодильника считывается и запоминается штрих-код на упаковке. Пользователь должен иметь возможность получить информацию о продуктах, хранящихся в какой-либо камере холодильника, или полный список хранящихся продуктов.

Для каждой холодильной камеры имеется регулятор температуры, при помощи которого можно устанавливать максимальную допустимую температуру в данной камере. Для понижения температуры в холодильной камере используется компрессор (для каждой камеры свой). С камерой связан индикатор температуры, который постоянно выдает текущее значение температуры в камере. При повышении температуры выше определенного предела, определяемого текущим положением регулятора, должен включаться компрессор. При снижении температуры ниже некоторого другого значения, связанного с первым, компрессор должен отключаться. Если дверь холодильной камеры открыта в течение слишком долгого времени, должен подаваться звуковой сигнал. Дополнительно, звуковой сигнал должен подаваться в любых нештатных ситуациях (например, при невозможности обеспечить требуемую температуру в течение заданного интервала времени либо при поломке индикатора температуры).

Необходимо предусмотреть специальный режим размораживания, включаемый для очистки пустой холодильной камеры ото льда. При попытке разморозить заполненную камеру должен включаться предупредительный сигнал.

4. Программное обеспечение микропроцессора стиральной машины

Требуется разработать модель программного обеспечения встроенного микропроцессора стиральной машины.

Машина должна выполнять следующие функции. В машине имеется бак для белья, клапаны для забора и слива воды, мотор, устройство подогрева воды, таймер, дверца для доступа в бак, датчик уровня воды в баке и несколько емкостей для различных моющих средств. В памяти машины хранятся программы стирки. Каждая программа определяет температуру воды, длительность стирки, используемые моющие средства (номер емкости и время подачи), частоту вращения бака во время стирки и отжима. Органы управления машиной должны позволять вводить в память машины новые программы стирки и удалять из памяти старые. Для использования машины необходимо открыть дверцу, поместить белье в бак, моющие средства в емкости, закрыть дверцу, выбрать программу стирки и нажать кнопку «ПУСК». Машина должна открыть клапан для забора воды, набрать необходимое количество воды, затем закрыть клапан. Далее машина действует по выбранной пользователем программе:

1) подогревает, если необходимо, воду до нужной температуры;
2) включает таймер и запускает вращение бака для стирки;
3) по таймеру подает в бак моющее(ие) средство(а);
4) по окончании стирки сливает воду и запускает отжим.

Пользователь имеет возможность принудительно остановить стирку и слить воду.

5. Программное обеспечение микропроцессора таксофона[7]

Требуется разработать модель программного обеспечения таксофона городской телефонной сети. Аппарат подключен к линии связи. В нем имеется устройство для считывания телефонных карт (для оплаты разговора) — приемник карт. Таксофон должен выполнять следующие функции.

При снятии трубки таксофон выдает на дисплей сообщение «вставьте карту». При помещении карты в приемник карт с нее считывается информация. Если кредит исчерпан или карта непригодна (не удается узнать кредит), то подается звуковой сигнал и соответствующее сообщение появляется на дисплей таксофона. Иначе на дисплее изображается количество «единиц» на карте, а на линию (на телефонную станцию) подается сигнал «Трубка». При поднятой трубке таксофон реагирует на сигналы «Тон» и «Занято» из линии.

При получении сигнала «Тон» из линии таксофон должен воспроизвести звуковой тон «Готов» в трубку. При получении сигнала «Занято» аппарат должен воспроизвести звуковой тон «занято» в трубку. При положенной трубке эти сигналы игнорируются.

На таксофоне имеются цифровые кнопки 0–9. После получения сигнала «Тон» последовательность семи нажатий кнопок с цифрами 0–9 осуществляет вызов абонента с данным кодом. Во время набора номера он должен отображаться на дисплее таксофона. После того как телефонная станция произвела коммутацию линии, на дисплее начинается отсчет времени разговора, а таксофон и аппарат вызванного абонента могут обмениваться данными. При получении сигнала «Данные» из линии таксофон должен воспроизвести данные, передаваемые с сигналом, в трубку. При получении данных из трубки аппарат должен преобразовать их в сигнал «данные» и передать их в линию. По ходу разговора уменьшается кредит на телефонной карте.

Обмен данными прерывается в следующих случаях:

— исчерпан кредит;

— карта вынута из приемника карт;

— от телефонной станции пришел сигнал «Тон» (вызванный абонент повесил трубку);

— повешена трубка таксофона.

Если трубка была повешена, аппарат посылает в линию сигнал «Конец» и выдает на дисплей сообщение «Выньте карту». После чего, пока карта находится в приемнике карт, таксофон подает предупредительный звуковой сигнал.

6. Программное обеспечение интернет-магазина

Требуется разработать модель программного обеспечения интернет-магазина. Интернет-магазин позволяет делать покупки с доставкой на дом. Клиентам магазина обеспечивается доступ к каталогу продаваемых товаров. В каталоге

[7] В настоящее время таксофоны используются нечасто. В некоторых странах, например в Англии, таксофоны остаются неотъемлемой частью городского пейзажа.

товары распределены по разделам. О каждом товаре доступна полная информация (название, вес, цена, изображение, дата изготовления и срок годности). Для удобства клиентов предусмотрена система поиска товаров в каталоге.

Помимо информации о товарах клиенты также должны получить информацию об условиях оплаты и доставки покупок.

При отборе клиентами товаров должна поддерживаться виртуальная «торговая тележка». Любое наименование товара может быть добавлено в «тележку» или изъято в любой момент по желанию покупателя с последующим пересчетом общей стоимости покупки. Необходимо также, чтобы текущее содержимое «тележки» постоянно отображалось на экране клиента.

По окончании выбора товаров производится оформление заказа и регистрация покупателя. Заказы передаются для обработки в систему автоматизации торговли. Проверка наличия товаров на складе и их резервирование интернет-магазином не производятся.

ЛАНЬ®

7. Программное обеспечения WWW-конференции

Требуется разработать модель программного обеспечения WWW-конференции. WWW-конференция представляет собой хранилище сообщений. Для каждого сообщения хранятся сведения об авторе, теме, дате сообщения и тексте сообщения. Всем пользователям конференции должен быть доступен иерархический список сообщений. Верхний уровень иерархии составляют сообщения, открывающие новые темы, нижние уровни — сообщения, полученные в ответ на сообщения верхнего уровня. Для удобства пользователей необходимо предусмотреть поиск сообщений по автору или по ключевым словам в теме или тексте сообщения.

Сообщения добавляются в конференцию зарегистрированными пользователями, которые должны указать свое имя и пароль. При этом есть возможность начать новую тему или ответить на ранее добавленные сообщения. После добавления сообщения оно должно быть доступно для чтения всем пользователям (даже незарегистрированным), а список сообщений должен быть обновлен. Автор сообщения имеет возможность поправить его текст даже после того, как сообщение было добавлено в конференцию. При этом в тексте сообщения появляется пометка об изменениях, указывается дата обновления. Один из зарегистрированных пользователей — модератор — имеет право по тем или иным причинам удалять сообщения любых авторов.

2. МОДЕЛИРОВАНИЕ И ДОКУМЕНТИРОВАНИЕ СЛОЖНЫХ СИСТЕМ

Вопрос: «Как написать хорошую программу на C++?» очень похож на вопрос «Как пишется хорошая английская проза?» На него есть два ответа: «Нужно знать, что вы собственно хотите написать» и «Практика и подражание хорошему стилю». Оба совета пригодны для C++ в той же мере, что и для английского языка, и обоим достаточно трудно следовать.

Б. Страуструп

В настоящей главе рассматривается унифицированный язык моделирования UML (Unified Modeling Language), который ориентирован на решение задач первых двух этапов жизненного цикла программ и признается многими специалистами как стандарт документирования результатов анализа и проектирования программных систем. Аббревиатура UML допускает соответствующий перевод и последующее сокращение, но ввиду неудобочитаемости получающегося «УЯМ» обычно используется исходное обозначение.

Около 300 ведущих компьютерных фирм, объединенных в консорциуме OMG (Object Management Group), придают большое значение работам по созданию и развитию языка UML. Причину такого внимания к языку поясним с помощью рисунка 2.1. Рисунок позволяет понять причины революционных перемен в области технологий программирования, вызванных появлением языка UML. На нем изображены две схемы. Первая из них (рис. 2.1, *а*) изображает ситуацию, существовавшую в области технологий программирования до создания языка UML, вторая (рис. 2.1, *б*) — показывает изменение ситуации после появления UML. На обеих схемах слева показаны программисты и воображаемые ими модели компьютерных программ, а справа изображены программы и предметные области, в которых эти программы используются. На второй схеме между предметными областями и программными кодами появились диаграммы языка UML.

Как видно из рисунка 2.1, *а*, соответствие текста программы (ее исходного кода) характеристикам объекта автоматизации фиксируется только в сознании программиста, а документальная связь между ними отсутствует.

Рассмотрим теперь ситуацию, возникшую после появления языка UML (рис. 2.1, *б*). Диаграммы и спецификации языка UML связали исходный текст программы с характеристиками объекта автоматизации. При этом UML диаграммы опираются на теоретический фундамент в виде теории множеств и теории графов. Наличие теоретической основы позволяет упростить операции преобразования UML диаграмм, нарисованных на экранах дисплеев, в память компьютеров и уменьшить объем памяти, необходимой для хранения диаграмм.

Рисунок также показывает, что UML диаграммы могут преобразовываться в исходный код (прямое преобразование), и наоборот — исходный код может преобразовываться в диаграммы (обратное преобразование). В некоторых случаях прямое преобразование может осуществляться автоматически с помощью программ конверторов. Обратное преобразование может выполнить только человек.

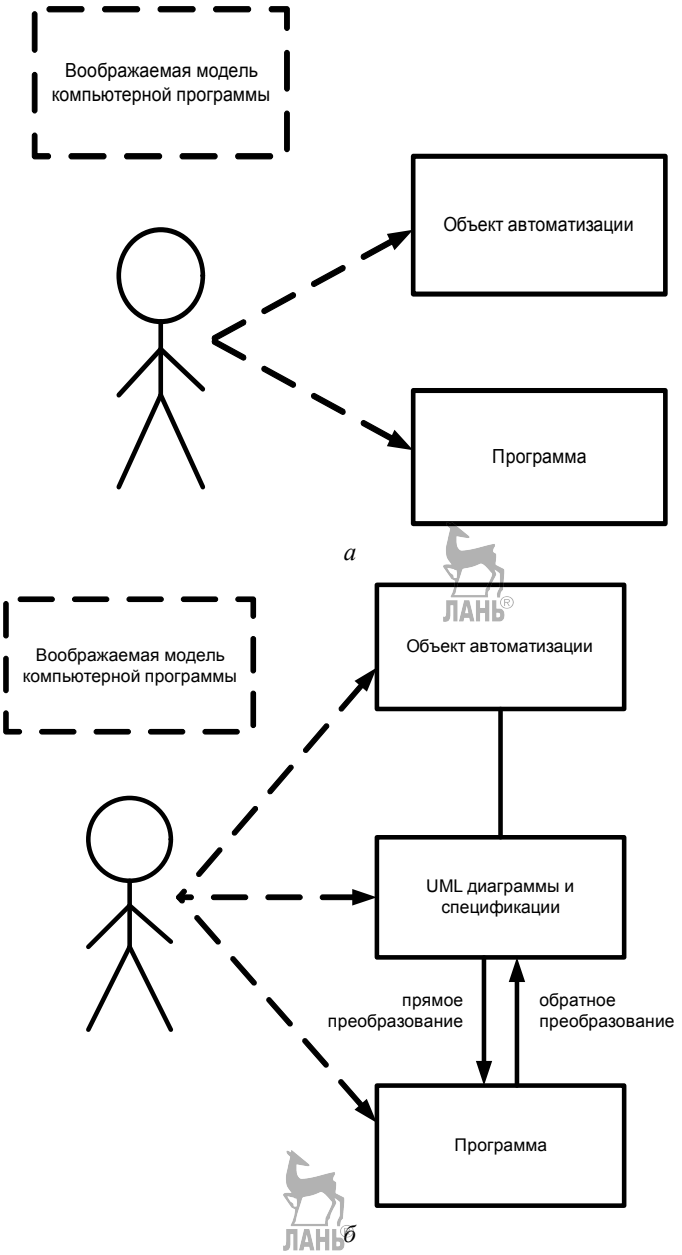


Рис. 2.1
Представление программиста о программе

Язык UML помог в решении болезненной для компьютерных фирм проблемы увольнения программистов с работы. Действительно, если программист уволится, то другой программист, пришедший ему на смену, чтобы войти в курс дела, должен разобраться во всех тонкостях исходного кода программы. Практика показывает, что если программа сложная, то сделать это трудно, а иногда и невозможно. До появления языка UML увольнение программиста

могло повлечь за собой потерю его программ. Сегодня, если программы документированы на языке UML, то после увольнения программиста другой программист, используя UML диаграммы и спецификации, может понять исходный код программы и заменить уволившегося.

Помимо решения проблемы увольнения программистов, документирование исходных кодов программ UML диаграммами и спецификациями создает единый язык общения между программистами, а также между программистами, системными аналитиками и заказчиками автоматизированной системы. Но самое главное, что дал UML — это возможность широкой стандартизации языков программирования. Известно, что в разных языках программирования используются одинаковые операции и методы, но они имеют разные названия и символьные обозначения. Язык UML позволяет стандартизовать как сами операции и методы языков программирования, так и их терминологию.

Появление диаграмм языка UML отражает наметившуюся в компьютерном мире тенденцию моделирования сложных систем визуальными (наглядными) моделями. Набор из нескольких наглядных моделей создает в сознании специалистов интегральный образ сложной компьютерной системы, которую они совместно проектируют. Вместе с тем, наглядные модели служат эффективным средством документирования автоматизированных систем, а также языком общения между программистами, системными аналитиками и заказчиками систем. Диаграммы и спецификации языка UML позволяют записать и проанализировать идеи начального этапа проектирования программ и связать полученные модели с исходными текстами будущих программ. При этом UML диаграммы, с одной стороны, являются довольно свободным способом записи мыслей, а с другой стороны — опираются на теоретический фундамент в виде теории множеств и теории графов. На идее прямого преобразования UML диаграмм в исходный код программы основаны принципы построения автоматизированных систем проектирования программных изделий: CASE-системы (Computer Aided Software Engineering).

Язык UML основан на некотором числе базовых понятий, которые могут быть изучены и применены большинством программистов и разработчиков, знакомых с методами объектно-ориентированного анализа и проектирования. При этом базовые понятия могут комбинироваться и расширяться таким образом, что специалисты объектного моделирования получают возможность самостоятельно разрабатывать модели больших и сложных систем в самых различных областях приложений.

Использование языка UML основывается на понимании общих принципов моделирования сложных систем и особенностей процесса объектно-ориентированного анализа и проектирования. Выбор выразительных средств построения моделей сложных систем предопределяет те задачи, которые могут быть решены с использованием данных моделей. При этом одним из основных принципов построения моделей сложных систем является принцип абстрагирования, который предписывает включать в модель только те аспекты проектируемой системы, которые имеют непосредственное отношение к выполнению системой своих функций или своего целевого предназначения. При этом все

второстепенные детали опускаются, чтобы чрезмерно не усложнять процесс анализа и исследования полученной модели.

Другим принципом моделирования сложных систем является принцип многомодельности. Этот принцип представляет собой утверждение о том, что никакая единственная модель не может с достаточной степенью адекватности описывать различные аспекты сложной системы. Набор из нескольких наглядных моделей (диаграмм языка UML) создает в сознании специалистов интегральный образ сложной компьютерной системы, которую они совместно проектируют. Применительно к методологии объектно-ориентированного проектирования это означает, что достаточно полная модель сложной системы допускает существование взаимосвязанных подмоделей, каждая из которых адекватно отражает некоторый аспект поведения или структуры системы. При этом наиболее общими представлениями о модели сложной системы принято считать статическое и динамическое представления, которые, в свою очередь, могут подразделяться на другие более частные представления. Феномен сложной системы как раз и состоит в том, что никакое ее единственное представление не является достаточным для адекватного выражения всех особенностей моделируемой системы.

Еще одним принципом прикладного системного анализа является принцип иерархического построения моделей сложных систем. Этот принцип предписывает рассматривать процесс построения модели на разных уровнях абстрагирования или детализации в рамках фиксированных представлений. При этом исходная или первоначальная модель сложной системы имеет наиболее общее представление. Такая модель строится на начальном этапе проектирования и может не содержать многих деталей и аспектов моделируемой системы. Язык UML позволяет адекватно отразить различные этапы развития программного проекта.

Таким образом, процесс объектно-ориентированного проектирования можно представить как последовательность шагов от наиболее общих моделей и представлений концептуального уровня к более частным и детальным представлениям логического и, наконец, физического уровня. При этом на каждом шаге данные модели последовательно дополняются все большим количеством деталей, что позволяет им более адекватно отражать различные аспекты конкретной реализации сложной системы. Заключительным шагом является получение исходного текста программы.

Интегрированная модель сложной системы представлена на рисунке 2.2.

Стандарт UML предлагает следующий набор диаграмм для моделирования (рис. 2.2):

— диаграммы вариантов использования (use case diagrams) — для моделирования бизнес-процессов организации и требований к создаваемой системе;

— диаграммы классов (class diagrams) — для моделирования статической структуры классов системы и связей между ними;

— диаграммы поведения системы (behavior diagrams):

  - диаграммы взаимодействия (interaction diagrams):

- диаграммы последовательности (sequence diagrams) и кооперативные диаграммы (collaboration diagrams) — для моделирования процесса обмена сообщениями между объектами;

o диаграммы состояний (statechart diagrams) — для моделирования поведения объектов системы при переходе из одного состояния в другое;

o диаграммы деятельностей (activity diagrams) — для моделирования поведения системы в рамках различных вариантов использования, или моделирования деятельностей;

— диаграммы реализации (implementation diagrams):

o диаграммы компонентов (component diagrams) — для моделирования иерархии компонентов (подсистем) системы;

o диаграммы развертывания (deployment diagrams) — для моделирования физической архитектуры системы.

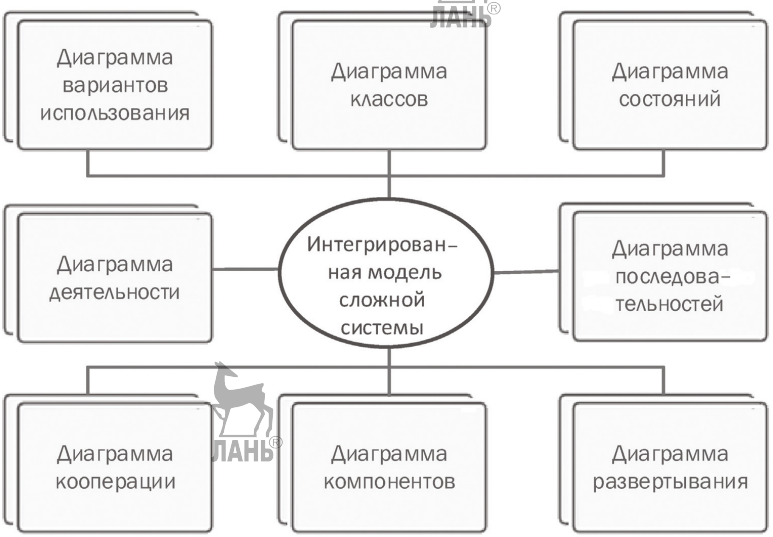


Рис. 2.2
Интегрированная модель сложной системы

Ввиду того, что настоящая глава носит ознакомительный характер, мы рассмотрим три вида диаграмм: диаграммы классов, диаграммы деятельности и диаграммы последовательности.

2.1. Особенности изображения диаграмм языка UML

Большинство диаграмм являются в своей основе графами специального вида, состоящими из вершин в форме геометрических фигур, которые связаны между собой ребрами или дугами. Поскольку информация, которую содержит в себе граф, имеет в основном топологический характер, ни геометрические размеры, ни расположение элементов диаграмм (за некоторыми исключениями, такими как диаграмма последовательностей, с метрической осью времени) не имеют принципиального значения.

Для диаграмм языка UML существуют четыре типа визуальных обозначений, которые важны с точки зрения заключенной в них информации:

— геометрические фигуры, называемые также символами, обозначают процессы или сущности, соответствуют понятию узла или вершины в теории графов; форма этих фигур (прямоугольник, эллипс) соответствует некоторым элементам языка UML (класс, вариант использования) и имеет фиксированную семантику. Геометрические фигуры могут иметь различную высоту и ширину с целью размещения внутри этих фигур других конструкций языка UML;

— связи, которые представляются различными линиями на плоскости. Связи в языке UML обобщают понятие дуг или ребер из теории графов, но имеют менее формальный характер;

— текст, который содержится внутри границ отдельных геометрических фигур на плоскости. Информация, содержащаяся внутри фигур, имеет значение для конкретной модели проектируемой системы, поскольку регламентирует реализацию соответствующих элементов в программном коде;

— графические символы, изображаемые вблизи от тех или иных визуальных элементов диаграмм.

Процесс построения отдельных типов диаграмм имеет свои особенности, которые тесно связаны с семантикой элементов этих диаграмм. Сам процесс объектно-ориентированного анализа и проектирования в контексте языка UML получил специальное название — рациональный унифицированный процесс (Rational Unified Process, RUP)[8].

Суть концепции RUP заключается в последовательной декомпозиции или разбиении процесса объектно-ориентированного анализа и проектирования программ на отдельные этапы, на каждом из которых осуществляется разработка соответствующих типов канонических диаграмм модели системы. При этом на начальных этапах RUP строятся логические представления статической модели структуры системы, затем логические представления модели поведения и лишь после этого физические представления модели системы. Как нетрудно заметить, в результате RUP должны быть построены канонические диаграммы на языке UML.

Рассмотрим принципы построения важнейших диаграмм: диаграммы классов, диаграммы деятельности и диаграммы последовательности.

Контрольные вопросы

1. Какие типы визуальных обозначений используются для построения диаграмм языка UML?

2.2. Диаграмма классов

Центральное место в объектно-ориентированном анализе и проектировании программ занимает разработка логической модели системы в виде диа-

[8] При дословном переводе термина RUP теряется некоторая дополнительная семантическая окраска, связанная с двусмысленным толкованием английского Rational. Речь идет о другом варианте перевода — унифицированный процесс от фирмы Rational Software.

граммы классов. Диаграмма классов служит для представления статической модели системы в терминологии классов объектно-ориентированного программирования. Диаграмма классов может отражать, в частности, различные взаимосвязи между отдельными сущностями предметной области, такими как классы и подсистемы, а также описывает их внутреннюю структуру и типы отношений. На данной диаграмме не указывается информация о временных аспектах функционирования системы. С этой точки зрения диаграмма классов является наиболее общим представлением концептуальной модели проектируемой системы.

Диаграмма классов представляет собой некоторый граф, вершинами которого являются элементы типа «класс», которые связаны различными типами структурных отношений. Когда говорят о данной диаграмме, имеют в виду статическую структурную модель проектируемой системы. Поэтому диаграмму классов принято считать графической моделью структурных взаимосвязей логической модели системы.

Диаграмма классов состоит из множества элементов, которые в совокупности отражают декларативные знания о предметной области. Несколько родственных по тематике диаграмм могут образовывать пакеты для представления более общей модели системы. Если диаграмма классов является частью некоторого пакета, то ее компоненты должны соответствовать элементам этого пакета, включая возможные ссылки на элементы из других пакетов.

Таким образом, статическая структурная модель может быть представлена в виде одной или нескольких диаграмм классов. Декомпозиция некоторого представления на отдельные диаграммы выполняется с целью удобства и графической визуализации структурных взаимосвязей предметной области. При этом компоненты диаграммы соответствуют элементам статической семантической модели. Модель системы, в свою очередь, должна быть согласована с внутренней структурой классов, которая описывается на языке UML.

2.2.1. Классы

В первой главе было установлено, что класс служит для обозначения множества объектов, которые обладают одинаковой структурой, поведением и отношениями с объектами из других классов. Графически класс в языке UML изображается в виде прямоугольника, который дополнительно может быть разделен горизонтальными линиями на разделы или секции (рис. 2.3). В этих разделах могут указываться имя класса, атрибуты (переменные) класса и операции (методы) класса.

Обязательным элементом обозначения класса является его имя. На начальных этапах разработки диаграммы отдельные классы могут обозначаться простым прямоугольником с указанием только имени соответствующего класса (рис. 2.3, *а*). По мере проработки класса описание класса дополняется атрибутами (рис. 2.3, *б*) и операциями (рис. 2.3, *в*).

Предполагается, что окончательный вариант диаграммы содержит наиболее полное описание классов, которые состоят из трех разделов или секций. Иногда в обозначениях классов используется дополнительный четвертый раз-

дел, в котором приводится семантическая информация справочного характера или явно указываются исключительные ситуации.

Если секция атрибутов и операций является пустой, в обозначении класса она выделяется горизонтальной линией, чтобы сразу отличить класс от других элементов языка UML. Примеры графического изображения классов на диаграмме классов приведены на рисунке 2.4. В первом случае для класса «Прямоугольник» (рис. 2.4, *а*) указаны только его атрибуты — точки на координатной плоскости, которые определяют его расположение. Для класса «Окно» (рис. 2.4 *б*) указаны только его операции, секция атрибутов оставлена пустой. Для класса «Счет» (рис. 2.4, *в*) дополнительно изображена четвертая секция, в которой указана исключительная ситуация: отказ от обработки просроченной кредитной карточки.

Рис. 2.3
Графическое изображение класса на диаграмме классов

Рис. 2.4
Примеры графического изображения классов на диаграмме

Имя класса. Имя класса должно быть уникальным в пределах пакета, который описывается некоторой совокупностью диаграмм классов (возможно, одной диаграммой). Оно указывается в первой верхней секции прямоугольника. Имя класса записывается по центру секции имени полужирным шрифтом и должно начинаться с заглавной буквы. Рекомендуется в качестве имен классов использовать существительные, записанные по практическим соображениям без пробелов. Необходимо помнить, что именно имена классов образуют словарь предметной области. Выбор единственного или множественного числа в обозначении имени класса не является принципиальным, однако нельзя забы-

вать, что в объектном программировании понятие «класс» обозначает множество однородных объектов.

С учетом последнего замечания примерами имен классов могут быть такие существительные, как **Сотрудники**, **Компании**, **Руководители**, **Двигатели**, **Шасси**, **Редукторы** и многие другие, имеющие непосредственное отношение к моделируемой предметной области и функциональному назначению проектируемой системы.

В первой секции может приводиться также информация о разработчике данного класса и статус состояния разработки, а также могут записываться и другие общие свойства этого класса, имеющие отношение к другим классам диаграммы или стандартным элементам языка UML. Для этой цели могут использоваться также комментарии.

Класс может не иметь экземпляров или объектов. В этом случае он называется абстрактным классом, а для обозначения его имени используется наклонный шрифт (курсив).

В языке UML принято общее соглашение о том, что любой текст, относящийся к абстрактному элементу, записывается курсивом. Данное обстоятельство является семантическим аспектом описания соответствующих элементов языка UML.

В некоторых случаях необходимо явно указать, к какому пакету диаграмм относится тот или иной класс. Для этой цели используется специальный символ-разделитель — двойное двоеточие «::». Синтаксис строки имени класса в этом случае будет следующий *<Имя_пакета>*::*<Имя_класса>*. Например, если определен пакет с именем **Банки**, то класс **Счета** в банке может быть записан в виде: **Банки::Счета**.

Атрибуты класса. Во второй сверху секции прямоугольника класса записываются его атрибуты (attributes) или свойства (property). В языке UML принята стандартная форма записи атрибутов класса, которая подчиняется определенным синтаксическим правилам. Каждому атрибуту класса соответствует отдельная строка текста, которая состоит из квантора видимости атрибута, имени атрибута, его кратности, типа значений атрибута и, возможно, его исходного значения:

<Квантор видимости атрибута> <Имя атрибута>
[*<Кратность атрибута>*] : *<Тип атрибута>*=
<Исходное значение атрибута> {*<Строка-свойство атрибута>*},

где угловыми скобками выделяются в тексте понятия, имеющие собственную структуру, без дополнительных обозначений приведены обязательные слова (в нашем случае — служебные символы).

Квантор видимости атрибута может принимать одно из трех возможных значений и, соответственно, отображается при помощи специальных символов:

— символ «+» обозначает атрибут с областью видимости типа «общедоступный» (public). Атрибут с этой областью видимости доступен в любом другом классе пакета, в котором определена рассматриваемая диаграмма;

— символ «#» обозначает атрибут с областью видимости типа «защищенный» (protected). Атрибут с этой областью видимости доступен только в подклассах (потомках) определяемого класса;

— символ «–» обозначает атрибут с областью видимости типа «закрытый» (private). Атрибут с этой областью видимости доступен только внутри определяемого класса.

Квантор видимости может быть опущен. В этом случае его отсутствие просто означает, что видимость атрибута не указывается. Эта ситуация отличается от принятых по умолчанию соглашений в традиционных языках программирования, когда отсутствие квантора видимости трактуется как «public» (язык Object Pascal) или «private» (язык C++). Вместо условных графических обозначений области видимости можно записывать соответствующее ключевое слово: **public**, **protected**, **private**.

Имя атрибута представляет собой строку текста, которая используется в качестве идентификатора соответствующего атрибута и поэтому должна быть уникальной в пределах данного класса. Имя атрибута является единственным обязательным элементом синтаксического обозначения атрибута. Примерами имен атрибутов являются: name, coordinate, visible, connect и т. п.

Кратность атрибута характеризует общее количество конкретных атрибутов данного типа, входящих в состав отдельного класса. По смыслу кратность атрибута более единицы соответствует понятию «массив». В общем случае кратность записывается в форме строки текста в квадратных скобках после имени соответствующего атрибута:

$$\mathbf{[НГ_1 .. ВГ_1, НГ_2 .. ВГ_2, ..., НГ_k .. ВГ_k],}$$

где $НГ_i$ — нижняя_граница и $ВГ_i$ — верхняя_граница ($НГ_i < ВГ_i$, i=1, 2, …, k) являются положительными целыми числами, каждая пара которых служит для обозначения отдельного замкнутого интервала целых чисел. В качестве верхней границы $ВГ_i$ может использоваться специальный символ «*», который означает произвольное положительное целое число. Другими словами, это означает неограниченное сверху значение кратности соответствующего атрибута.

Значения кратности из интервала следуют в монотонно возрастающем порядке без пропуска отдельных чисел, лежащих между нижней и верхней границами. При этом придерживаются следующего правила: соответствующие нижние и верхние границы интервалов включаются в значение кратности. Если в качестве кратности указывается единственное число, то кратность атрибута принимается равной данному числу. Если же указывается единственный знак «*», то это означает, что кратность атрибута может быть произвольным положительным целым числом или нулем.

В качестве примера рассмотрим следующие варианты задания кратности атрибутов:

— **[0 .. 1]** означает, что кратность атрибута может принимать значение 0 или 1. При этом 0 означает отсутствие значения для данного атрибута;

— **[0 .. *]** означает, что кратность атрибута может принимать любое положительное целое значение, большее или равное 0. Эта кратность может быть записана короче в виде **[*]**;

— **[1 : *]** означает, что кратность атрибута может принимать любое положительное целое значение, большее или равное 1;

— **[1 .. 5]** означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 4, 5;

— **[1 .. 3, 5, 7]** означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 5, 7;

— **[1 .. 3, 7 .. 10]** означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 7, 8, 9, 10;

— **[1 .. 3, 7 .. *]** означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, а также любое положительное целое значение, большее или равное 7.

Если кратность атрибута не указана, то по умолчанию принимается ее значение, равное **[1 .. 1]**, т. е. в точности 1. По смыслу кратность, равная единице, соответствует понятию «переменная».

Тип атрибута представляет собой выражение, семантика которого определяется языком спецификации соответствующей модели. В языке UML тип атрибута иногда определяется языком программирования, который предполагается использовать для реализации данной модели. В простейшем случае тип атрибута указывается строкой текста, имеющей осмысленное значение в пределах пакета или модели, к которым относится рассматриваемый класс.

Можно привести следующие примеры задания имен и типов атрибутов классов:

— **Color : TColor** — здесь **Color** является именем атрибута, **TColor** — именем типа данного атрибута;

— **Employee_Name [1 .. 3] : String** — здесь **Employee_Name** — имя_сотрудника, является именем атрибута, который служит для представления информации об имени конкретного сотрудника. Тип атрибута **String** — строка, как раз и указывает на тот факт, что отдельное значение имени представляет собой строку текста из одного или нескольких слов. Поскольку во многих языках программирования существует тип данных **String**, использование соответствующего англоязычного термина не вызывает недоразумения у большинства программистов. Однако, хотя в оригинальном языке UML все термины даются в англоязычном представлении, использование в данной ситуации **Имя_сотрудника [1 .. 3] : Строка** не исключается и определяется только соображениями удобства;

— **Visible : Boolean** — здесь **Visible** — видимость есть имя абстрактного атрибута, который может характеризовать наличие визуального представления соответствующего класса на экране монитора. В этом случае тип **Boolean** означает, что возможными значениями данного атрибута является одно из двух логических значений: истина (true) или ложь (false). При этом значение «истина» может соответствовать наличию графического изображения на экране монитора, а значение «ложь» — его отсутствию, о чем дополнительно может указываться

в пояснительном тексте. Поскольку кратность атрибута «видимость» не указана, она принимает значение 1 по умолчанию. В этой ситуации англоязычное имя типа атрибута вполне оправдано наличием соответствующего базового типа в языках программирования. Абстрактный характер данного атрибута обозначается курсивным текстом в записи данного атрибута;

— **Shape: TPolygon** — здесь имя атрибута **Shape** — форма, может характеризовать такой класс, который является геометрической фигурой на плоскости.

Исходное значение атрибута служит для задания некоторого начального значения для соответствующего атрибута в момент создания отдельного экземпляра класса. Здесь необходимо придерживаться правила принадлежности значения типу конкретного атрибута. Если исходное значение не указано, то значение соответствующего атрибута не определено на момент создания нового экземпляра класса.

В качестве примеров исходных значений атрибутов можно привести следующие варианты задания атрибутов:

— **Color: TColor =(255, 0, 0)** — в трехцветной модели цвета (Red, Green, Blue — Красный, Зеленый, Синий) это соответствует чистому красному цвету в качестве исходного значения для данного атрибута;

— **Employee_Name [1..3] : String=Кузьмин Иван Иванович**;

— **Visible: Boolean=True** — может соответствовать ситуации, когда в момент создания экземпляра класса создается видимое на экране монитора окно, соответствующее данному объекту;

— **Shape: TPolygon**=**Rectangle —** (прямоугольник) вряд ли требует комментариев, поскольку здесь речь идет о геометрической форме создаваемого объекта.

Строка-свойство атрибута служит для указания значений атрибута, которые не могут быть изменены в программе при работе с данным типом объектов. Фигурные скобки как раз и обозначают фиксированное значение соответствующего атрибута для класса в целом, которое должны принимать все вновь создаваемые экземпляры класса без исключения. Это значение принимается за исходное значение атрибута, которое не может быть переопределено в последующем. Отсутствие строки-свойства по умолчанию трактуется так, что значение соответствующего атрибута может быть изменено в программе. Например, строка-свойство в записи атрибута **Salary: Currency={15000}** может служить для обозначения фиксированной заработной платы для каждого объекта класса «Сотрудник» определенной должности в некоторой организации. С другой стороны, запись данного атрибута в виде **Salary: Currency=15000** означает уже нечто иное, а именно — при создании нового экземпляра Сотрудник (аналогия — прием на работу нового сотрудника) для него устанавливается по умолчанию заработная плата, равная 15 000 руб. Однако для отдельных сотрудников могут быть сделаны исключения, как в большую, так и в меньшую сторону, о чем необходимо позаботиться дополнительно в программе.

Операции. В третьей сверху секции класса записываются операции или методы класса. Операция представляет собой некоторый сервис, который может предоставить каждый экземпляр класса по определенному требованию.

Совокупность операций характеризует ответственность экземпляра класса. Запись операций класса в языке UML также стандартизована и подчиняется определенным синтаксическим правилам. При этом каждой операции класса соответствует отдельная строка, которая состоит из квантора видимости операции, имени операции, выражения типа, возвращаемого операцией значения и, возможно, строки-свойства данной операции:

<Квантор видимости операции> <Имя операции>
(Список параметров операции):
<Выражение типа, возвращаемого операцией значения>
{строка-свойство операции}.

Квантор видимости операции, как и в случае атрибутов класса, может принимать одно из трех возможных значений и, соответственно, отображается при помощи специального символа. Символ «+» обозначает операцию с областью видимости типа общедоступный (**public**). Символ «#» обозначает операцию с областью видимости типа защищенный (**protected**). И, наконец, символ «–» используется для обозначения операции с областью видимости типа закрытый (**private**).

Квантор видимости для операции может быть опущен. В этом случае его отсутствие просто означает, что видимость операции не указывается. Вместо условных графических обозначений также можно записывать соответствующее ключевое слово: **public**, **protected**, **private**.

Имя операции представляет собой строку текста, которая используется в качестве идентификатора соответствующей операции и поэтому должна быть уникальной в пределах данного класса. Имя операции является единственным обязательным элементом синтаксического обозначения операции.

Список параметров операции является перечнем разделенных запятой формальных параметров, каждый из которых может быть представлен в следующем виде:

<Вид параметра операции> <Имя параметра операции> :
<Выражение типа параметра операции>=
<Значение параметра операции по умолчанию>.

Вид параметра операции задается одним из ключевых слов **in**, **out** или **inout** со значением **in** по умолчанию, в случае, если вид параметра не указывается.

Имя параметра операции есть идентификатор соответствующего формального параметра.

Выражение типа параметра операции является зависимой от конкретного языка программирования спецификацией типа возвращаемого значения для соответствующего формального параметра.

Значение параметра операции по умолчанию в общем случае представляет собой выражение для значения формального параметра, синтаксис которого зависит от конкретного языка программирования и подчиняется принятым в нем ограничениям.

Выражение типа, возвращаемого операцией значения, также является зависимой от языка реализации спецификацией типа или типов значений пара-

метров, которые возвращаются объектом после выполнения соответствующей операции. Двоеточие и выражение типа возвращаемого значения могут быть опущены, если операция не возвращает никакого значения. Для указания кратности возвращаемого значения данная спецификация может быть записана в виде списка отдельных выражений.

Строка-свойство операции служит для указания значений свойств, которые могут быть применены к данному элементу. Строка-свойство не является обязательной, она может отсутствовать, если никакие свойства не специфицированы.

Операция-селектор, которая не может изменять состояние системы и, соответственно, не имеет никакого побочного эффекта, обозначается строкой-свойством {**запрос**} или {**query**}. В противном случае предполагается операция-модификатор, которая может изменять состояние системы, хотя нет никаких гарантий, что она будет это делать.

Для повышения производительности системы одни операции могут выполняться параллельно или одновременно, а другие только последовательно. В этом случае для указания параллельности выполнения операции используется строка-свойство вида {**concurrency**=*разновидность*}, где *разновидность* может принимать одно из следующих значений: ***последовательная*** (**sequential**), ***параллельная*** (**concurrent**), ***охраняемая*** (**guarded**). При этом придерживаются следующей семантики для данных значений:

— *последовательная* — для данной операции необходимо обеспечить ее единственное выполнение в системе, одновременное выполнение других операций может привести к ошибкам или нарушениям целостности объектов класса;

— *параллельная* — данная операция в силу своих особенностей может выполняться параллельно с другими операциями в системе, при этом параллельность должна поддерживаться на уровне реализации модели;

— *охраняемая* — все обращения к данной операции должны быть строго упорядочены во времени с целью сохранения целостности объектов данного класса, при этом могут быть приняты дополнительные меры по контролю исключительных ситуаций на этапе ее выполнения.

С целью сокращения обозначений допускается использование одного имени в качестве строки-свойства для указания соответствующего значения параллельности. Отсутствие данной строки-свойства означает, что семантика параллельности для операции не определена. Поэтому следует предположить худший с точки зрения производительности случай, когда данная операция требует последовательного выполнения.

Появление сигнатуры операции на самом верхнем уровне объявляет эту операцию на весь класс, при этом данная операция наследуется всеми потомками данного класса. Если в некотором классе операция не выполняется (т. е. некоторый метод не применяется), то такая операция может быть помечена как *абстрактная* (abstract). Подчиненное появление записи данной операции без свойства {*абстрактная*} указывает на тот факт, что соответствующий класс-потомок может выполнять данную операцию в качестве своего метода.

Если для некоторой операции необходимо дополнительно указать особенности ее реализации (например, алгоритм), то это может быть сделано в форме примечания, записанного в виде текста, который присоединяется к записи операции в соответствующей секции класса. Если объекты класса принимают и реагируют на некоторый сигнал, то запись данной операции помечается ключевым словом *сигнал* (signal). Это обозначение равнозначно обозначению некоторой операции. Реакция объекта на прием сигнала может быть показана в виде некоторого автомата. Кроме других случаев это обозначение может быть использовано, чтобы показать реакцию объектов класса на ошибочные ситуации или исключения, которые могут моделироваться как сигналы или сообщения.

Поведение операции может быть указано дополнительно в форме присоединенного к операции примечания. В этом случае текст примечания заключается в скобки, если он представляет собой формальную спецификацию на некотором языке программирования и соответствует элементу «семантическое ограничение языка UML». В противном случае текст примечания является простым описанием на естественном языке и обозначается прямоугольником с загнутым верхним правым уголком.

Список формальных параметров и тип возвращаемого значения могут не указываться. Квантор видимости атрибутов и операций может быть указан в виде специального значка или символа, которые используются для графического представления моделей в некотором инструментальном средстве. При этом обязательной частью строки записи операции является наличие имени операции и круглых скобок.

В качестве примеров записи операций можно привести следующие обозначения отдельных операций:

— **+создать ()** может обозначать операцию по созданию отдельного объекта класса, которая является общедоступной и не содержит формальных параметров. Эта операция не возвращает никакого значения после своего выполнения;

— **+нарисовать(форма:Многоугольник=прямоугольник, цвет_заливки: Color=(0, 0, 255))** может обозначать операцию по изображению на экране монитора прямоугольной области синего цвета, если не указываются другие значения в качестве аргументов данной операции;

— **запросить_счет_клиента (номер_счета: Integer):Carrency** обозначает операцию по установлению наличия средств на текущем счете клиента банка. При этом аргументом данной операции является номер счета клиента, который записывается в виде целого числа (например, «123456»). Результатом выполнения этой операции является некоторое число, записанное в принятом денежном формате;

— **выдать_сообщение ():{“Ошибка деления на ноль”}** смысл данной операции не требует пояснения, поскольку содержится в строке-свойстве операции. Данное сообщение может появиться на экране монитора в случае попытки деления некоторого числа на ноль, что недопустимо.

Контрольные вопросы

1. Какие графические формы используются для обозначения класса?
2. Какие правила существуют для обозначения имени класса?
3. Какой прием используется для обозначения абстрактного класса?
4. Опишите синтаксис атрибута класса с пояснением правил обозначения компонентов атрибута.
5. Опишите синтаксис операции класса с пояснением правил обозначения компонентов описания операции.

2.2.2. Отношения между классами

Кроме внутреннего устройства или структуры классов на диаграмме классов указываются различные отношения между классами. При этом совокупность типов таких отношений фиксирована в языке UML и предопределена семантикой этих типов отношений. Базовыми отношениями, или связями, в языке UML являются:

— отношение зависимости (dependency relationship);
— отношение ассоциации (association relationship);
— отношение обобщения (generalization relationship);
— отношение реализации (realization relationship).

Каждое из этих отношений имеет собственное графическое представление на диаграмме, которое отражает взаимосвязи между объектами соответствующих классов.

Отношение зависимости. Отношение зависимости используется в такой ситуации, когда некоторое изменение одного элемента модели может потребовать изменения другого зависимого от него элемента модели. Например, некоторый класс А использует методы другого класса В; изменение класса В может потребовать внесения изменений в класс А.

Отношение зависимости графически изображается пунктирной линией между соответствующими элементами со стрелкой на одном из ее концов. На диаграмме классов данное отношение связывает отдельные классы между собой, при этом стрелка направлена от класса-клиента зависимости к независимому классу или классу-источнику. На рисунке 2.5 изображены два класса: Класс_А и Класс_Б, при этом Класс_Б является источником некоторой зависимости, а Класс_А — клиентом этой зависимости.

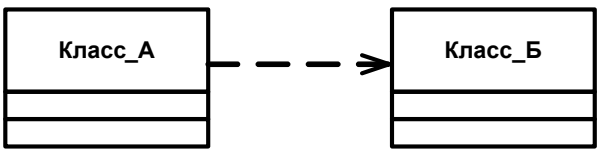


Рис. 2.5
Графическое изображение отношения зависимости
на диаграмме классов

Стрелка может помечаться необязательным, но стандартным ключевым словом в кавычках и необязательным индивидуальным именем. Для отношения зависимости предопределены ключевые слова, которые обозначают некоторые

специальные виды зависимостей. Эти ключевые слова (стереотипы) записываются в кавычках рядом со стрелкой, которая соответствует данной зависимости. Примеры стереотипов для отношения зависимости представлены ниже:

— **«access»** (доступ) — служит для обозначения доступности открытых атрибутов и операций класса-источника для класса-клиента;

— **«bind»** (связывать) — класс-клиент может использовать некоторый шаблон для своей последующей параметризации;

— **«derive»** (извлекать) — атрибуты класса-клиента могут быть вычислены по атрибутам класса-источника;

— **«import»** (импортировать) — открытые атрибуты и операции класса-источника становятся частью класса-клиента, как если бы они были объявлены непосредственно в нем;

— **«refine»** (уточнять) — указывает, что класс-клиент служит уточнением класса-источника в силу причин исторического характера, когда появляется дополнительная информация в ходе работы над проектом.

Отношение зависимости является наиболее общей формой отношения в языке UML. Все другие типы рассматриваемых отношений можно считать частным случаем данного отношения. Чаще всего отношение зависимости является временным в процессе разработки диаграммы классов. По мере уточнения модели оно конкретизируется отношением ассоциации или отношением обобщения, или отношением реализации.

Пример отношения зависимости представлен на рисунке 2.6.

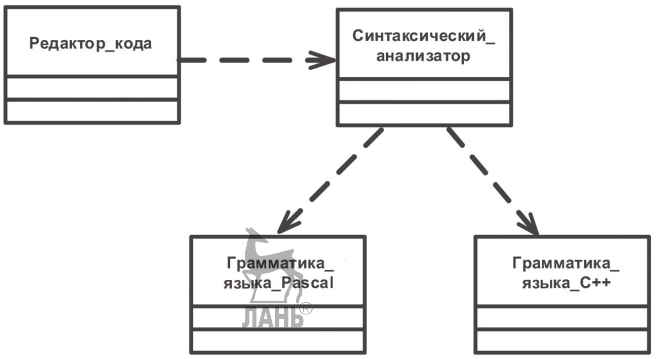


Рис. 2.6
Отношение зависимости между классами

Отношение ассоциации. ***Ассоциация*** (association) — семантическое отношение между двумя и более классами, которое специфицирует характер связи между соответствующими экземплярами этих классов.

Отношение ассоциации соответствует наличию произвольного отношения или взаимосвязи между классами. Данное отношение обозначается сплошной линией со стрелкой или без нее и с дополнительными символами, которые характеризуют специальные свойства ассоциации.

В качестве дополнительных специальных символов могут использоваться имя ассоциации, а также имена и кратность классов-ролей ассоциации. Имя ас-

социации является необязательным элементом ее обозначения. Если оно задано, то записывается с заглавной (большой) буквы рядом с линией соответствующей ассоциации.

Наиболее простой случай данного отношения — бинарная ассоциация. Она связывает в точности два класса и, как исключение, может связывать класс с самим собой. Для бинарной ассоциации на диаграмме может быть указан порядок следования классов. Направление этой стрелки указывает на порядок классов, один из которых является первым (исход стрелки), а другой — вторым (вход стрелки). Использовании линии без стрелки означает, что порядок следования классов в рассматриваемом отношении не определен.

В качестве простого примера отношения бинарной ассоциации рассмотрим отношение между двумя классами — классом «Компания» и классом «Сотрудник» (рис. 2.7). Они связаны между собой бинарной ассоциацией «Работа», имя которой указано на рисунке рядом со стрелкой ассоциации. Для данного отношения определен порядок следования классов, первым из которых является класс «Сотрудник», а вторым — класс «Компания». Отдельным примером или экземпляром данного отношения может являться пара значений (Петров И. И., «Рога&Копыта»). Это означает, что сотрудник Петров И. И. работает в компании «Рога&Копыта».

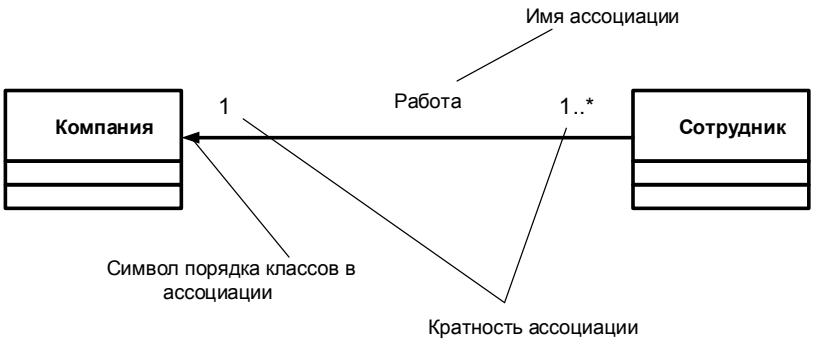


Рис. 2.7
Графическое изображение отношения бинарной ассоциации между классами

Тернарная ассоциация и ассоциации более высокой арности в общем случае называются N-арной ассоциацией (читается — «эн арная ассоциация»). Такая ассоциация связывает некоторым отношением три и более классов, при этом один класс может участвовать в ассоциации более чем один раз. Каждый класс ассоциации играет определенную роль в соответствующем отношении, что может быть явно указано на диаграмме. Каждый экземпляр N-арной ассоциации представляет собой N-арный кортеж значений объектов из соответствующих классов. Бинарная ассоциация является частным случаем N-арной ассоциации, когда значение N=2, и имеет свое собственное обозначение.

N-арная ассоциация графически обозначается ромбом, от которого ведут линии к символам классов данной ассоциации. В этом случае ромб соединяется с символами соответствующих классов сплошными линиями. Обычно линии

проводятся от вершин ромба или от середины его сторон. Имя N-арной ассоциации записывается рядом с ромбом соответствующей ассоциации.

Порядок классов в N-арной ассоциации, в отличие от порядка множеств в отношении, на диаграмме не фиксируется. Некоторый класс может быть присоединен к ромбу пунктирной линией. Это означает, что данный класс обеспечивает поддержку свойств соответствующей N-арной ассоциации, а сама N-арная ассоциация имеет атрибуты, операции и/или ассоциации. Другими словами, такая ассоциация, в свою очередь, является классом с соответствующим обозначением в виде прямоугольника и самостоятельным элементом языка UML — ассоциацией-классом (Association Class). N-арная ассоциация не может содержать символ агрегации ни для какой из своих ролей.

В качестве примера конкретной тернарной ассоциации рассмотрим отношение между тремя классами: «Футбольная команда», «Год» и «Игра». Данная ассоциация указывает на наличие отношения между этими тремя классами, которое может представлять информацию об играх футбольных команд в национальном чемпионате в течение нескольких последних лет (рис. 2.8).

Рис. 2.8
Графическое изображение тернарной ассоциации между тремя классами

Следующий элемент обозначений — кратность отдельных классов, являющихся концами ассоциации. Кратность отдельного класса обозначается в виде интервала целых чисел, аналогично кратности атрибутов и операций классов. Интервал записывается рядом с концом ассоциации и для N-арной ассоциации означает потенциальное число отдельных экземпляров или значений кортежей этой ассоциации, которые могут иметь место, когда остальные N-1 экземпляров или значений классов фиксированы.

Так, для рассмотренного ранее примера (рис. 2.7) кратность «1» для класса «Компания» означает, что каждый сотрудник может работать только в одной компании. Кратность «1..*» для класса «Сотрудник» означает, что в каждой компании могут работать несколько сотрудников, общее число которых заранее неизвестно и ничем не ограничено. Заметим, что вместо кратности «1..*» записать только символ «*» нельзя, поскольку последний означает кратность «0..*». Для данного примера это означало бы, что отдельные компании могут совсем не иметь сотрудников в своем штате. Но такая кратность вполне приемлема в других ситуациях, как это видно из рассмотренного выше примера (рис. 2.7).

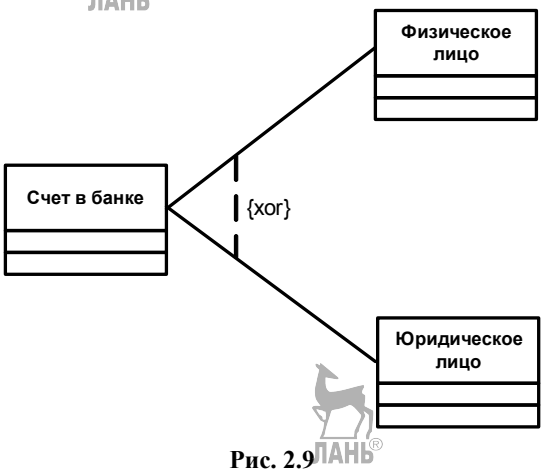
Что касается других свойств отношения, ассоциации, то в случае их наличия они могут рассматриваться в качестве атрибутов класса ассоциации и могут быть указаны на диаграмме обычным для класса способом в соответствующей секции прямоугольника класса.

Частным случаем отношения ассоциации является так называемая исключающая ассоциация (Xor-association). Семантика данной ассоциации указывает на тот факт, что из нескольких потенциально возможных вариантов данной ассоциации в каждый момент времени может использоваться только один ее экземпляр. На диаграмме классов исключающая ассоциация изображается пунктирной линией, соединяющей две и более ассоциации, рядом с которой записывается строка-ограничение «{xor}».

Например, счет в банке может быть открыт либо для физического лица, либо для юридического лица, что изображается с помощью исключающей ассоциации (рис. 2.9).

Специальной формой или частным случаем отношения ассоциации является отношение агрегации, которое, в свою очередь, тоже имеет специальную форму — отношение композиции. Поскольку эти отношения имеют свои специальные обозначения и относятся к базовым понятиям языка UML, рассмотрим их последовательно.

ЛАНЬ®



ЛАНЬ®

Рис. 2.9

Графическое изображение исключающей ассоциации между тремя классами

Отношение агрегации. Напомним, что отношение агрегации имеет место между несколькими классами в том случае, если один из классов представляет собой некоторую сущность, включающую в себя в качестве составных частей другие сущности.

Графически отношение агрегации изображается сплошной линией, один из концов которой представляет собой не закрашенный внутри ромб. Этот ромб указывает на тот из классов, который представляет собой «целое». Остальные классы являются его «частями» (рис. 2.10).

Примером отношения агрегации может служить известное каждому из читателей деление персонального компьютера на составные части: системный

блок, монитор, клавиатуру и мышь. Используя обозначения языка UML, компонентный состав ПК можно представить в виде соответствующей диаграммы классов (рис. 2.11), которая в данном случае иллюстрирует отношение агрегации.

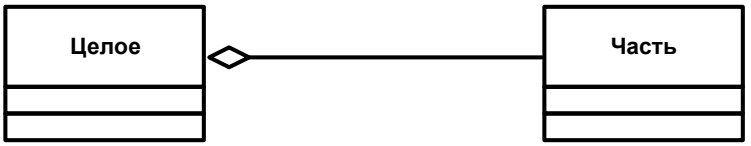


Рис. 2.10
Графическое изображение отношения агрегации в языке UML

Рис. 2.11
Диаграмма классов для иллюстрации отношения агрегации
на примере персонального компьютера

Отношение композиции. Отношение композиции является частным случаем отношения агрегации. Это отношение служит для выделения специальной формы отношения «часть-целое», при которой составляющие части в некотором смысле находятся внутри целого. Специфика взаимосвязи между ними заключается в том, что части не могут выступать в отрыве от целого, т. е. с уничтожением целого уничтожаются и все его составные части.

Возможно, не самый лучший, но наверняка понятный всем пример этого отношения представляет собой состоящая из частей живая клетка в биологии. Другой пример — окно интерфейса программы, которое может состоять из строки заголовка, кнопок управления размером, полос прокрутки, главного меню, рабочей области и строки состояния. Нетрудно понять, что подобное окно представляет собой класс, а его компоненты являются как классами, так и атрибутами или свойствами окна. Последнее обстоятельство весьма характерно для отношения композиции, поскольку создание класса «Окно» должно сопровождаться автоматическим созданием его составляющих, а при ликвидации окна также должны быть ликвидированы его компоненты.

Графически отношение композиции изображается сплошной линией, один из концов которой представляет собой закрашенный внутри ромб. Этот ромб указывает на тот из классов, который представляет собой класс-композицию или «целое». Остальные классы являются его «частями» (рис. 2.12).

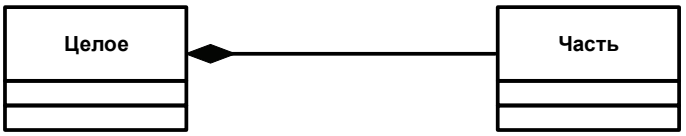


Рис. 2.12
Графическое изображение отношения композиции в языке UML

В качестве дополнительных обозначений для отношений композиции и агрегации могут использоваться дополнительные обозначения, применяемые для отношения ассоциации. А именно: указание кратности класса ассоциации и имени данной ассоциации, которые не являются обязательными. Применительно к упомянутому выше примеру класса «Окно_программы» его диаграмма классов может иметь следующий вид (рис. 2.13).

Данный пример может иллюстрировать и другие особенности разрабатываемой компьютерной программы, которые не указывались в явном виде при описании этого примера. Так, в частности, указание кратности 1 рядом с классом «Рабочая_область» характерно для однодокументных приложений.

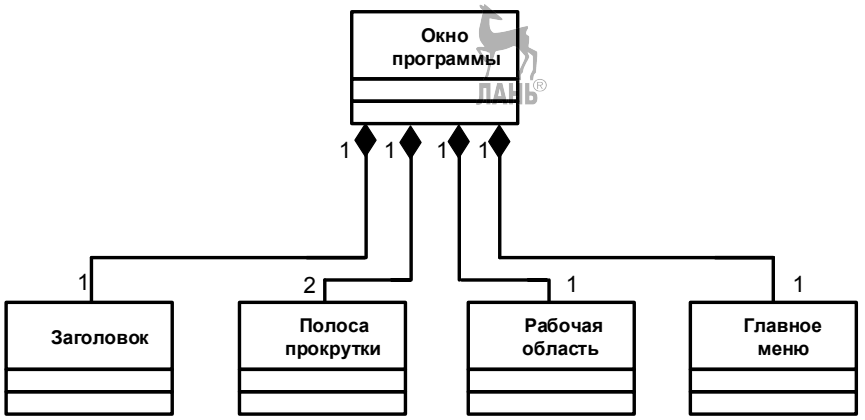


Рис. 2.13
Диаграмма классов для иллюстрации отношения композиции
на примере класса окна программы

Отношение обобщения. Отношение обобщения является обычным таксономическим отношением (отношением классификации) между более общим элементом (родителем или предком) и более частным или специальным элементом (дочерним или потомком).

Применительно к диаграмме классов данное отношение описывает иерархическое строение классов и наследование их свойств и поведения. При этом предполагается, что класс-потомок обладает всеми свойствами и поведением класса-предка, а также имеет свои собственные свойства и поведение, которые отсутствуют у класса-предка. На диаграммах отношение обобщения обозначается сплошной линией с треугольной стрелкой на одном из концов (рис. 2.14). Стрелка указывает на более общий класс (класс-предок или суперкласс), а ее отсутствие — на более специальный класс (класс-потомок или подкласс).

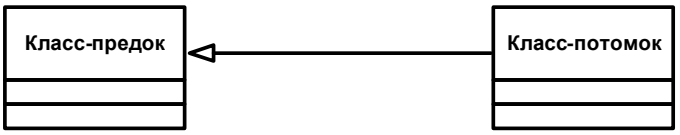


Рис. 2.14
Графическое изображение отношения обобщения в языке UML

Как правило, на диаграмме может указываться несколько линий для одного отношения обобщения, что отражает его таксономический характер. В этом случае более общий класс разбивается на подклассы одним отношением Обобщения. Например, класс **Геометрическая_фигура_на_плоскости** может выступать в качестве суперкласса для подклассов, соответствующих конкретным геометрическим фигурам, таким как **Прямоугольник**, **Окружность**, **Эллипс**, **Треугольник**. Данный факт может быть представлен графически в форме диаграммы классов следующего вида (рис. 2.15).

Это обозначение по форме соответствует дереву. В этом случае класс-предок является корнем этого дерева, а классы-потомки — его листьями.

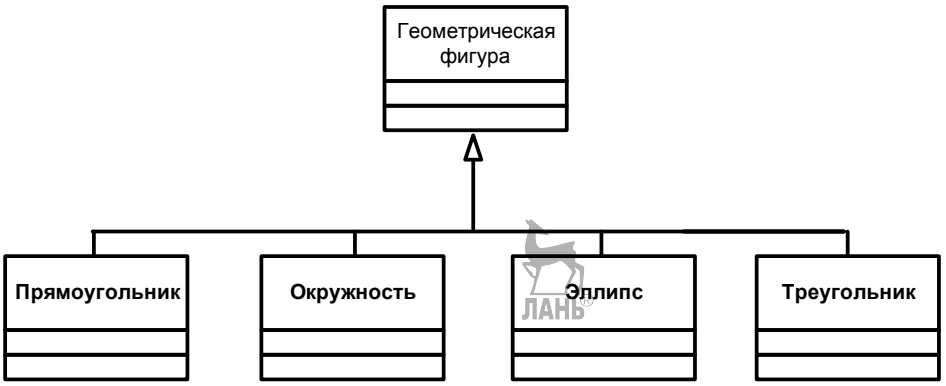


Рис. 2.15
Пример графического изображения отношения обобщения классов

Отношение реализации. *Отношение реализации* (реализация, realization) возникает между классами в том случае, когда один класс задает требования к поведению системы (функциональную *спецификацию*), а другой является полной или частичной программной, или аппаратной реализацией этого поведения. Другими словами, *реализацией* называется семантическое отношение между классификаторами, при котором один из них описывает контракт, а другой гарантирует его выполнение. В UML и, соответственно, в языках программирования определены специальные классы — *интерфейсы*, предназначенные для отдельного перечисления множества видимых операций одного или совокупности других классов без указания деталей реализации. Поскольку одна и та же спецификация может быть реализована многими способами, а реализация может относиться к нескольким спецификациям, то множественность этой связи M:N («многие ко многим»). Тот, кто реализует, называется *клиентом*, а носитель спецификации называется *источником* (*поставщиком*). Клиент не насле-

дует операции источника, он их определяет вновь или наследует от своего предка, однако для полноты должны быть объявлены все операции источника. Отношение реализации изображается пунктирной стрелкой с не закрашенным треугольным наконечником, идущей от реализации к спецификации (от клиента к поставщику). В реализацию могут быть добавлены дополнительные операции (по сравнению со спецификацией), если они способствуют выполнению основных операций.

Пожалуй, самой главной особенностью интерфейсов является то, что они позволяют отделить спецификацию контракта (сам интерфейс) от их реализации (классом или компонентом). Кроме того, интерфейсы могут пересекать границу между логическими и физическими частями системной архитектуры. Например, как видно из рисунка 2.16, класс в представлении системы с точки зрения проектирования (скажем, **AccountBusinessRules** в системе ввода заказов) может осуществлять реализацию интерфейса (**IRuleAgent**).

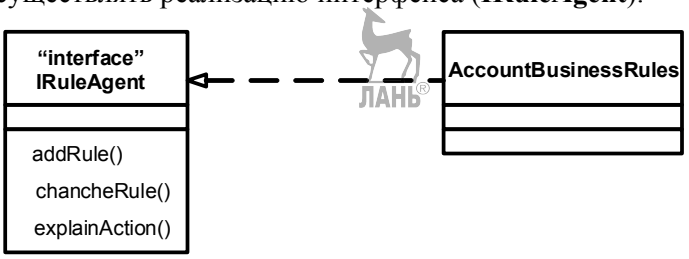


Рис. 2.16
Реализация интерфейса

Контрольные вопросы

1. Что представляет собой отношение зависимости?
2. Как изображается отношение зависимости на диаграмме классов?
3. В чем особенность отношения ассоциации?
4. Какие средства используются для изображения на диаграмме классов отношения ассоциации?
5. Какое отношение между классами называется отношением агрегации?
6. Как изображается на диаграмме классов отношение агрегации?
7. В каких ситуациях используется отношение композиции?
8. Какие изобразительные средства применяются для описания на диаграмме классов отношения композиции?
9. Что представляет собой отношение обобщения?
10. Какие средства используются для обозначения на диаграмме классов отношения обобщения?
11. В чем суть отношения реализации на диаграмме классов?
12. Какие средства используются для обозначения на диаграмме классов отношения реализации?

2.2.3. Шаблоны или параметризованные классы

Шаблон (template), или параметризованный класс (parametrized class), предназначен для обозначения такого класса, который имеет один (или более)

нефиксированный формальный параметр. Он определяет целое семейство или множество классов, каждый из которых может быть получен связыванием этих параметров с действительными значениями. Обычно параметрами шаблонов служат типы атрибутов классов, такие как целые числа, перечисление, массив строк и др. В более сложном случае формальные параметры могут представлять и операции класса.

Графически шаблон изображается прямоугольником, к верхнему правому углу которого присоединен маленький прямоугольник из пунктирных линий (рис. 2.17), большой прямоугольник может быть разделен на секции, аналогично обозначению для класса. В верхнем прямоугольнике указывается список формальных параметров для тех классов, которые могут быть получены на основе данного шаблона. В верхней секции шаблона записывается его имя по правилам записи имен для классов.

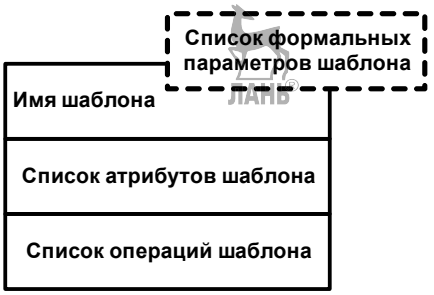


Рис. 2.17
Графическое изображение шаблона на диаграмме классов

Шаблон не может быть непосредственно использован в качестве класса, поскольку содержит неопределенные параметры. Чаще всего в качестве шаблона выступает некоторый суперкласс, параметры которого уточняются в его классах-потомках. Очевидно, в этом случае между ними существует отношение зависимости с ключевым словом «bind», когда класс-клиент может использовать некоторый шаблон для своей последующей параметризации. В более частном случае между шаблоном и формируемым от него классом имеет место отношение обобщения с наследованием свойств шаблона (рис. 2.18). В данном примере отмечен тот факт, что класс «Адрес» может быть получен из шаблона **Линейный_список** на основе актуализации формальных параметров S, I, J фактическими атрибутами «улица, дом, квартира».

Этот же шаблон может использоваться для задания (инстанцирования) другого класса, скажем, класса **Точки_на_плоскости**. В этом случае класс **Точки_на_плоскости** актуализирует те же формальные параметры, но с другими значениями, например, «bind<название_точки, *x*, *y*>». Концепция шаблонов является достаточно мощным средством в объектно-ориентированном программировании, и поэтому ее использование в языке UML позволяет не только сократить размеры диаграмм, но и наиболее корректно управлять наследованием свойств и поведения отдельных элементов модели.

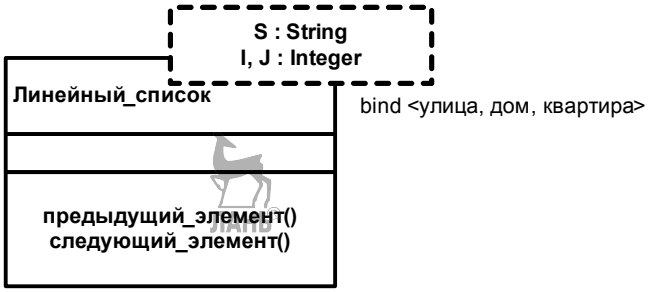


Рис. 2.18
Пример использования шаблона на диаграмме классов

Контрольные вопросы

1. Что называется шаблоном?
2. Почему шаблон называется параметризованным классом?
3. Приведите пример шаблона.

2.2.4. Повторное использование диаграмм классов

Процесс разработки диаграммы классов занимает центральное место в объектно-ориентированном анализе и проектировании сложных систем. От умения правильно выбрать классы и установить между ними взаимосвязи часто зависит не только успех процесса проектирования, но и производительность выполнения программы. Как показывает практика, каждый программист в своей работе стремится в той или иной степени использовать уже накопленный личный опыт при разработке новых проектов. Это обусловлено желанием свести новую задачу к уже решенным задачам, чтобы иметь возможность использовать не только проверенные фрагменты программного кода, но и отдельные компоненты в целом (библиотеки компонентов).

Такой стереотипный подход позволяет существенно сократить сроки реализации проекта, однако приемлем лишь в том случае, когда новый проект концептуально и технологически не слишком отличается от предыдущих проектов. В противном случае платой за сокращение сроков проекта может стать его реализация на устаревшей технологической базе. Что касается собственно объектной структуризации предметной области, то здесь уместно придерживаться тех рекомендаций, которые накоплены в объектно-ориентированном программировании.

Контрольные вопросы

1. Какие мотивы стимулируют повторное использование диаграмм классов?

2.2.5. Пример диаграммы классов

В качестве примера возьмем отдел кадров некоторого предприятия и построим его модель. Мы будем постепенно вводить понятия нашей предметной области и строить отношения между понятиями.

Начнем с базовых понятий **Man** (человек) и **Employee** (сотрудник).

Класс Man характеризуется свойством name (фамилия) и методами, позволяющими установить и прочитать значение свойства (рис. 2.19). Для построения объекта введем соответствующий конструктор. Можно также включить в класс Man паспортные данные, пол и т. п., а также операции над указанными свойствами, но в целях экономии места мы эти свойства опустим.

Теперь введем понятие **Employee**, включающее свойство position (должность), конструктор и операции над должностью (рис. 2.19).

Класс **Man** является более абстрактным, а класс **Employee** — специализированным. Поэтому вводим между классами отношение наследования.

Добавим теперь в модель понятие **IdCard** (пропуск). Результат этого действия показан на рисунке 2.20.

Отношение между классами представляет собой ассоциацию, в нашем случае это означает отношение между экземплярами соответствующих классов. Класс **Employee** имеет поле **card**, у которого тип **IdCard**, также класс имеет методы для присваивания значения(**setIdCard**) этому полю и для получения значения (**getIdCard**). Из экземпляра объекта **Employee** мы можем узнать о связанном с ним объектом типа IdCard, поэтому стрелочка на линии направлена от **Employee** к **IdCard**. Каждому сотруднику соответствует только один пропуск, верно и обратное. Поэтому мощность отношения равна 1:1.

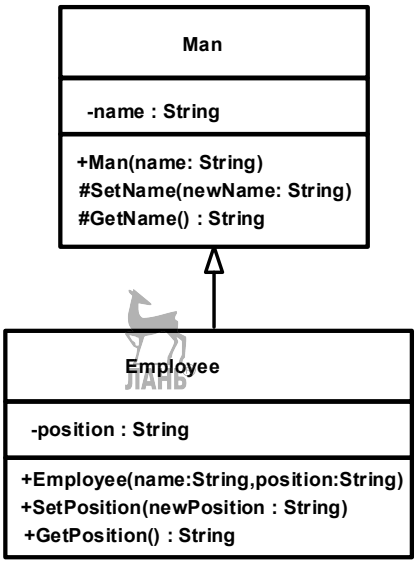


Рис. 2.19
Отношение наследования

Продолжим построение модели отдела кадров. Обычно каждое предприятие состоит из отделов, поэтому введем в модель класс **Department** (отдел) — наше предприятие структурировано по отделам. В каждом отделе может работать один или более человек. Можно сказать, что отдел включает в себя одного или более сотрудников и таким образом их агрегирует. На предприятии могут быть сотрудники, которые не принадлежат ни одному отделу, например директор предприятия.

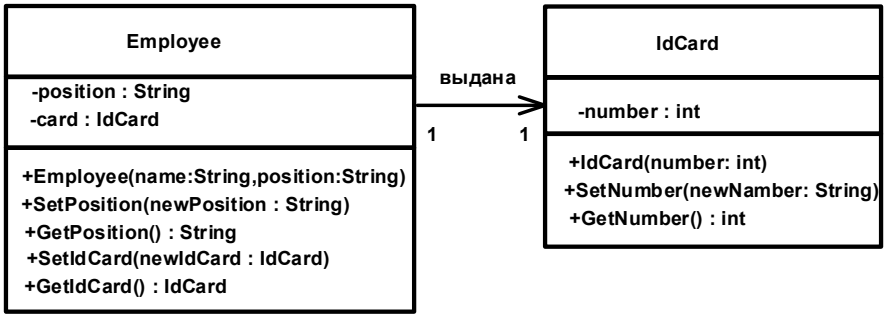


Рис. 2.20
Ассоциация

Наш класс **Department**, помимо конструктора и метода изменения имени отдела, имеет методы для занесения в отдел нового сотрудника, для удаления сотрудника и для получения всех сотрудников, входящих в данный отдел.

Навигация на диаграмме не показана, значит, она является двунаправленной: от объекта типа «Department» можно узнать о сотруднике и от объекта типа «Employee» можно узнать, к какому отделу он относится.

Так как нам нужно легко узнавать, к какому отделу относится сотрудник, то добавим в класс **Employee** поле и методы для назначения и получения отдела. Наконец, соединим классы отношением агрегации. Результаты наших действий приведены на рисунке 2.21.

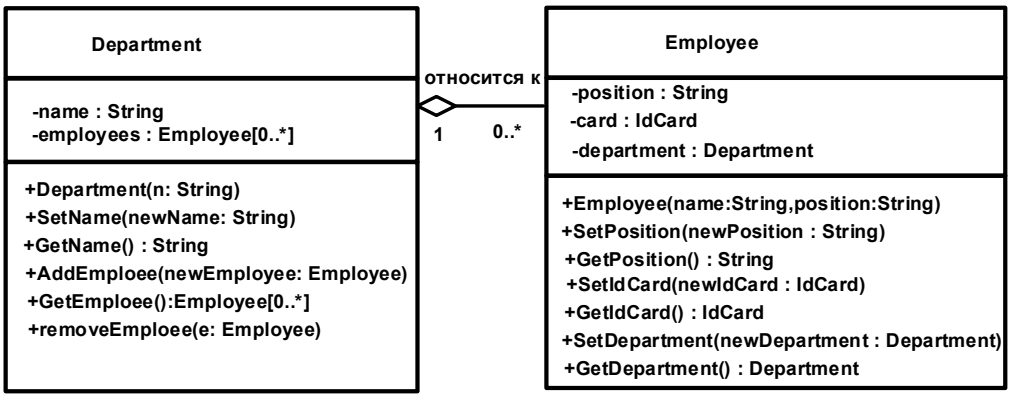


Рис. 2.21
Агрегация

Предположим, что одним из требований к нашей системе является требование о том, чтобы хранить данные о прежней занимаемой должности на предприятии. Введем новый класс **PastPosition**. В него, помимо свойства «имя» (name), введем и свойство «department», которое свяжет его с классом **Department**.

Данные о прошлых занимаемых должностях являются частью данных о сотруднике, таким образом, между ними связь целое-часть и в то же время данные о прошлых должностях не могут существовать без объекта типа **Employee**. Уничтожение объекта **Employee** должно привести к уничтожению объектов

PastPosition. Поэтому соединим классы **Employee** и **PastPosition** отношением «композиция».

В класс **Employee** добавим свойства и методы для работы с данными о прошлой должности. Фрагмент структуры классов приведен на рисунке 2.22.

Для организации диалога с пользователем введем в систему класс **Menu**. Включим один метод **ShowEmployees**, который показывает список сотрудников и их должности. Параметром для метода является массив объектов **Employee**. Таким образом, изменения, внесенные в класс **Employee,** могут потребовать и изменения класса **Menu**. Заметим, что класс **Menu** не относится к прикладной области, а представляет собой «системный» класс воображаемого приложения. Отношение зависимости приведено на рисунке 2.23.

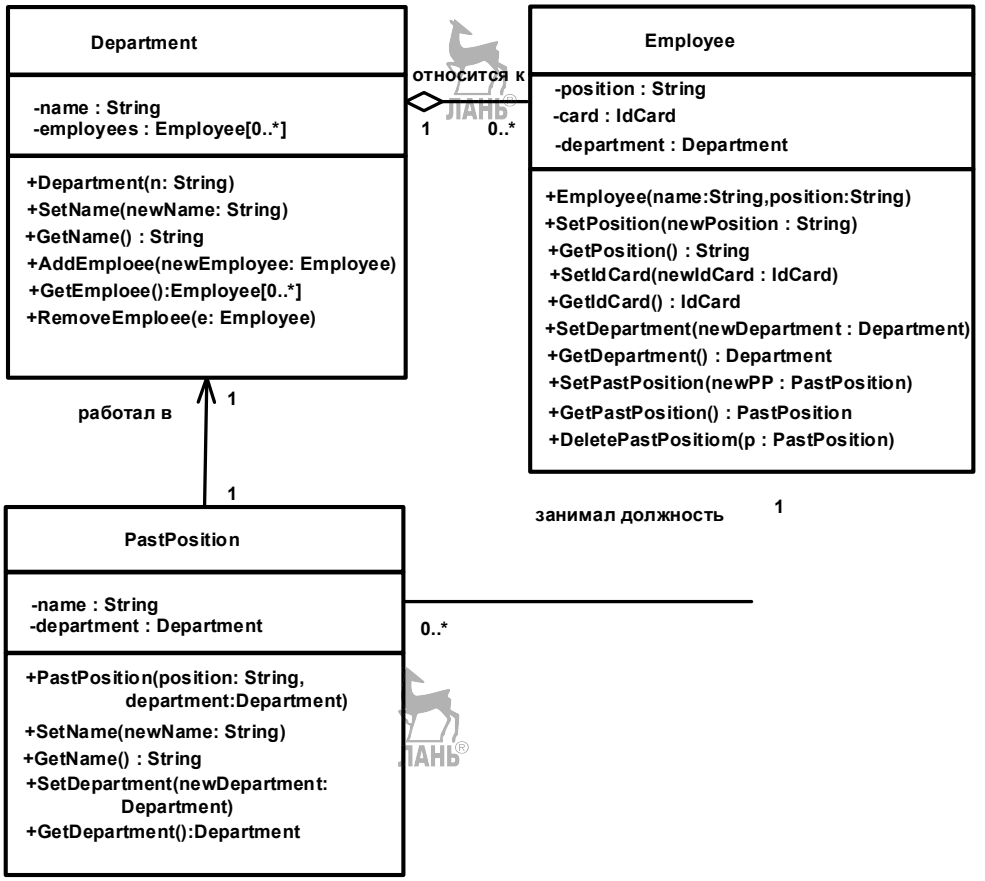


Рис. 2.22
Композиция

Для демонстрации отношения «реализация» создадим интерфейс **Unit**. Если представить, что организация может делиться не только на отделы, а, например, на цеха, филиалы и т. д., интерфейс **Unit** представляет собой самую абстрактную единицу деления. В каждой единице деления работает какое-то количество сотрудников, поэтому метод для получения количества работающих

людей будет актуален для каждого класса, реализующего интерфейс **Unit**. Отношение реализации показано на рисунке 2.24.

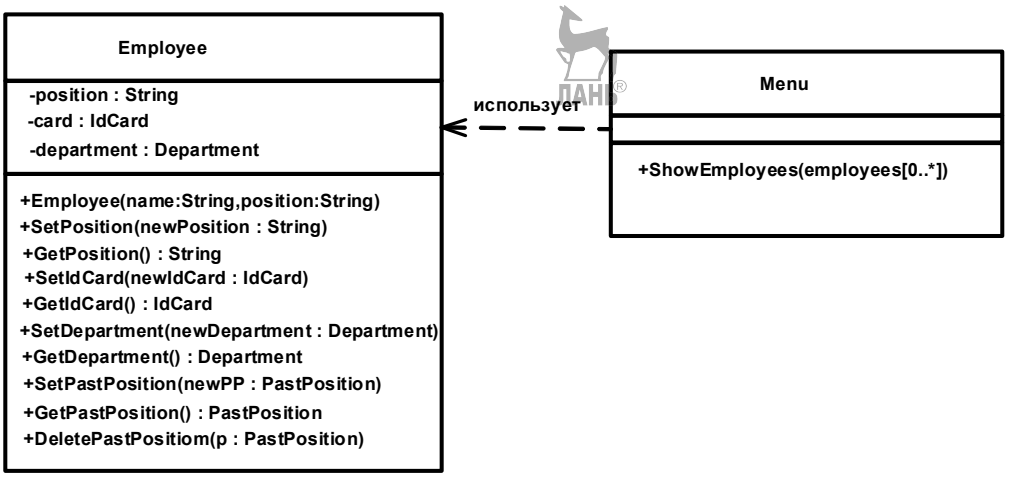


Рис. 2.23
Зависимость

На этом мы заканчиваем разработку структурной модели отдела кадров. Получившаяся в итоге диаграмма классов представлена на рисунке 2.25.

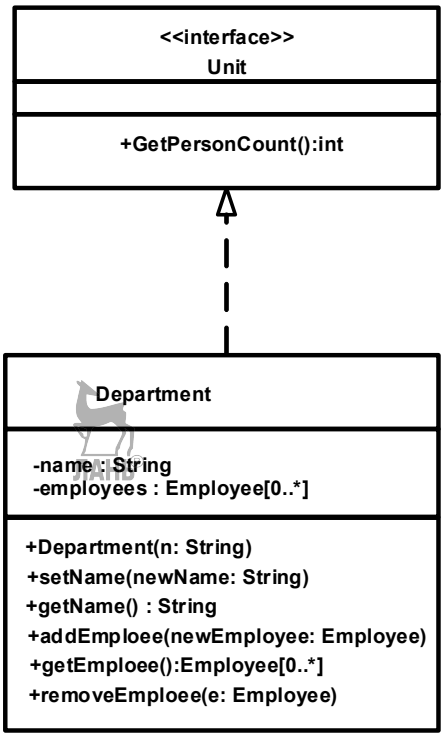


Рис. 2.24
Реализация

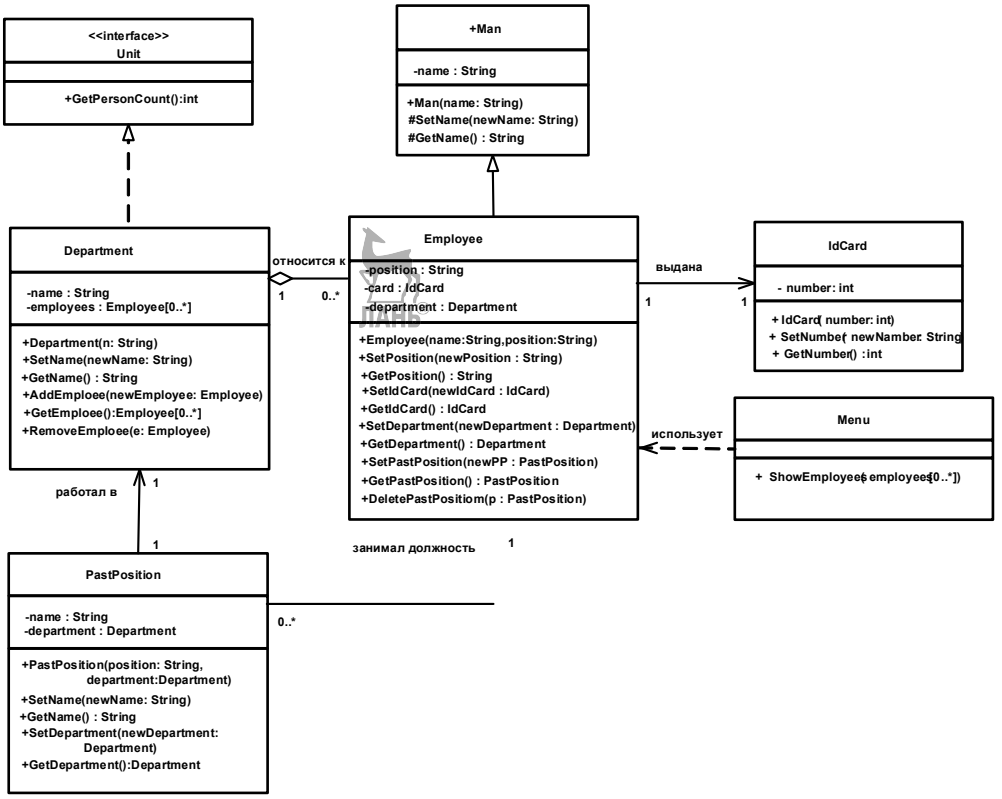


Рис. 2.25
Диаграмма классов отдела кадров

Контрольные вопросы

1. Какие рассуждения использовались при построении каждого вида отношений?

2.3. Диаграмма деятельности

При моделировании поведения проектируемой или анализируемой системы возникает необходимость не только представить процесс изменения ее состояний, но и детализировать особенности алгоритмической и логической реализации выполняемых системой операций. Традиционно для этой цели использовались блок-схемы или структурные схемы алгоритмов. Каждая такая схема акцентирует внимание на последовательности выполнения определенных действий или элементарных операций, которые в совокупности приводят к получению желаемого результата.

Важно подчеркнуть то обстоятельство, что с увеличением сложности системы строгое соблюдение последовательности выполняемых операций приобретает существенное значение. Если попытаться заварить кофе холодной водой, то мы можем только испортить одну порцию напитка. Нарушение последовательности операций при ремонте двигателя может привести к его поломке или выхо-

ду из строя. Катастрофические последствия могут произойти в случае отклонения от установленной последовательности действий при взлете или посадке авиалайнера, запуске ракеты, регламентных работах на атомной электростанции.

Для моделирования процесса выполнения операций в языке UML используются диаграммы деятельности (activity diagram). Диаграмма деятельности показывает зависимости между состояниями действий. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, а переход в следующее состояние срабатывает только при завершении этой, операции в предыдущем состоянии. Графически диаграмма деятельности представляется в форме графа деятельности, вершинами которого являются состояния действия, а дугами — переходы от одного состояния действия к другому.

Диаграммы деятельности позволяют реализовать в языке UML особенности процедурного и синхронного управления, обусловленного завершением внутренних деятельностей и действий. Основным направлением использования диаграмм деятельности является визуализация особенностей реализации операций классов, когда необходимо представить алгоритмы их выполнения. При этом каждое состояние действия может являться выполнением операции некоторого класса либо ее части, позволяя использовать диаграммы деятельности для описания реакций на внутренние события системы.

В контексте языка UML деятельность (activity) представляет собой некоторую совокупность отдельных вычислений, выполняемых автоматом. При этом отдельные элементарные вычисления могут приводить к некоторому результату или действию (action). На диаграмме деятельности отображается логика или последовательность перехода от одной деятельности к другой, при этом внимание фиксируется на результате деятельности. Сам же результат может привести к изменению состояния системы или возвращению некоторого значения.

Хотя диаграмма деятельности предназначена для моделирования динамических систем, время в явном виде отсутствует на этой диаграмме.

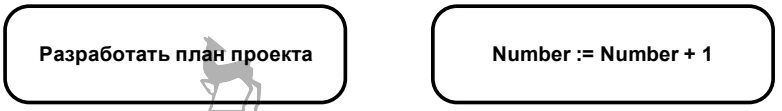
2.3.1. Состояние действия

Состояние действия (action state) является специальным случаем состояния с некоторым входным действием и, по крайней мере, одним выходящим из состояния переходом. Этот переход неявно предполагает, что входное действие уже завершилось. Состояние действия не может иметь внутренних переходов, поскольку оно является элементарным в рамках рассматриваемой диаграммы. Обычное использование состояния действия заключается в моделировании одного шага выполнения алгоритма (процедуры) или потока управления.

Графически состояние действия изображается фигурой, напоминающей прямоугольник, боковые стороны которого заменены выпуклыми дугами (рис. 2.26). Внутри этой фигуры записывается выражение действия (action-expression), которое должно быть уникальным в пределах одной диаграммы деятельности.

Действие может быть записано на естественном языке, некотором псевдокоде или языке программирования. Никаких дополнительных или неявных ограничений при записи действий не накладывается. Рекомендуется в качестве

имени простого действия использовать глагол с пояснительными словами (рис. 2.26, *а*). Если же действие может быть представлено в некотором формальном виде, то целесообразно записать его на том языке программирования, на котором предполагается реализовывать конкретный проект (рис. 2.26, *б*).



а, простое действие *б, выражение*

Рис. 2.26
Графическое изображение состояния действия

Иногда возникает необходимость представить на диаграмме деятельности некоторое сложное действие, которое, в свою очередь, состоит из нескольких более простых действий. В этом случае можно использовать специальное обозначение так называемого состояния поддеятельности (subactivity state). Такое состояние является графом деятельности и обозначается специальной пиктограммой в правом нижнем углу символа состояния действия (рис. 2.27). Эта конструкция может применяться к любому элементу языка UML, который поддерживает вложенность своей структуры. При этом пиктограмма может быть дополнительно помечена типом вложенной структуры.

Рис. 2.27
Графическое изображение состояния поддеятельности

Каждая диаграмма деятельности должна иметь единственное начальное и единственное конечное состояние. При этом каждая деятельность начинается в начальном состоянии и заканчивается в конечном состоянии. Саму диаграмму деятельности принято располагать таким образом, чтобы действия следовали сверху вниз. В этом случае начальное состояние будет изображаться в верхней части диаграммы, а конечное — в ее нижней части.

Контрольные вопросы

1. Какие графические формы используются для обозначения класса состояния действия?
2. Какие лингвистические приемы используется для обозначения состояния действия?
3. Какие графические формы используются для обозначения класса состояния поддеятельности?

2.3.2. Переходы

При построении диаграммы деятельности используются только переходы, которые срабатывают сразу после завершения деятельности или выполнения соответствующего действия. Этот переход переводит деятельность в последующее состояние сразу, как только закончится действие в предыдущем состоянии. На диаграмме такой переход изображается сплошной линией со стрелкой.

Если из состояния действия выходит единственный переход, то он может быть никак не помечен. Если же таких переходов несколько, то сработать может только один из них. Именно в этом случае для каждого из таких переходов должно быть явно записано в прямых скобках условие выбора перехода. При этом для всех выходящих из некоторого состояния переходов должно выполняться требование истинности только одного из них. Подобный случай встречается тогда, когда последовательно выполняемая деятельность должна разделиться на альтернативные ветви в зависимости от значения некоторого промежуточного результата. Такая ситуация обычно называется ветвлением, а для ее обозначения применяется специальный символ.

Графически ветвление на диаграмме деятельности обозначается небольшим ромбом, внутри которого нет никакого текста (рис. 2.28). В этот ромб может входить только одна стрелка от того состояния действия, после выполнения которого поток управления должен быть продолжен по одной из взаимно исключающих ветвей. Принято входящую стрелку присоединять к верхней или левой вершине символа ветвления. Выходящих стрелок может быть две или более, но для каждой из них явно указывается соответствующее сторожевое условие в форме булевского выражения.

В качестве примера рассмотрим фрагмент известного алгоритма нахождения корней квадратного уравнения. В общем случае после приведения уравнения второй степени к каноническому виду: $ax^2 + bx + c = 0$ необходимо вычислить его дискриминант. Причем в случае отрицательного дискриминанта уравнение не имеет решения на множестве действительных чисел, и дальнейшие вычисления должны быть прекращены. При неотрицательном дискриминанте уравнение имеет решение, корни которого могут быть получены на основе конкретной расчетной формулы.

Графически фрагмент процедуры вычисления корней квадратного уравнения может быть представлен в виде диаграммы деятельности с четырьмя состояниями действия и ветвлением (рис. 2.28). Каждый из переходов, выходящих из состояния «Вычислить дискриминант», имеет сторожевое условие, определяющее единственную ветвь, по которой может быть продолжен процесс вычисления корней в зависимости от знака дискриминанта. Очевидно, что в случае его отрицательности мы сразу попадаем в конечное состояние, тем самым завершая выполнение алгоритма в целом.

Первое из состояний действий рассматриваемого алгоритма следует считать состоянием поддеятельности, поскольку приведение квадратного уравнения к каноническому виду может потребовать нескольких элементарных дейст-

вий (приведение подобных и перенос отдельных членов уравнения из правой его части в левую). Поэтому для данного состояния целесообразно добавить соответствующую пиктограмму (рис. 2.28).

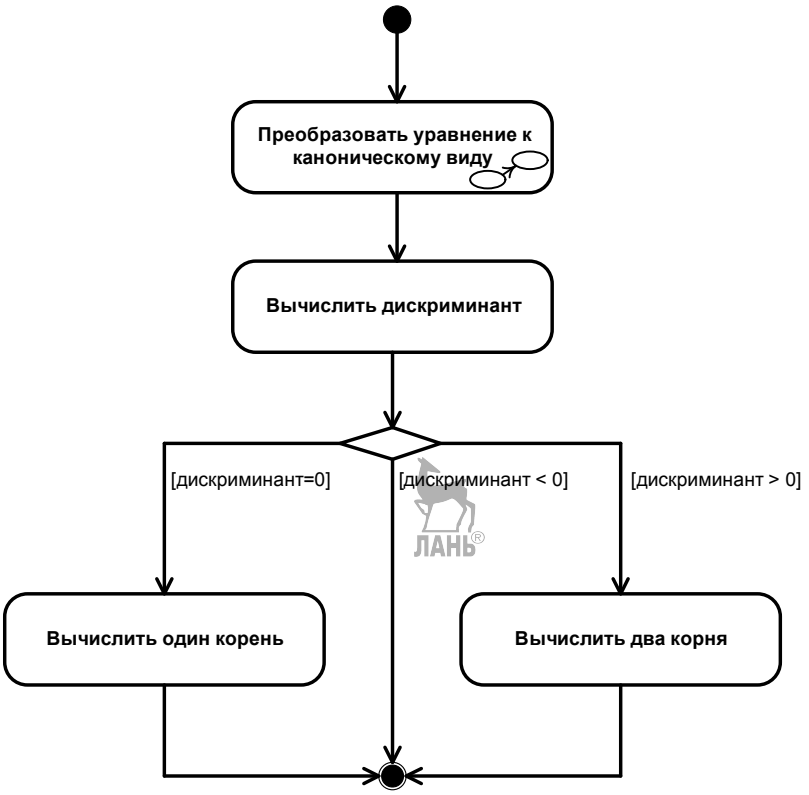


Рис. 2.28
Фрагмент диаграммы деятельности для алгоритма нахождения корней квадратного уравнения

В рассмотренном примере, как видно из рисунка 2.28, выполняемые действия соединяются в конечном состоянии. Однако это вовсе не является обязательным. Можно изобразить еще один символ ветвления, который будет иметь несколько входящих переходов и один выходящий.

В следующем примере (рис. 2.29) рассчитывается общая стоимость товаров, покупаемых по кредитной карточке в супермаркете. Если эта стоимость превышает 1000 руб., то выполняется аутентификация личности владельца карточки. В случае положительной проверки (карточка действительная) или если стоимость товаров не превышает 1000 руб., происходит снятие суммы со счета и оплата стоимости товаров. При отрицательном результате (карточка недействительная) оплаты не происходит и товар остается у продавца. Как видно из этого же рисунка, допускается использовать вместо сторожевого условия слово «иначе», указывающее на тот переход, который должен сработать в случае невыполнения сторожевого условия ветвления.

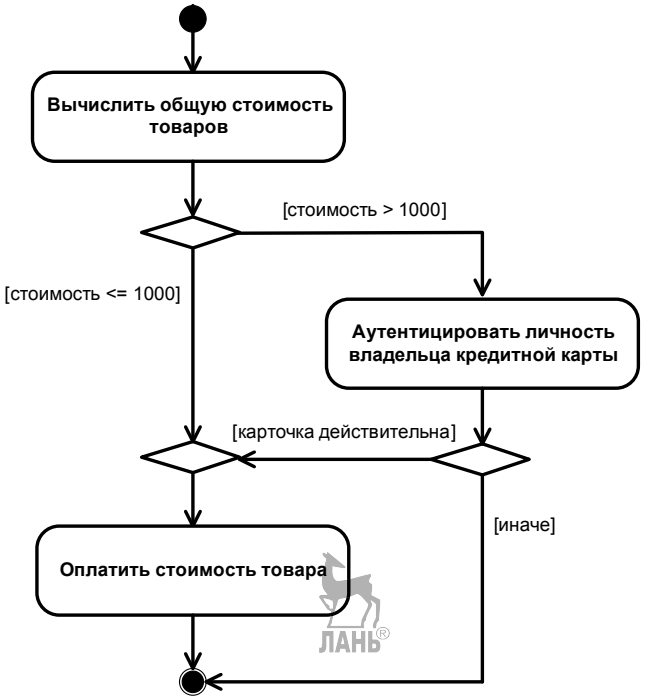


Рис. 2.29
Различные варианты ветвлений на диаграмме деятельности

Один из наиболее значимых недостатков обычных блок-схем или структурных схем алгоритмов связан с проблемой изображения параллельных ветвей отдельных вычислений. Поскольку распараллеливание вычислений существенно повышает общее быстродействие программных систем, необходимы графические примитивы для представления параллельных процессов. В языке UML для этой цели используется специальный символ для разделения и слияния параллельных вычислений или потоков управления. Таким символом является прямая черточка. Как правило, такая черточка изображается отрезком горизонтальной линии, толщина которой несколько шире основных сплошных линий диаграммы деятельности. При этом разделение (concurrent fork) имеет один входящий переход и несколько выходящих (рис. 2.30, *а*). Слияние (concurrent join), наоборот, имеет несколько входящих переходов и один выходящий (рис. 2.30, *б*).

а, разделение *б, слияние*

Рис. 2.30
Графическое изображение разделения и слияния параллельных потоков управления

Для иллюстрации особенностей параллельных процессов выполнения действий рассмотрим пример с употреблением напитка. Достоинство этого примера состоит в том, что он практически не требует никаких дополнительных пояснений в силу своей очевидности (рис. 2.31).

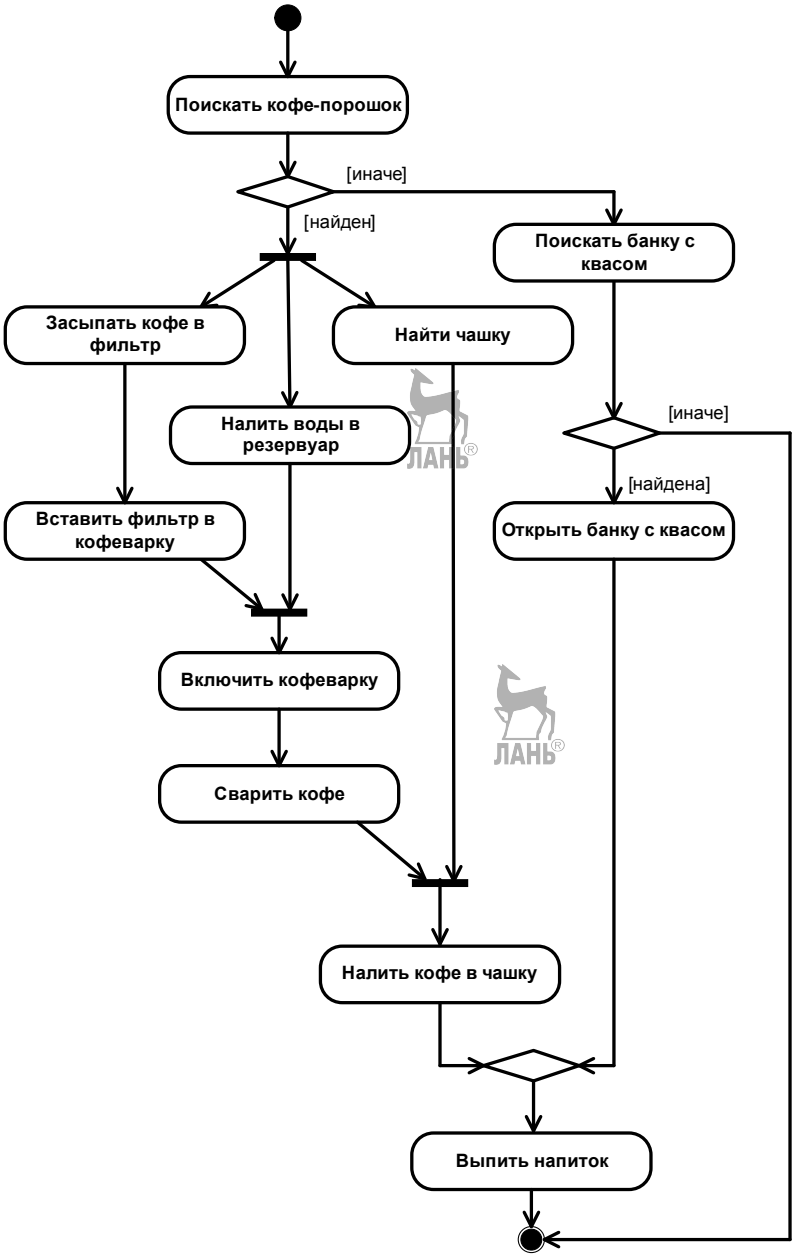


Рис. 2.31
Диаграмма деятельности для примера с приготовлением напитка

Наличие параллельности проявляется в том, что мы можем заняться поисками чашки во время приготовления кофе. Впрочем, поскольку выбор конкретных напитков остается за читателем, разработка соответствующей диаграммы деятельности может быть предложена в качестве упражнения.

Контрольные вопросы

1. Какие графические формы используются для обозначения переходов на диаграмме деятельности?

2.3.3. Дорожки

Диаграммы деятельности могут быть использованы не только для спецификации алгоритмов вычислений или потоков управления в программных системах. Не менее важная область их применения связана с моделированием бизнес-процессов. Действительно, деятельность любой компании (фирмы) также представляет собой не что иное, как совокупность отдельных действий, направленных на достижение требуемого результата. Однако применительно к бизнес-процессам желательно выполнение каждого действия ассоциировать с конкретным подразделением компании. В этом случае подразделение несет ответственность за реализацию отдельных действий, а сам бизнес-процесс представляется в виде переходов действий из одного подразделения к другому.

Для моделирования этих особенностей в языке UML используется специальная конструкция, получившее название дорожки (swimlanes). Имеется в виду визуальная аналогия с плавательными дорожками в бассейне, если смотреть на соответствующую диаграмму. При этом все состояния действия на диаграмме деятельности делятся на отдельные группы, которые отделяются друг от друга вертикальными линиями. Две соседние линии и образуют дорожку, а группа состояний между этими линиями выполняется отдельным подразделением (отделом, группой, отделением, филиалом) компании (рис. 2.32).

Названия подразделений явно указываются в верхней части дорожки. Пересекать линию дорожки могут только переходы, которые в этом случае обозначают выход или вход потока управления в соответствующее подразделение компании. Порядок следования дорожек не несет какой-либо семантической информации и определяется соображениями удобства.

В качестве примера рассмотрим фрагмент диаграммы деятельности торговой компании, обслуживающей клиентов по телефону. Подразделениями компании являются отдел приема и оформления заказов, отдел продаж и склад.

Этим подразделениям будут соответствовать три дорожки на диаграмме деятельности, каждая из которых специфицирует зону ответственности подразделения. В данном случае диаграмма деятельности заключает в себе не только информацию о последовательности выполнения рабочих действий, но и о том, какое из подразделений торговой компании должно выполнять то или иное действие (рис. 2.32).

Из указанной диаграммы деятельности сразу видно, что после принятия заказа от клиента отделом приема и оформления заказов осуществляется распараллеливание деятельности на два потока (переход-разделение). Первый из них остается в этом же отделе и связан с получением оплаты от клиента за заказанный товар. Второй инициирует выполнение действия по подбору товара в отделе продаж (модель товара, размеры, цвет, год выпуска и пр.). По окончании этой работы инициируется действие по отпуску товара со склада. Однако подготовка товара к отправке в торговом отделе начинается только после того, как будет получена оплата за товар от клиента и товар будет отпущен со склада (переход-соединение). Только после этого товар отправляется клиенту, переходя в его собственность. Нетрудно заметить, что на рисунке 2.32 рассмотрен идеальный случай безусловного наличия товара на складе.

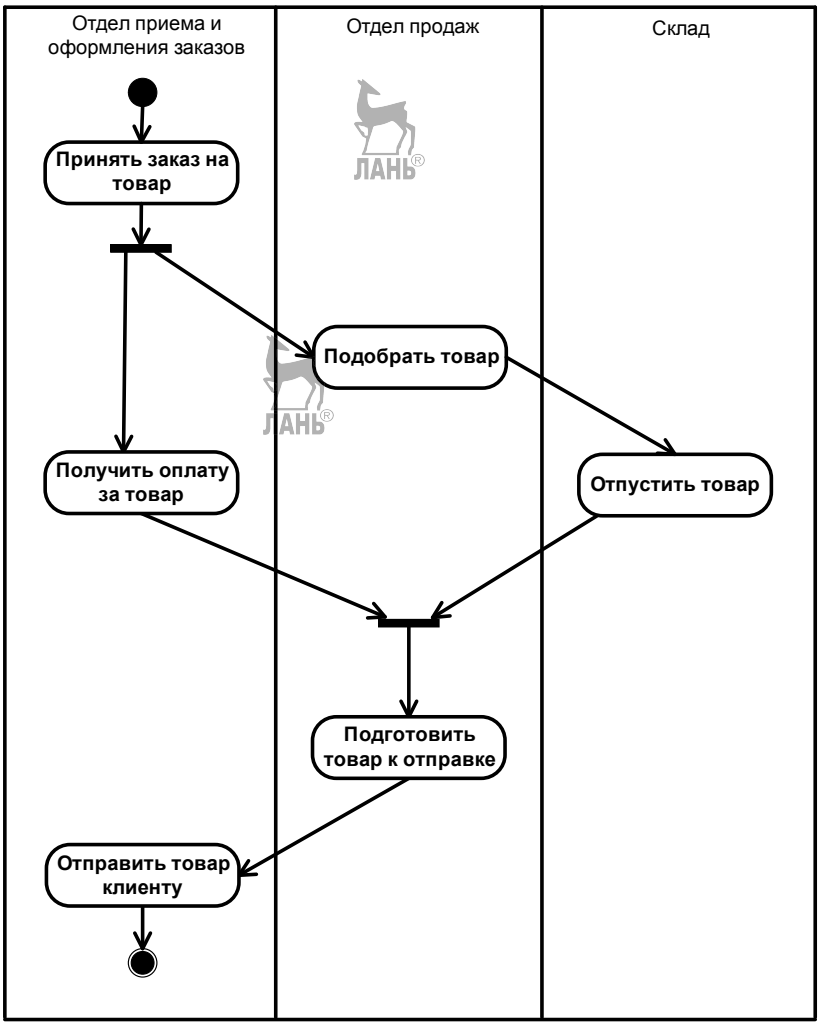


Рис. 2.32
Фрагмент диаграммы деятельности для торговой компании

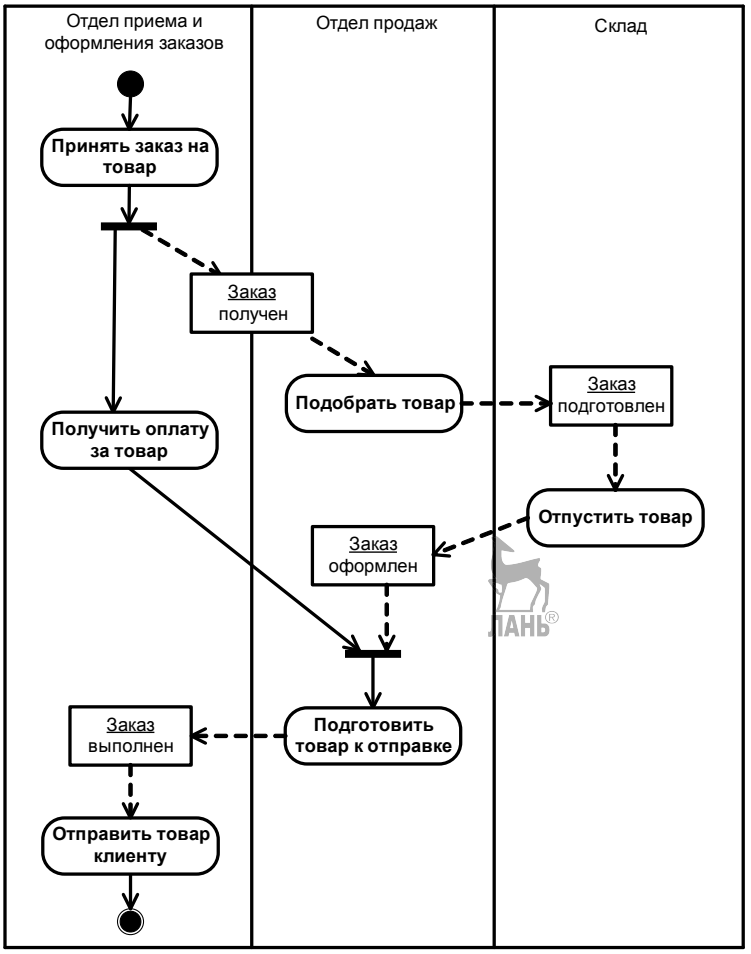


Рис. 2.33
Фрагмент диаграммы деятельности торговой компании с объектом-заказом

Контрольные вопросы

1. Какие графические формы используются для моделирования потоков управления на диаграмме деятельности?

2.3.4. Объекты

В общем случае действия на диаграмме деятельности выполняются над теми или иными объектами. Эти объекты либо инициируют выполнение действий, либо определяют некоторый результат этих действий. При этом действия специфицируют вызовы, которые передаются от одного объекта графа деятельности к другому. Поскольку в таком ракурсе объекты играют определенную роль в понимании процесса деятельности, иногда возникает необходимость явно указать их на диаграмме деятельности. На диаграмме деятельности один и тот же объект может быть изображен несколько раз, что не должно вносить неопределенность в спецификацию состояний действия.

Для графического представления объектов используется прямоугольник с подчеркнутым именем объекта. Далее, после имени, может указываться характеристика состояния объекта в прямых скобках. Такие прямоугольники объектов присоединяются к состояниям действия отношением зависимости пунктирной линией со стрелкой. Соответствующая зависимость определяет состояние конкретного объекта после выполнения предшествующего действия.

На диаграмме деятельности с дорожками расположение объекта может иметь некоторый дополнительный смысл. А именно, если объект расположен на границе двух дорожек, то это может означать, что переход к следующему состоянию действия в соседней дорожке ассоциирован с готовностью некоторого документа (объект в некотором состоянии). Если же объект целиком расположен внутри дорожки, то и состояние этого объекта целиком определяется действиями данной дорожки.

Возвращаясь к предыдущему примеру с торговой компанией, можно заметить, что центральным объектом процесса продажи является заказ или вернее состояние его выполнения. Вначале до звонка от клиента заказ как объект отсутствует и возникает лишь после такого звонка. Однако этот заказ еще не заполнен до конца, поскольку требуется еще подобрать конкретный товар в отделе продаж. После его подготовки он передается на склад, где вместе с отпуском товара заказ окончательно дооформляется. Наконец, после получения подтверждения об оплате товара эта информация заносится в заказ, и он считается выполненным и закрытым. Данная информация может быть представлена графически в виде модифицированного варианта диаграммы деятельности этой же торговой компании (рис. 2.33).

Контрольные вопросы

1. Какие графические формы используются для моделирования объектов на диаграмме деятельности?

2.3.5. Рекомендации по построению диаграмм деятельности

Диаграммы деятельности играют важную роль в понимании процессов реализации алгоритмов выполнения операций классов и потоков управления в моделируемой системе. Используемые для этой цели традиционные блок-схемы алгоритмов обладают серьезными ограничениями в представлении параллельных процессов и их синхронизации. Применение дорожек и объектов открывает дополнительные возможности для наглядного представления бизнес-процессов, позволяя специфицировать деятельность подразделений компаний и фирм.

С одной стороны, на начальных этапах проектирования, когда детали реализации деятельностей в проектируемой системе неизвестны, построение диаграммы деятельности начинают с выделения поддеятельностей, которые в совокупности образуют деятельность подсистем. В последующем, по мере разработки диаграмм классов и состояний, эти поддеятельности уточняются в виде отдельных вложенных диаграмм деятельности компонентов подсистем, какими выступают классы и объекты.

С другой стороны, отдельные участки рабочего процесса в существующей системе могут быть хорошо отлаженными, и у разработчиков может возникнуть желание сохранить этот механизм выполнения действий в проектируемой системе. Тогда строится диаграмма деятельности для этих участков, отражающая конкретные особенности выполнения действий с использованием дорожек и объектов. В последующем такая диаграмма вкладывается в более общие диаграммы деятельности для подсистемы и системы в целом, сохраняя свой уровень детализации.

Таким образом, процесс объектно-ориентированного анализа и проектирования сложных систем представляется как последовательность итераций нисходящей и восходящей разработки отдельных диаграмм, включая и диаграмму деятельности. Доминирование того или иного из направлений разработки определяется особенностями конкретного проекта и его новизной.

В случае типового проекта большинство деталей реализации действий могут быть известны заранее на основе анализа существующих систем или предшествующего опыта разработки систем-прототипов. Для этой ситуации доминирующим будет восходящий процесс разработки (зачем изобретать велосипед заново?!). Использование типовых решений может существенно сократить время разработки и избежать возможных ошибок при реализации проекта.

При разработке проекта новой системы, процесс функционирования которой основан на новых технологических решениях, ситуация представляется более сложной. А именно, до начала работы над проектом могут быть неизвестны не только детали реализации отдельных деятельностей, но и само содержание этих деятельностей становится предметом разработки. В данном случае доминирующим будет нисходящий процесс разработки от более общих схем к уточняющим их диаграммам. При этом достижение такого уровня детализации всех диаграмм, который достаточен для понимания особенностей реализации всех действий и деятельностей, может служить признаком завершения отдельных этапов работы над проектом.

В заключение следует заметить, что диаграмма деятельности так же, как и другие виды канонических диаграмм, не содержит средств выбора оптимальных решений. При разработке сложных проектов проблема выбора оптимальных решений становится весьма актуальной. Рациональное расходование средств, затраченных на разработку и эксплуатацию системы, повышение ее производительности и надежности зачастую определяют конечный результат всего проекта. В такой ситуации можно рекомендовать использование дополнительных средств и методов, ориентированных на аналитико-имитационное исследование моделей системы на этапе разработки ее проекта.

В частности, при построении диаграмм деятельности сложных систем могут быть успешно использованы различные классы сетей Петри (классические, логико-алгебраические, стохастические, нечеткие и др.) и нейронных сетей. Применение этих формализмов позволяет не только получить оптимальную структуру поведения системы на ее модели, но и специфицировать целый ряд дополнительных характеристик системы, которые не могут быть представлены на диаграмме деятельности и других диаграммах UML.

Контрольные вопросы

1. Выделите в тексте основные рекомендации по построению диаграмм деятельности.

2.4. Диаграмма последовательности

Как было отмечено в главе 1 книги, одной из характерных особенностей систем различной природы и назначения является взаимодействие между собой отдельных элементов, из которых образованы эти системы. Речь идет о том, что различные составные элементы систем не существуют изолированно, а оказывают определенное влияние друг на друга, что и отличает систему как целостное образование от простой совокупности элементов.

В языке UML взаимодействие элементов рассматривается в информационном аспекте их коммуникации, т. е. взаимодействующие объекты обмениваются между собой некоторой информацией. При этом информация принимает форму законченных сообщений. Другими словами, хотя сообщение и имеет информационное содержание, оно приобретает дополнительное свойство оказывать направленное влияние на своего получателя. Это полностью согласуется с принципами объектно-ориентированного анализа и проектирования, когда любые виды информационного взаимодействия между элементами системы должны быть сведены к отправке и приему сообщений между ними.

Для моделирования взаимодействия объектов в языке UML используются соответствующие диаграммы взаимодействия. Говоря об этих диаграммах, имеют в виду два аспекта взаимодействия. Во-первых, взаимодействия объектов можно рассматривать во времени, и тогда для представления временных особенностей передачи и приема сообщений между объектами используется диаграмма последовательности. Этот вид канонических диаграмм является предметом изучения настоящей главы.

Ранее, при изучении диаграмм взаимодействия, было отмечено одно немаловажное обстоятельство. Хотя рассмотренные диаграммы и используются для спецификации динамики поведения систем, время в явном виде в них не присутствует. Однако временной аспект поведения может иметь существенное значение при моделировании синхронных процессов, описывающих взаимодействия объектов. Именно для этой цели в языке UML используются диаграммы последовательности (sequence diagram).

2.4.1. Объекты

На диаграмме последовательности изображаются исключительно те объекты, которые непосредственно участвуют во взаимодействии и не показываются возможные статические ассоциации с другими объектами. Для диаграммы последовательности ключевым моментом является именно динамика взаимодействия объектов во времени. Графически каждый объект изображается прямоугольником и располагается в верхней части своей линии жизни (рис. 2.34). Внутри прямоугольника записываются имя объекта и имя класса, разделенные двоеточием.

Крайним слева на диаграмме изображается объект, который является инициатором взаимодействия (объект 1 на рис. 2.34). Правее изображается другой объект, который непосредственно взаимодействует с первым. Таким образом, все объекты на диаграмме последовательности образуют некоторый порядок, определяемый степенью активности этих объектов при взаимодействии друг с другом. При этом масштаб на оси времени не указывается, поскольку диаграмма последовательности моделирует лишь временную упорядоченность взаимодействий типа «раньше-позже».

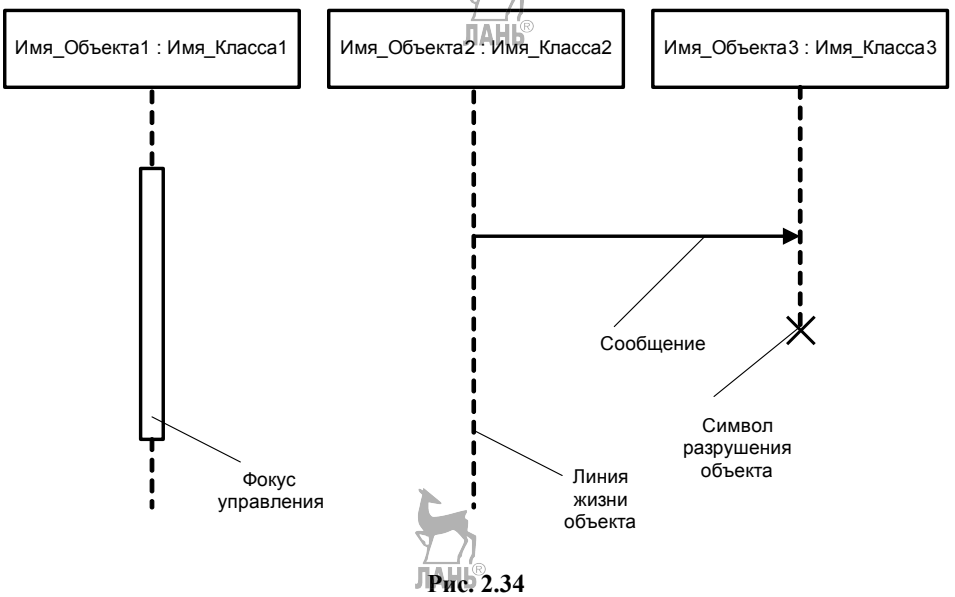


Рис. 2.34

Различные графические примитивы диаграммы последовательности

Линия жизни объекта. Линия жизни объекта (object lifeline) изображается пунктирной вертикальной линией, ассоциированной с единственным объектом на диаграмме последовательности. Линия жизни служит для обозначения периода времени, в течение которого объект существует в системе и, следовательно, может потенциально участвовать во всех ее взаимодействиях. Если объект существует в системе постоянно, то и его линия жизни должна продолжаться по всей плоскости диаграммы последовательности от самой верхней ее части до самой нижней (объекты 1 и 2 на рис. 2.34).

Отдельные объекты, сыграв свою роль в системе, могут быть уничтожены (разрушены), чтобы освободить занимаемые ими ресурсы. Для таких объектов линия жизни обрывается в момент его уничтожения. Для обозначения момента уничтожения объекта в языке UML используется специальный символ: «крестик» (объект 3 на рис. 2.34). Ниже этого символа пунктирная линия не изображается, поскольку соответствующего объекта в системе уже нет, и этот объект должен быть исключен из всех последующих взаимодействий.

Вовсе не обязательно создавать все объекты в начальный момент времени. Отдельные объекты в системе могут создаваться по мере необходимости, существенно экономя ресурсы системы и повышая ее производительность.

В этом случае прямоугольник такого объекта изображается не в верхней части диаграммы последовательности, а в той ее части, которая соответствует моменту создания объекта (объект 6 на рис. 2.35). При этом прямоугольник объекта вертикально располагается в том месте диаграммы, которое по оси времени совпадает с моментом его возникновения в системе. Очевидно, объект обязательно создается со своей линией жизни и, возможно, с фокусом управления.

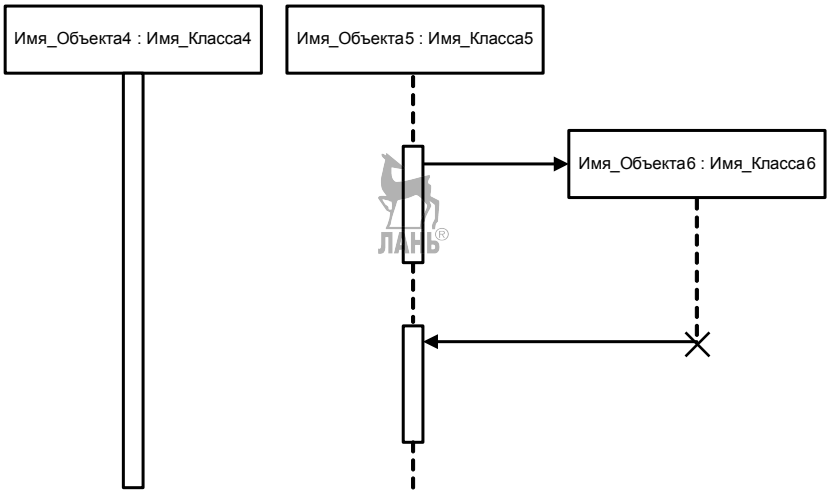


Рис. 2.35
Графическое изображение различных вариантов линий жизни и фокусов управления объектов

Фокус управления. Как было рассмотрено в главе 1, в процессе функционирования объектно-ориентированных систем одни объекты могут находиться постоянно в активном состоянии (актер), воздействуя на другие объекты, или в состоянии пассивного ожидания сообщений от других объектов (сервер). Чтобы явно выделить подобную активность объектов, в языке UML применяется специальное понятие, получившее название фокуса управления (focus of control). Фокус управления изображается в форме вытянутого узкого прямоугольника (см. объект 1 на рис. 2.34), верхняя сторона которого обозначает начало получения фокуса управления объекта (начало активности), а ее нижняя сторона — окончание фокуса управления (окончание активности). Этот прямоугольник располагается ниже обозначения соответствующего объекта и может заменять его линию жизни (объект 4 на рис. 2.35), если на всем ее протяжении он является активным.

С другой стороны, периоды активности объекта могут чередоваться с периодами его пассивности или ожидания (агент). В этом случае у такого объекта имеются несколько фокусов управления (объект 5 на рис. 2.35). Важно понимать, что получить фокус управления может только существующий объект, у которого в этот момент имеется линия жизни. Если же некоторый объект был уничтожен, то вновь возникнуть в системе он уже не может. Вместо него лишь может быть создан другой экземпляр этого же класса, который, строго говоря, будет являться другим объектом.

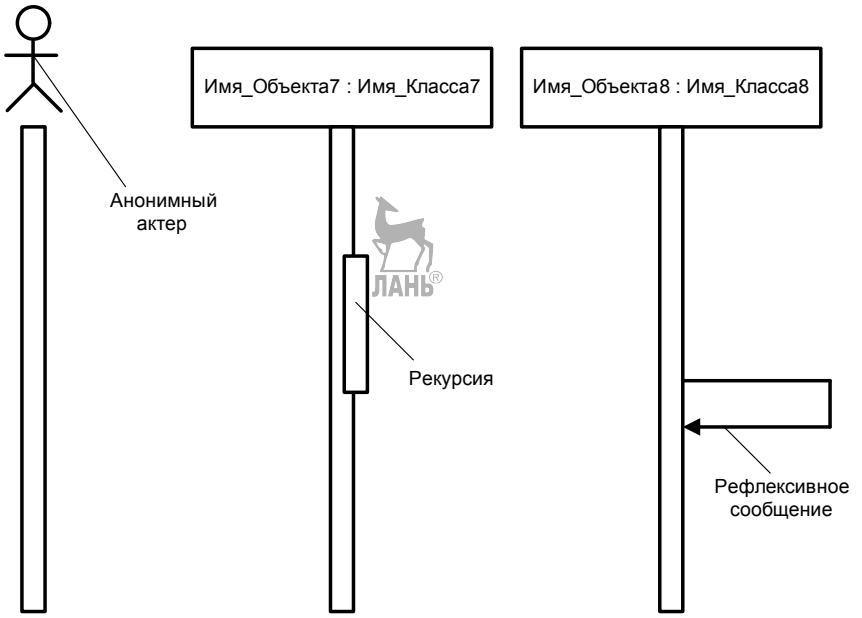


Рис. 2.36
Графическое изображение актера, рекурсии и рефлексивного сообщения на диаграмме последовательности

В отдельных случаях инициатором взаимодействия в системе может быть внешний пользователь. В этом случае внешний пользователь изображается на диаграмме последовательности самым первым объектом слева со своим фокусом управления (рис. 2.36). Чаще всего внешний пользователь является актером, и его фокус управления будут существовать в системе постоянно, отмечая характерную для пользователя активность в инициировании взаимодействий с системой. При этом сам актер может иметь собственное имя либо оставаться анонимным.

Иногда некоторый объект может инициировать рекурсивное взаимодействие с самим собой. Речь идет о том, что наличие во многих языках программирования специальных средств построения рекурсивных процедур требует визуализации соответствующих понятий в форме графических примитивов. На диаграмме последовательности рекурсия обозначается небольшим прямоугольником, присоединенным к правой стороне фокуса управления того объекта, для которого изображается это рекурсивное взаимодействие (объект 7 на рис. 2.36).

Контрольные вопросы

1. Какие графические средства используются для моделирования объектов диаграммы последовательности?
2. Что представляет собой линия жизни объекта?
3. Какое назначение понятия «фокус управления» диаграммы последовательности?

2.4.2. Сообщения

Как было отмечено выше, цель взаимодействия в контексте языка UML заключается в том, чтобы специфицировать коммуникацию между множеством взаимодействующих объектов. Каждое взаимодействие описывается совокупностью сообщений, которыми участвующие в нем объекты обмениваются между собой. В этом смысле сообщение (message) представляет собой законченный фрагмент информации, который отправляется одним объектом другому. При этом прием сообщения инициирует выполнение определенных действий, направленных на решение отдельной задачи тем объектом, которому это сообщение отправлено.

Таким образом, сообщения не только передают некоторую информацию, но и требуют или предполагают от принимающего объекта выполнения ожидаемых действий. Сообщения могут инициировать выполнение операций объектом соответствующего класса, а параметры этих операций передаются вместе с сообщением. На диаграмме последовательности все сообщения упорядочены по времени своего возникновения в моделируемой системе.

В таком контексте каждое сообщение имеет направление от объекта, который инициирует и отправляет сообщение, к объекту, который его получает. Часто отправителя сообщения называют клиентом, а получателя — сервером. При этом сообщение от клиента имеет форму запроса некоторого сервиса, а реакция сервера на запрос после получения сообщения может быть связана с выполнением определенных действий или передачи клиенту необходимой информации тоже в форме сообщения.

В языке UML могут встречаться несколько разновидностей сообщений, каждое из которых имеет свое графическое изображение (рис. 2.37).

Первая разновидность сообщения (рис. 2.37, *а*) является наиболее распространенной и используется для вызова процедур, выполнения операций или обозначения отдельных вложенных потоков управления. Начало этой стрелки всегда соприкасается с фокусом управления или линией жизни того объекта-клиента, который инициирует это сообщение. Конец стрелки соприкасается с линией жизни того объекта, который принимает это сообщение и выполняет в ответ определенные действия. При этом принимающий объект зачастую получает и фокус управления, становясь активным.

Вторая разновидность сообщения (рис. 2.37, *б*) используется для обозначения простого (не вложенного) потока управления. Каждая такая стрелка указывает на прогресс одного шага потока. При этом соответствующие сообщения обычно являются асинхронными, т. е. могут возникать в произвольные моменты времени. Передача такого сообщения обычно сопровождается получением фокуса управления объектом, его принявшим.

Третья разновидность (рис. 2.37, *в*) явно обозначает асинхронное сообщение между двумя объектами в некоторой процедурной последовательности. Примером такого сообщения может служить прерывание операции при возникновении исключительной ситуации. В этом случае информация о такой ситуации передается вызывающему объекту для продолжения процесса дальнейшего взаимодействия.

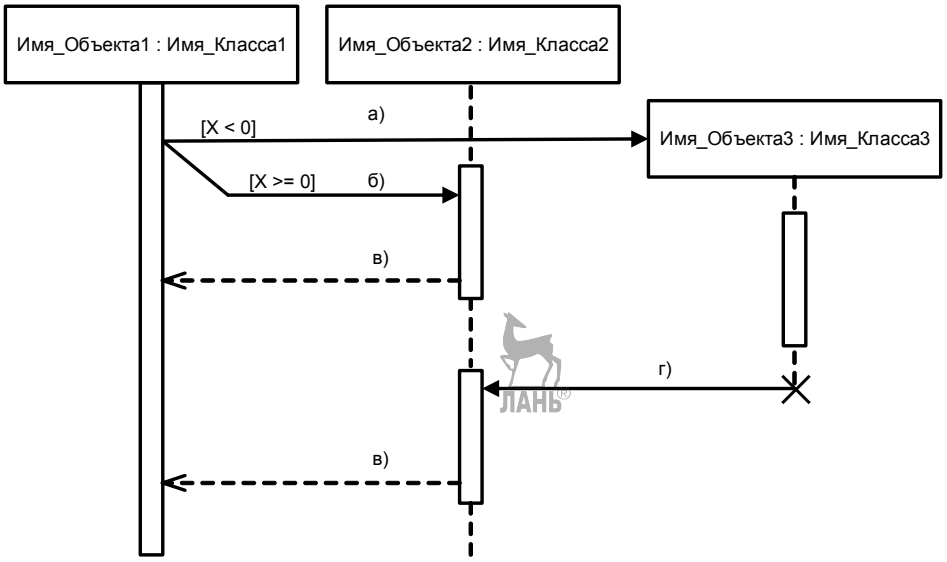


Рис. 2.37
Графическое изображение различных видов сообщений между объектами на диаграмме последовательности

Наконец, последняя разновидность сообщения (рис. 2.37, *г*) используется для возврата из вызова процедуры. Примером может служить простое сообщение о завершении некоторых вычислений без предоставления результата расчетов объекту-клиенту. В процедурных потоках управления эта стрелка может быть опущена, поскольку ее наличие неявно предполагается в конце активизации объекта. В то же время считается, что каждый вызов процедуры имеет свою пару — возврат вызова. Для непроцедурных потоков управления, включая параллельные и асинхронные сообщения, стрелка возврата должна указываться явным образом.

Обычно сообщения изображаются горизонтальными стрелками, соединяющими линии жизни или фокусы управления двух объектов на диаграмме последовательности. При этом неявно предполагается, что время передачи сообщения достаточно мало по сравнению с процессами выполнения действий объектами. Считается также, что за время передачи сообщения с соответствующими объектами не может произойти никаких событий. Другими словами, состояния объектов остаются без изменения. Если же это предположение не может быть признано справедливым, то стрелка сообщения изображается под некоторым наклоном так, чтобы конец стрелки располагался ниже ее начала.

В отдельных случаях объект может посылать сообщения самому себе, инициируя так называемые рефлексивные сообщения (объект 8 на рис. 2.36). Такие сообщения изображаются прямоугольником со стрелкой, начало и конец которой совпадают. Подобные ситуации возникают, например, при обработке нажатий на клавиши клавиатуры при вводе текста в редактируемый документ, при наборе цифр номера телефона абонента.

Таким образом, в языке UML каждое сообщение ассоциируется с некоторым действием, которое должно быть выполнено принявшим его объектом.

При этом действие может иметь некоторые аргументы или параметры, в зависимости от конкретных значений которых может быть получен различный результат. Соответствующие параметры будет иметь и вызывающее это действие сообщение. Более того, значения параметров отдельных сообщений могут содержать условные выражения, образуя ветвление или альтернативные пути основного потока управления.

Ветвление потока управления. Для изображения ветвления рисуются две или более стрелки, выходящие из одной точки фокуса управления объекта (фокус управления объекта 1 на рис. 2.36). При этом соответствующие условия должны быть явно указаны рядом с каждой из стрелок в форме условия передачи управления. Как нетрудно представить, если условие записано в форме булевского выражения, то ветвление будет содержать только две ветви. В любом случае условия должны взаимно исключать одновременную передачу альтернативных сообщений.

С помощью ветвления можно изобразить и более сложную логику взаимодействия объектов между собой. Если условий более двух, то для каждого из них необходимо предусмотреть ситуацию единственного выполнения.

Стереотипы сообщений. В языке UML предусмотрены некоторые стандартные действия, выполняемые в ответ на получение соответствующего сообщения. Они могут быть явно указаны на диаграмме последовательности в форме стереотипа рядом с сообщением, к которому они относятся. Используются следующие обозначения для моделирования действий:

— **call** (вызвать) — сообщение, требующее вызова операции или процедуры принимающего объекта. Если сообщение с этим стереотипом рефлексивное, то оно инициирует локальный вызов операции у самого пославшего это сообщение объекта;

— **return** (возвратить) — сообщение, возвращающее значение выполненной операции или процедуры вызвавшему ее объекту. Значение результата может инициировать ветвление потока управления;

— **create** (создать) — сообщение, требующее создания другого объекта для выполнения определенных действий. Созданный объект может получить фокус управления, а может и не получить его;

— **destroy** (уничтожить) — сообщение с явным требованием уничтожить соответствующий объект. Посылается в том случае, когда необходимо прекратить нежелательные действия со стороны существующего в системе объекта либо когда объект больше не нужен и должен освободить задействованные им системные ресурсы;

— **send** (послать) — обозначает посылку другому объекту некоторого сигнала, который асинхронно инициируется одним объектом и принимается (перехватывается) другим. Отличие сигнала от сообщения заключается в том, что сигнал должен быть явно описан в том классе, объект которого инициирует его передачу.

Ниже представлена диаграмма последовательности для рассмотренного выше случая ветвления, дополненная стереотипными значениями (рис. 2.38).

Кроме стереотипов, сообщения могут иметь собственное обозначение операции, вызов которой они инициируют у принимающего объекта. В этом случае рядом со стрелкой записывается имя операции с круглыми скобками, в которых могут

указываться параметры или аргументы соответствующей операции. Если параметры отсутствуют, то скобки все равно должны присутствовать после имени операции. Примерами таких операций могут служить следующие: «выдать клиенту наличными сумму (п)», «установить соединение между абонентами (*a*, *b*)», «сделать вводимый текст невидимым ()», «подать звуковой сигнал тревоги ()».

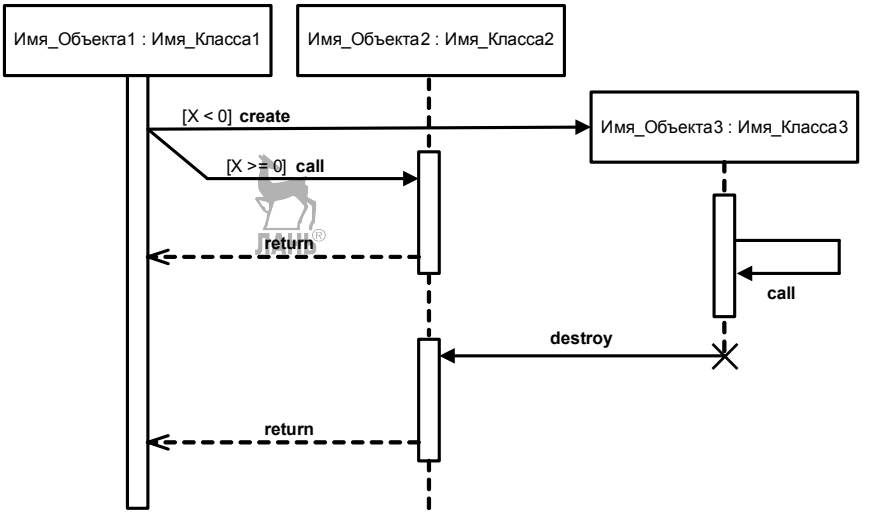


Рис. 2.38
Диаграмма последовательности со стереотипными значениями сообщений

Временные ограничения на диаграммах последовательности. В отдельных случаях выполнение тех или иных действий на диаграмме последовательности может потребовать явной спецификации временных ограничений, накладываемых на сам интервал выполнения операций или передачу сообщений. В языке UML для записи временных ограничений используются фигурные скобки. Временные ограничения могут относиться как к выполнению определенных действий объектами, так и к самим сообщениям, явно специфицируя условия их передачи или приема. Важно понимать, что в отличие от условий ветвления, которые должны выполняться альтернативно, временные ограничения имеют обязательный или директивный характер для ассоциированных с ними объектов.

Временные ограничения могут записываться рядом с началом стрелки соответствующего сообщения. Но наиболее часто они записываются слева от этой стрелки на одном уровне с ней. Если временная характеристика относится к конкретному объекту, то имя этого объекта записывается перед именем характеристики и отделяется от нее точкой.

Примерами таких ограничений на диаграмме последовательности могут служить ситуации, когда необходимо явно специфицировать время, в течение которого допускается передача сообщения от клиента к серверу или обработка запроса клиента сервером:

— {время_приема_сообщения – время_отправки_сообщения < 1 с};
— {время_ожидания_ответа < 5 с};

— {время_передачи_пакета < 10 с};
— {объект_1. время_подачи_сигнала_тревоги > 30 с}.

В первом из рассмотренных случаев знак «–» во временном ограничении обозначает арифметическую операцию вычитания (минус). Другие знаки являются обычными знаками сравнения величин. В последнем случае перед временной характеристикой указано имя объекта, к которому она относится.

Комментарии или примечания. Комментарии или примечания уже рассматривались ранее при изучении других видов диаграмм. Они могут включаться и в диаграммы последовательности, ассоциируясь с отдельными объектами или сообщениями. При этом используется стандартное обозначение для комментария — прямоугольник с «заломленным» правым верхним углом. Внутри этого прямоугольника записывается текст комментария на естественном языке.

Контрольные вопросы

1. Какие средства используются для спецификации коммуникации между множеством взаимодействующих объектов?
2. С какими разновидностями сообщений мы познакомились в данной главе?
3. Какие существуют стереотипы сообщений?
4. Как на диаграммах последовательности задаются временные ограничения?

2.4.3. Пример построения диаграммы последовательности

В качестве примера рассмотрим построение диаграммы последовательности для моделирования процесса телефонного разговора с использованием обычной телефонной сети. Объектами в этом примере являются: два абонента *A* и *B*, два телефонных аппарата *C* и *D*, коммутатор и сам разговор как объект моделирования. При этом как коммутатор, так и разговор в нашей задаче являются анонимными объектами.

На первом этапе располагаем выбранные объекты на предполагаемой диаграмме (рис. 2.39). Вызывающего абонента (*A*) мы будем рассматривать как актера, а второго (*B*) — как агента. Поэтому первый получает фокус управления сразу после своего появления в системе, а второй — во время получения вызова. Коммутатор также имеет постоянную активность, что изображается его фокусом управления. Разговор как объект появляется только после установки соединения и уничтожается с его прекращением. Поэтому он будет изображен позже на этой же диаграмме последовательности. Дополненный фрагмент диаграммы последовательности изображен на рисунке 2.40.

Процесс взаимодействия в этой системе начинается с поднятия трубки телефонного аппарата абонентом *A*. Тем самым он посылает сообщение телефонному аппарату *C*, которое переводит этот аппарат в активное состояние и вызывает действие — подачу тонового сигнала в телефонную трубку для первого абонента. Следующее действие также инициируется абонентом *A*: набор цифр телефонного номера. Это представлено в форме итеративного сообщения со знаком «*» слева от его имени.

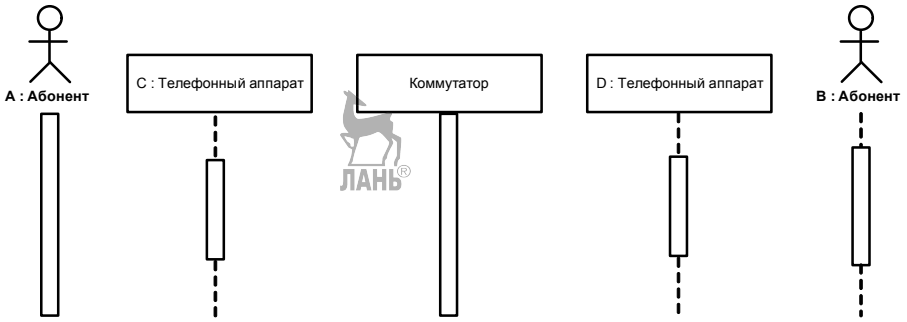


Рис. 2.39
Начальный фрагмент диаграммы последовательности
для моделирования телефонного разговора

Заметим, что поднятие телефонной трубки и набор цифр номера являются физическими действиями и поэтому изображаются в форме простых асинхронных сообщений. После набора цифр номера абонента аппарат *C* вызывает рекурсивную процедуру посылки коммутационных импульсов на коммутатор. Последний инициирует создание нового объекта в моделируемой системе — телефонного разговора (рис. 2.40).

После создания анонимный объект «разговор» сразу получает фокус активности и посылает сообщение телефонному аппарату *D* на выполнение действия — звонка вызова. При этом второй абонент снимает трубку (асинхронное сообщение), тем самым устанавливается прямое соединение между абонентами *A* и *B*. После того как абоненты опустят трубки, разговор заканчивается. Тем самым объект «разговор» уничтожается.

Окончательный вариант диаграммы последовательности может содержать некоторые временные ограничения и комментарии (рис. 2.41). Назначение отдельных сообщений соответствуют рассмотренным действиям.

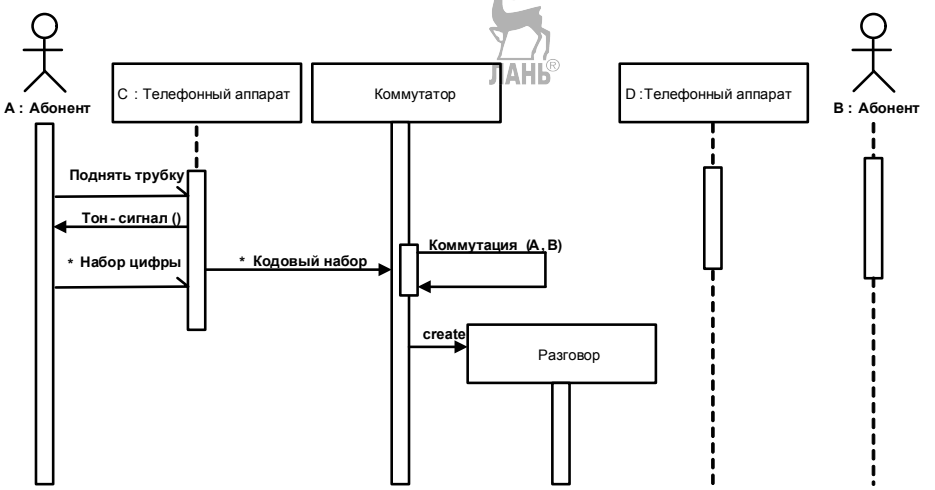


Рис. 2.40
Дополненный фрагмент диаграммы последовательности
для моделирования телефонного разговора

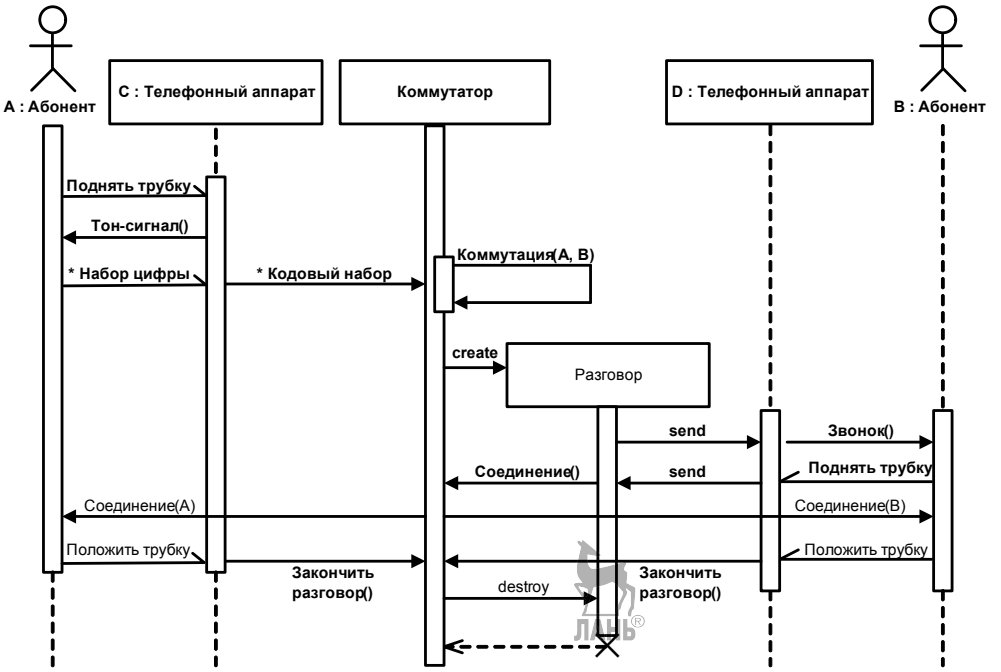


Рис. 2.41
Окончательный вариант диаграммы последовательности для моделирования телефонного разговора

Задание. Дополнить диаграмму последовательности для этого примера временными ограничениями предлагается выполнить самостоятельно в качестве упражнения.

Контрольные вопросы

1. Выделите в тексте настоящего параграфа основные рассуждения, которые предшествовали построению фрагментов диаграммы последовательности.

2.4.4. Заключительные рекомендации по построению диаграмм последовательности

Как уже отмечалось, построение диаграммы последовательности целесообразно начинать с выделения из всей совокупности тех и только тех классов, объекты которых участвуют в моделируемом взаимодействии. После этого все объекты наносятся на диаграмму с соблюдением некоторого порядка инициализации сообщений. Здесь необходимо установить, какие объекты будут существовать постоянно, а какие временно — только на период выполнения ими требуемых действий.

Когда объекты визуализированы, можно приступать к спецификации сообщений. При этом следует учитывать те роли, которые играют сообщения в системе. При необходимости уточнения этих ролей надо использовать их разновидности и стереотипы. Для уничтожения объектов, которые создаются на время выполнения своих действий, нужно предусмотреть явное сообщение.

Наиболее простые случаи ветвления процесса взаимодействия можно изобразить на одной диаграмме с использованием соответствующих графических примитивов. Однако следует помнить, что каждый альтернативный поток управления может существенно затруднить понимание построенной модели. Поэтому общим правилом является визуализация каждого потока управления на отдельной диаграмме последовательности. В этой ситуации такие отдельные диаграммы должны рассматриваться совместно как одна модель взаимодействия.

Дальнейшая детализация диаграммы последовательности связана с введением временных ограничений на выполнение отдельных действий в системе. Для простых асинхронных сообщений временные ограничения могут отсутствовать. Однако необходимость синхронизировать сложные потоки управления, как правило, требуют введение в модель таких ограничений. Общая их запись должна следовать семантике языка объектных ограничений, который рассмотрен в приложении.

В заключение следует отметить одно немаловажное обстоятельство, характерное для разработки всех канонических диаграмм. Хотя в языке UML определена графическая нотация для всех элементов канонических диаграмм, способы графического изображения отдельных инструментальных средств имеют свои специфические особенности. Применение того или иного инструментального CASE-средства накладывает определенные ограничения на визуализацию моделей программных систем. Речь идет о том, что некоторые элементы языка UML могут вообще отсутствовать в CASE-средствах. Выход из подобной ситуации может быть связан либо с выбором другого инструментария, поддерживающего последние версии языка UML, либо упрощении модели на основе ее типизации.

ЛАНЬ®

Контрольные вопросы

1. Выделите в тексте настоящего параграфа основные рекомендации по построению диаграммы последовательности.

2.5. Рекомендации по изображению диаграмм языка UML

При графическом изображении диаграмм следует придерживаться следующих основных рекомендаций:

— каждая диаграмма должна служить законченным представлением соответствующего фрагмента моделируемой предметной области. Речь идет о том, что в процессе разработки диаграммы необходимо учесть все сущности, важные с точки зрения контекста данной модели и диаграммы. Отсутствие тех или иных элементов на диаграмме служит признаком неполноты модели и может потребовать ее последующей доработки;

— все сущности на диаграмме модели должны быть одного концептуального уровня. Здесь имеется в виду согласованность не только имен одинаковых элементов, но и возможность вложения отдельных диаграмм друг в друга для достижения полноты представлений. В случае достаточно сложных моде-

лей систем желательно придерживаться стратегии последовательного уточнения или детализации отдельных диаграмм;

— вся информация о сущностях должна быть явно представлена на диаграммах. Речь идет о том, что, хотя в языке UML при отсутствии некоторых символов на диаграмме могут быть использованы их значения по умолчанию (например, в случае неявного указания видимости атрибутов и операций классов), необходимо стремиться к явному указанию свойств элементов диаграмм;

— диаграммы не должны содержать противоречивой информации. Противоречивость модели может служить причиной серьезнейших проблем при ее реализации и последующем использовании на практике. Например, наличие замкнутых путей при изображении отношений агрегации или композиции приводит к ошибкам в программном коде, который будет реализовывать соответствующие классы. Наличие элементов с одинаковыми именами и различными атрибутами свойств в одном пространстве имен также приводит к неоднозначной интерпретации и может служить источником проблем. Наличие в инструментальных CASE-средствах встроенной поддержки визуализации различных диаграмм языка UML позволяет в некоторой степени исключить ошибочное использование тех или иных графических символов, а также контролировать уникальность имен элементов диаграмм. Однако, поскольку вся ответственность за окончательный контроль непротиворечивости модели лежит на разработчике, необходимо помнить, что неформальный характер языка UML может служить источником потенциальных ошибок, которые в полном объеме вряд ли будут выявлены инструментальными средствами. Именно это обстоятельство требует от всех разработчиков глубокого знания нотации и семантики всех элементов языка UML;

— диаграммы не следует перегружать текстовой информацией. Принято считать, что визуализация модели является наиболее эффективной, если она содержит минимум пояснительного текста. Как правило, наличие больших фрагментов развернутого текста служит признаком недостаточной проработанности модели или ее неоднородности, когда в рамках одной модели представляется различная по характеру информация. Поскольку общая декомпозиция модели на отдельные типы диаграмм способна удовлетворить самые детальные представления разработчиков о системе, важно уметь правильно отображать те или иные сущности и аспекты моделирования в соответствующие элементы канонических диаграмм;

— каждая диаграмма должна быть самодостаточной для правильной интерпретации всех ее элементов и понимания семантики всех используемых графических символов. Любые пояснительные тексты, которые не являются собственными элементами диаграммы (например, комментариями), не должны приниматься во внимание разработчиками. В то же время отдельные достаточно общие фрагменты диаграмм могут уточняться или детализироваться на других диаграммах этого же типа, образуя вложенные или подчиненные диаграммы. Таким образом, модель системы на языке UML представляет собой пакет иерархически вложенных диаграмм, детализация которых должна быть доста-

точной для последующей генерации программного кода, реализующего проект соответствующей системы;

— количество типов диаграмм для конкретной модели приложения не является строго фиксированным. Речь идет о том, что для простых приложений нет необходимости строить все без исключения типы диаграмм. Некоторые из них могут просто отсутствовать в проекте системы, и этот факт не будет считаться ошибкой разработчика. Например, модель системы может не содержать диаграмму развертывания для приложения, выполняемого локально на компьютере пользователя. Важно понимать, что перечень диаграмм зависит от специфики конкретного проекта системы.

Контрольные вопросы

1. Выделите в тексте настоящего параграфа основные рекомендации по построению диаграмм языка UML.

Задачи

Для приведенных формулировок задач разработать диаграмму классов, диаграмму(ы) деятельности, диаграмму(ы) последовательности.

1. Программное обеспечение микропроцессора турникета метро

Требуется разработать модель программного обеспечения встроенного процессора турникета для метрополитена. Турникет служит для взимания оплаты за проезд. Оплата принимается с карточек метро, приобретаемых в кассах метрополитена. После каждого прохода через турникет количество оплаченных поездок на карточке уменьшается. Турникет должен пропускать пассажиров, поместивших в приемное устройство карту установленного образца с неисчерпанным кредитом. Турникет должен препятствовать походу пассажиров, которые не произвели оплату. Турникет также должен препятствовать проходу пассажиров в непредусмотренном направлении. Турникет имеет приемник карт, устройство для перекрывания доступа и три оптических датчика для определения направления прохода пассажира.

2. Система учета товаров

Требуется разработать модель системы поддержки заказа и учета товаров в бакалейном магазине. В лавке для каждого товара фиксируется место хранения (определенная полка), количество и поставщик этого товара. Система поддержки заказа и учета товаров должна обеспечивать возможность добавления/удаления нового товара, изменения информации об имеющемся товаре (например, при изменении количества товара и т. д.) и выдачи необходимой справочной информации (например, список товаров, количество которых необходимо пополнить).

3. Система автоматизации для библиотеки

Требуется разработать модель системы, автоматизирующей деятельность библиотеки. В библиотеке осуществляется регистрация всех читателей и ведутся каталоги поступивших в библиотеку книг, кроме того фиксируется

информация о том, какие книги у какого читателя находятся в данный момент. Система поддержки управления библиотекой должна обеспечивать возможность добавления/удаления читателей и, соответственно, книг в каталоги, регистрацию взятых и возвращенных читателем книг, а также выдавать полезную справочную информацию (например, о наличии в данный момент указанной книги).

4. Программное обеспечение микропроцессора информационного табло аэропорта

Требуется разработать модель программного обеспечения информационного табло, отображающего состояние авиарейсов. На табло отражается следующая информация о рейсе: номер рейса, пункт вылета, время прилета по расписанию, ожидаемое время прилета, статус (отложен, вылетел, прилетел). Система поддержки информационного табло должна обеспечивать добавление и удаление информации о рейсах, а также внесение изменений в состояние табло, если произошло некоторое событие (например, вылет какого-то рейса отложен на N минут, произошла посадка самолета указанного рейса и т. д.).

5. Программное обеспечение микропроцессора будильника

Требуется разработать модель программного обеспечения встроенного микропроцессора для будильника. Будильник постоянно отображает текущее время (часы, минуты). Управление будильником осуществляется следующими кнопками: кнопкой режима установки времени, кнопкой режима установки времени срабатывания, а также двумя отдельными кнопками для установки часов и минут. На будильнике имеется переключатель режима работы со следующими положениями: «ВЫКЛ», «ВКЛ», «РАДИО» и «ТАЙМЕР», а также кнопка сброса сигнала «СБРОС».

Для установки текущего времени нужно нажать кнопку режима установки и, при нажатой кнопке, нажимать кнопки установки часов и минут. При каждом нажатии кнопки, устанавливаемое значение увеличивается на 1 единицу (1 ч или 1 мин соответственно). При достижении максимального значения производится сброс. Для установки времени срабатывания будильника нужно нажать кнопку режима установки времени срабатывания и, держа кнопку нажатой, нажимать кнопки установки часов и минут. Когда переключатель режима работы находится в положении «ВКЛ», при достижении времени срабатывания происходит подача звукового сигнала в течение 1 минуты. Сигнал можно прервать, нажав кнопку «СБРОС». При этом сигнал должен быть возобновлен через 5 минут. При установке переключателя в положение «ВЫКЛ» звуковой сигнал не подается. Когда переключатель находится в положении «РАДИО», работает радиоприемник. При переводе переключателя в положение «ТАЙМЕР» включается радиоприемник на 30 минут, а затем часы переходят в состояние будильника (аналогично положению «ВКЛ»). При нажатии кнопки режима установки времени будильник должен отображать время срабатывания.

6. Программное обеспечение микропроцессора игрального автомата

Требуется разработать модель программного обеспечения встроенного микропроцессора для игрального автомата. Игральный автомат представляет собой прозрачный контейнер, наполненный призами. Внутри контейнера имеется захватное устройство на передвижной платформе. Платформа может перемещаться вдоль и поперек контейнера. Захват может опускаться на дно контейнера и поднимать захваченный груз вверх. Начальное положение платформы — в переднем правом углу контейнера. Захват поднят вверх. Передняя стенка контейнера представляет собой панель управления автоматом. Игра происходит следующим образом. Игрок опускает жетон в приемник. Затем игрок должен поместить манипулятор в определенную точку контейнера при помощи кнопок «Вперед» и «Стоп». Перемещение захвата вперед осуществляется при нажатой кнопке «Вперед». Движение останавливается при отпускании кнопки. Сразу после этого автоматически начинается движение захвата влево. Движение захвата останавливается при нажатии кнопки «Стоп». Сразу после этого захват открывается, опускается вниз, закрывается (возможно, захватывая при этом приз), поднимается наверх, возвращается вправо до конца, возвращается назад до конца и открывает захват. Если был захвачен приз, то он попадает в лоток выдачи. На этом игра заканчивается.

7. Программное обеспечение микропроцессора кодового замка

Требуется разработать модель программного обеспечения встроенного микропроцессора для кодового замка, регулирующего доступ в помещение. Кодовый замок состоит из панели с кнопками (цифры 0...9, кнопка «Вызов»), электромеханического замка, звонка и системы коммутации. Панель с кнопками устанавливается с наружной стороны двери, замок устанавливается с внутренней стороны двери, звонок устанавливается внутри охраняемого помещения. Нормальное положение замка — закрыто. Доступ в помещение осуществляется после набора кода доступа (4 цифры) (сигнал «открыть» замку). Закрытие двери осуществляется автоматически через определенный интервал времени. Кнопка «Вызов» используется для подачи звукового сигнала внутри помещения. Кнопка «Контроль» используется для смены кодов. Смена кода доступа осуществляется следующим образом. При открытой двери нужно набрать код контроля (4 цифры) и новый код доступа. Для смены кода контроля нужно при открытой двери и нажатой кнопке «Вызов» набрать код контроля, после чего — новый код контроля.

8. Виртуальная биржа труда

Требуется разработать модель программного обеспечения виртуальной биржи труда, предоставляющей вакансии и их размещение. Виртуальная биржа должна содержать сведения о работодателях: адрес, название организации, банковский счет, пароль, список предоставляемых вакансий, сумма на счету. Информация о вакансии содержит наименование работы, конечный срок подачи заявки на вакансию, примерная стоимость работы (максимум, который работодатель готов выплатить). Также должны храниться сведения о пользователях,

ищущих работу: ФИО, адрес, пароль, данные об участии в тех или иных проектах (биография), номер счета и сумма на счету. Виртуальная биржа должна обеспечивать перевод денег со счетов пользователей за 15% от переводимой суммы, пополнение счетов пользователей происходит бесплатно. Пользователь может подать заявку на участие в конкурсе на ту или иную вакансию, как только настанет конечная дата подачи заявок, работодатель выбирает из подавших заявки одного пользователя, которому и предоставляется вакансия. После выполнения контракта работнику переводится стоимость работы от работодателя.

3. ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ РАЗРАБОТКА ПРОГРАММНЫХ СИСТЕМ

Работу программистов следует оценивать не по их изобретательности и логике, а по полноте анализа каждой ситуации.

Алан. Дж. Перлис

Любая технология проектирования программ включает набор основных моделей, принципов и приемов отображения действительности и построения программ. Основное назначение технологии проектирования сложных программ состоит в предоставлении проектировщику удобных инструментов проработки принципов (архитектуры) программного изделия и выбор из них оптимальных принципов. Усилия, потраченные на проектирование программного изделия, затем с лихвой окупаются на последующих, более трудоемких этапах создания, сопровождения и развития конкурентных программных систем. Язык программирования, построенный в соответствии с технологией и содержащий средства поддержки технологии, как правило, пользуется успехом и бывает «долгожителем». Поэтому изучение объектно-ориентированного программирования должно начинаться с рассмотрения объектно-ориентированной технологии проектирования программных систем.

Объектно-ориентированный анализ и проектирование принципиально отличаются от популярных предшествующих технологий: восходящего проектирования и нисходящего (структурного) проектирования. Теперь нужно по-другому представлять себе процесс декомпозиции, как объекта автоматизации, так и автоматизированной системы. Архитектура получающегося программного продукта в значительной степени выходит за рамки традиционных для структурного программирования представлений.

Объектно-ориентированная технология анализа и проектирования программных систем основывается на рассмотренной в главе 1 *объектной модели* и рассмотренной в главе 2 *системе обозначений.*

В этой главе мы выясним, чем является и чем не является объектно-ориентированная разработка программ, и в чем отличия этого подхода к проектированию программных изделий от других подходов.

Мы также рассмотрим доводы в пользу применения объектно-ориентированного анализа и проектирования для преодоления сложности, связанной с разработкой программных систем. Кроме того, мы определим ряд фундаментальных преимуществ, достигаемых в результате применения такого подхода. Наконец, мы должны изучить принципы, на которых этот процесс проектирования основан: абстрагирование, инкапсуляцию, модульность, иерархичность, типизацию, параллелизм и сохраняемость.

3.1. Среда и объект проектирования

3.1.1. Проблемы создания программных систем

Существует много проблем создания программных систем. Одни проблемы появляются и исчезают, как только находится приемлемое решение. Другие проблемы со временем не проходят, потому что они связаны с постепенным улучшением с помощью компьютера качества жизни человека, причем каждое удовлетворение потребностей вызывает у людей желание получить новые удобства. Перечислим некоторые непреходящие проблемы, не требующие особых комментариев.

1. Постоянный рост потребности в новых программах. Даже в периоды кризисов и спада производства объявление «Требуется программист» никогда не исчезает с досок объявлений.

2. Возрастание сложности решаемых с помощью компьютера задач. Например, еще недавно большим достижением считалось решение задачи об оптимальном раскрое листа железа на детали. Сейчас решаются задачи создания интегрированных систем автоматизированного проектирования.

3. Медленный рост производительности труда программиста. Статистика примерно такая: через каждые десять лет производительность труда программиста увеличивается на 10–15%. Для сравнения приведем аналогичный показатель роста производительности компьютера, за те же 10 лет производительность компьютера растет на порядок.

4. Проблема надежности программных систем. Эта проблема комплексная, но мы лишь напомним об известном всем «зависании» программы или, еще хуже, операционной системы. Наконец, мы должны доверять компьютерным вычислениям, иначе, зачем они вообще нам нужны.

5. Защита программ и данных от несанкционированного доступа. Понятие «взламывание программы» известно всем.

Существует много проблем организационного и технологического характера. Рассмотрим некоторые из них.

Проблема создания сложных программных систем. Сложность задач автоматизации порождает сложность программ. Проблемы, которые мы пытаемся решить с помощью программного обеспечения, часто неизбежно содержат сложные элементы, а к соответствующим программам предъявляется множество различных, порой взаимоисключающих требований.

Возникает вопрос: чем сложная программа принципиально отличается от несложной программы? Попытаемся ответить на него. Существует множество программ, которые задумываются, разрабатываются, сопровождаются и используются одним и тем же человеком. Обычно это начинающий программист или профессионал, работающий изолированно. Мы не хотим сказать, что все такие программные системы плохо сделаны или, тем более, усомниться в квалификации их создателей. Но такие системы, как правило, имеют очень ограниченную область применения и короткое время жизни. Нас интересует разработка того, что мы будем называть *промышленными программными продуктами, промышленными программными изделиями.* Они применяются для решения

самых разных задач. Системы подобного типа обычно имеют большое время жизни, и большое количество пользователей оказывается в зависимости от их нормального функционирования. Существенная черта промышленной программы — уровень сложности: один разработчик практически не в состоянии охватить все аспекты такой системы. Наверное, это и есть главный признак сложности программ: сложность промышленных программ превышает возможности отдельно взятого человеческого интеллекта.

Рассмотрим характеристики электронной системы многомоторного самолета, сотовой телефонной коммутаторной системы, робота. Достаточно трудно понять, даже в общих чертах, как работает каждая такая система, а перед программистами стоит задача улучшения ее характеристик. Более того, заказчик предъявляет к программам требования (часто неформулируемые явно), такие как удобство, производительность, стоимость, выживаемость и надежность! Наконец, основная задача разработчиков программ состоит в создании иллюзии простоты, в защите пользователей от сложности описываемого предмета или процесса.

Сложность часто возникает из-за недопонимания между пользователями-заказчиками системы и ее разработчиками: пользователи с трудом могут объяснить в форме, понятной разработчикам, что на самом деле нужно сделать. Бывают случаи, когда пользователь лишь смутно представляет, что ему нужно от будущей программной системы. Это в основном происходит не из-за ошибок с той или иной стороны; просто каждая из групп специализируется в своей области, и ей недостает знаний партнера. У пользователей и разработчиков разные взгляды на сущность проблемы, и они делают различные выводы о возможных путях ее решения. На самом деле, даже если пользователь точно знает, что ему нужно, то с трудом может однозначно зафиксировать свои требования.

Дополнительные сложности возникают в результате изменений требований к программной системе уже в процессе разработки. В основном требования корректируются из-за того, что само осуществление программного проекта часто изменяет проблему. Рассмотрение первых результатов — схем, прототипов — и использование системы после того, как она разработана и установлена, заставляют пользователей лучше понять и отчетливей сформулировать то, что им действительно нужно. В то же время этот процесс повышает квалификацию разработчиков в предметной области и позволяет им задавать более осмысленные вопросы, которые проясняют темные места в проектируемой системе.

Проблема индустриализации производства программных изделий. Домостроительная компания обычно не имеет собственного лесхоза, который бы ей поставлял лес для пиломатериалов; совершенно необычно, чтобы монтажная фирма соорудила свой завод для изготовления железобетонных панелей под будущее здание. Однако в программной индустрии такая практика — дело обычное. Программирование обладает предельной гибкостью, и разработчик может сам обеспечить себя всеми необходимыми элементами, относящимися к любому уровню абстракции. Такая гибкость чрезвычайно соблазнительна. Она позволяет разработчику создавать своими силами все базовые строительные блоки будущей конструкции, из которых составляются элементы более высо-

ких уровней абстракции. В отличие от строительной индустрии, где существуют единые стандарты на многие конструктивные элементы и качество материалов, в программной индустрии таких стандартов почти нет. В среде программистов нередко встречается мнение: «Быстрее самому написать программный модуль, чем разбираться в чужом!» Идеологи объектно-ориентированного программирования рассматривали класс как маленькое изделие и в качестве одного из основных достоинств технологии называли возможность торговли классами. Прошло уже более тридцати лет, но утверждать о существовании такой индустрии может только отъявленный оптимист.

Проблема управления процессом разработки. Всего 50 лет назад программы объемом в несколько тысяч строк на языке ассемблера выходили за пределы наших возможностей. Сегодня обычными стали программные системы, размер которых исчисляется десятками тысяч или даже миллионами строк на языках высокого уровня. Ни один человек никогда не сможет полностью понять такую систему. Даже если мы правильно разложим ее на составные части, мы все равно получим сотни, а иногда и тысячи отдельных модулей. Поэтому такой объем работ потребует привлечения команды разработчиков, в идеале как можно меньшей по численности. Но какой бы ни была численность команды, всегда будут возникать значительные трудности, связанные с организацией коллективной разработки. Чем больше разработчиков, тем сложнее связи между ними и тем сложнее координация, особенно если участники работ географически удалены друг от друга, что типично в случае очень больших проектов. Таким образом, при коллективном выполнении проекта главной задачей руководства является поддержание единства и целостности разработки.

Показательным примером является организация разработки популярной в настоящее время программы создания и ведения баз данных mySQL. Вначале М. Видениус задумал создать «хранилище данных». В 1995 году программа оформилась как небольшая система управления базой данных и вышла на рынок. К 2000 году к развитию программы было привлечено уже несколько десятков разработчиков. В 2010 году коллектив разработчиков насчитывает несколько сотен человек, причем территориально разработчики распределены по всем континентам (пока они не обнаружены в Антарктиде). Общение и координация работ осуществляется с помощью современных информационных технологий. Для решения ключевых вопросов коллектив собирается в назначенном месте. Уже неоднократно сменилась даже фирма, а коллектив остается, и развитие программной системы успешно продолжается.

Проблема сложности описания поведения больших дискретных систем. Внутри большой прикладной программы могут существовать сотни и даже тысячи переменных и несколько потоков управления. Полный набор этих переменных, их текущих значений, текущего адреса и стека вызова для каждого процесса описывает состояние прикладной программы в каждый момент времени. Так как исполнение нашей программы осуществляется на цифровом компьютере, мы имеем систему с дискретными состояниями. Аналоговые системы, такие как движение брошенного мяча, напротив, являются непрерывными. Когда мы говорим, что система описывается непрерывной функцией, мы имеем

в виду, что в ней нет скрытых сюрпризов. Небольшие изменения входных параметров всегда вызовут небольшие изменения выходных. С другой стороны, дискретные системы по самой своей природе имеют конечное число возможных состояний, хотя в больших системах это число в соответствии с правилами комбинаторики очень велико. Мы стараемся проектировать системы, разделяя их на части так, чтобы одна часть минимально воздействовало на другую. Однако переходы между дискретными состояниями не могут моделироваться непрерывными функциями. Каждое событие, внешнее по отношению к программной системе, может перевести ее в новое состояние, и, более того, переход из одного состояния в другое не всегда детерминирован. При неблагоприятных условиях внешнее событие может нарушить текущее состояние системы из-за того, что ее создатели не смогли предусмотреть все возможные варианты. Представим себе пассажирский самолет, в котором системы управления полетом, комфорта пассажиров и электроснабжения объединены. Было бы очень неприятно, если бы от включения пассажиром индивидуального освещения самолет немедленно вошел бы в глубокое пике. В непрерывных системах такое поведение было бы невозможным, но в дискретных системах остается возможность непредсказуемого изменения внутреннего состояния системы. Это, очевидно, и является главной причиной обязательного тестирования программных систем; но дело в том, что за исключением самых тривиальных случаев всеобъемлющее тестирование таких программ провести невозможно. И пока у нас нет ни математических инструментов, ни интеллектуальных возможностей для полного моделирования поведения больших дискретных систем, мы должны удовлетвориться разумным уровнем уверенности в их правильности.

Последствия неограниченной сложности. Наше неумение создавать сложные программные системы проявляется в проектах, которые выходят за рамки установленных сроков и бюджетов и к тому же не соответствуют начальным требованиям. Это приводит к разбазариванию человеческих ресурсов — самого драгоценного товара — и к существенному ограничению возможностей создания новых продуктов. Сейчас просто не хватает хороших программистов, чтобы обеспечить всех пользователей нужными программами. Более того, существенный процент персонала, занятого разработками, в любой организации часто должен заниматься сопровождением и сохранением устаревших программ. С учетом прямого и косвенного вклада индустрии программного обеспечения в развитие экономики большинства ведущих стран, нельзя позволить, чтобы существующая ситуация осталась без изменений.

Как мы можем изменить положение дел? Так как проблема возникает в результате сложности структуры программных продуктов, мы предлагаем сначала рассмотреть способы работы со сложными структурами.

Контрольные вопросы

1. Охарактеризуйте проблему создания сложных программных систем.
2. Опишите состояние проблемы индустриализации производства программных изделий.

3. В чем заключается проблема управления разработкой программных изделий?

4. Опишите причины возникновения проблемы, связанной со сложностью описания поведения программных систем.

3.1.2. Признаки сложной системы

Можно вывести пять общих признаков любой сложной системы.

1. *Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые, в свою очередь, также могут быть разделены на подсистемы и т. д., вплоть до самого низкого уровня.*

Тот факт, что многие сложные системы имеют почти разложимую иерархическую структуру, является главным фактором, позволяющим нам понять, описать и даже «увидеть» такие системы и их части. В самом деле, скорее всего, мы можем понять лишь те системы, которые имеют иерархическую структуру. Важно осознать, что архитектура сложных систем складывается и из компонентов, и из иерархических отношений этих компонентов. Все системы имеют подсистемы, и все системы являются частями более крупных систем. Особенности системы обусловлены отношениями между ее частями, а не частями как таковыми.

Что же следует считать простейшими элементами системы? Опыт подсказывает нам следующий ответ:

2. *Выбор, какие компоненты в данной системе считаются элементарными, относительно произволен и в большой степени оставляется на усмотрение исследователя.*

Низший уровень для одного наблюдателя может оказаться достаточно высоким для другого. Например, для специалиста по бухгалтерским системам база данных может рассматриваться как элементарный компонент, однако с таким утверждением не согласятся специалисты по разработке самих баз данных.

3. *Внутрикомпонентная связь обычно сильнее, чем связь между компонентами.*

Это обстоятельство позволяет отделять «высокочастотные» взаимодействия внутри отдельного компонента от «низкочастотной» динамики взаимодействия между компонентами. Различие внутрикомпонентных и межкомпонентных взаимодействий обусловливает разделение функций между частями системы и дает возможность относительно изолированно изучать каждую часть.

Многие сложные системы организованы достаточно экономными средствами. Поэтому можно отметить следующий признак сложных систем:

4. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.

Сложные системы часто содержат одинаковые структурные части. Эти части могут использовать общие более мелкие компоненты.

Сложные системы имеют тенденцию к развитию во времени, причем их развитие из простых систем гораздо быстрее, если для них существуют устойчивые промежуточные формы.

5. Любая работающая сложная система является результатом развития работавшей более простой системы.

Сложная система, спроектированная «с нуля», никогда не заработает. Следует начинать с работающей простой системы.

В процессе развития системы объекты, первоначально рассматриваемые как сложные, становятся элементарными, и из них строятся более сложные системы. Более того, невозможно сразу правильно создать элементарные объекты: с ними надо сначала повозиться, чтобы больше узнать о реальном поведении системы, и затем уже совершенствовать их.

Контрольные вопросы

1. Перечислите пять признаков сложной системы.

3.1.3. Каноническая модель сложной системы

Обнаружение общих абстракций и механизмов значительно облегчает понимание сложных систем. Например, опытный пилот, сориентировавшись всего за несколько минут, может взять на себя управление многомоторным реактивным самолетом, на котором он раньше никогда не летал, и спокойно его вести (позволим себе такое преувеличение). Определив элементы, общие для всех подобных самолетов (такие как руль управления, элероны и дроссельный клапан), пилот затем найдет отличия этого конкретного самолета от других. Если пилот уже знает, как управлять одним самолетом определенного типа, ему гораздо легче научиться управлять другим похожим самолетом.

Этот пример наводит на мысль, что мы обращались с термином *иерархия* в весьма приблизительном смысле. Наиболее интересные сложные системы содержат много разных иерархий. В самолете, например, можно выделить несколько систем: питания, управления полетом и т. д. Такое разбиение дает структурную иерархию типа «быть частью» (part of). В этом случае говорят, что элементы системы вступают друг с другом в отношение агрегации. Эту же систему можно разложить совершенно другим способом. Например, реактивный двигатель — особый тип (разновидность) авиационного двигателя; турбореактивный двигатель — особый тип реактивного двигателя. С другой стороны, понятие «авиационный двигатель» обобщает свойства, присущие всем авиационным двигателям; «реактивный двигатель» — это просто особый тип авиационного двигателя со свойствами, которые отличают его, например, от поршневого двигателя. Свойства турбореактивного двигателя позволяют легко отличить его от реактивных-прямоточных. Эта вторая иерархия представляет собой иерархию типа «такой же как» (is-a). В этом случае говорят, что элементы вступают друг с другом в отношение наследования. Назовем эти иерархии *структурой классов* (наследование) и *структурой объектов* (агрегация).

Сложные системы включают также и другие типы иерархии. Особое значение имеют их модульная структура, которая описывает отношения между физическими компонентами системы, и иерархия процессов, которая описывает отношения между динамическими компонентами. Существуют и другие виды иерархий. Таким образом, многомодельность является принципом моделирования сложных систем. Этот принцип представляет собой утверждение о том, что

никакая единственная модель не может с достаточной степенью адекватности описывать различные аспекты сложной системы.

Объединяя структурные понятия с пятью признаками сложных систем, мы приходим к тому, что фактически все сложные системы можно представить одной и той же канонической моделью, которая показана на рисунке 3.1. Каждую сложную систему можно представить множеством ортогональных иерархических моделей. Каждая иерархия является многоуровневой, причем в ней элементы более высокого уровня построены из более простых элементов. Какой уровень сделать элементарным зависит от рассматриваемой задачи. Объекты одного уровня имеют четко выраженные связи, особенно это касается компонентов структуры объектов. Внутри любого рассматриваемого уровня находится следующий уровень сложности.

Показывая множество иерархий, мы демонстрируем избыточность рассматриваемой модели. Например, если бы мы не знали структуру классов нашей системы, нам пришлось бы повторять одни и те же сведения для каждого экземпляра класса. С введением структуры классов мы размещаем в ней общие свойства экземпляров классов.

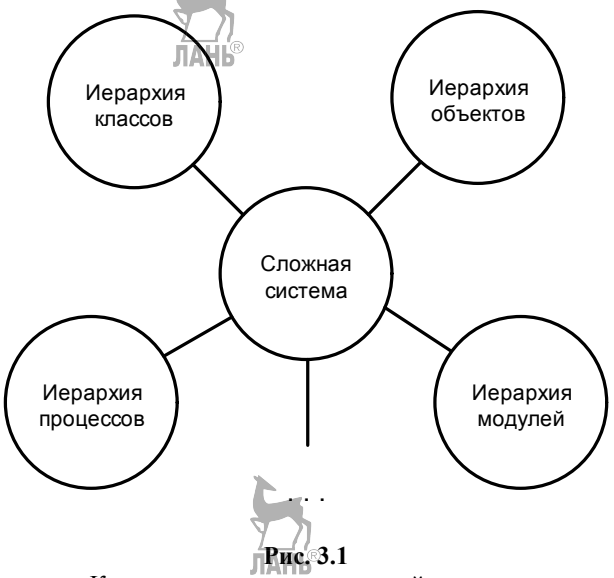


Рис. 3.1
Каноническая модель сложной системы

Опыт показывает, что наиболее успешны те программные системы, в которых заложены хорошо продуманные структуры классов и объектов и которые обладают пятью признаками сложных систем, описанными выше. Оценим важность этого наблюдения и выразимся более категорично: очень редко можно встретить программную систему, разработанную точно по графику, уложившуюся в бюджет, удовлетворяющую требованиям заказчика, и в которой не были бы учтены соображения, изложенные выше.

Структуры классов и объектов системы вместе будем называть *архитектурой* системы.

Контрольные вопросы

1. Какая идея заложена в каноническую модель сложной системы?
2. Что называется архитектурой сложной системы?

3.1.4. Человеческие возможности и сложные системы

Если мы знаем, как должны быть спроектированы сложные программные системы, то почему при создании таких систем мы сталкиваемся с серьезными проблемами? Существует, однако, еще одна, по-видимому, главная причина: физическая ограниченность возможностей человека при работе со сложными системами. Когда мы начинаем анализировать сложную систему, в ней обнаруживается много составных частей, которые взаимодействуют друг с другом различными способами, причем ни сами части системы, ни способы их взаимодействия не обнаруживают никакого сходства. Это пример неорганизованной сложности. Когда мы начинаем организовывать систему в процессе ее проектирования, необходимо думать сразу о многом. Например, в системе управления движением самолетов приходится одновременно контролировать состояние многих летательных аппаратов, учитывая такие их параметры, как местоположение, скорость и курс. При анализе дискретных систем необходимо рассматривать большие, сложные и не всегда детерминированные пространства состояний. К сожалению, один человек не может следить за всем этим одновременно. Эксперименты психологов показывают, что максимальное количество структурных единиц информации, за которыми человеческий мозг может одновременно следить, приблизительно равно 7±2. Вероятно, это связано с объемом краткосрочной памяти у человека.

Таким образом, мы оказались перед серьезной дилеммой. Сложность программных систем возрастает, но способность нашего мозга справиться с этой сложностью ограничена. Как же нам выйти из создававшегося затруднительного положения?

Роль декомпозиции. Способ управления сложными системами был известен еще в древности — divide et impera (разделяй и властвуй). При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. В этом случае мы не превысим пропускной способности человеческого мозга: для понимания любого уровня системы нам необходимо одновременно держать в уме информацию лишь о немногих ее частях.

Декомпозицией системы называется разложение ее на все меньшие и меньшие подсистемы.

Алгоритмическая декомпозиция представляет собой разделение алгоритмов (процессов), где каждый модуль системы выполняет один из этапов общего процесса.

Объектно-ориентированная декомпозиция рассматривает мир как совокупность автономных действующих лиц (объектов), которые взаимодействуют друг с другом, чтобы обеспечить поведение системы, соответствующее более высокому уровню. Таким образом, каждый объект обладает своим собственным поведением и каждый из них моделирует некоторый объект реального мира.

С этой точки зрения объект является вполне осязаемой вещью, которая демонстрирует вполне определенное поведение. Объекты что-то умеют делать, и мы можем, послав им сообщение, попросить их выполнить то-то и то-то. Так как наша декомпозиция основана на объектах, а не на алгоритмах, мы называем ее *объектно-ориентированной декомпозицией.*

Какая декомпозиция сложной системы правильнее — по алгоритмам или по объектам? В этом вопросе есть подвох, и правильный ответ на него: важны оба аспекта. Разделение по алгоритмам концентрирует внимание на порядке происходящих событий, а разделение по объектам придает особое значение агентам, которые являются либо объектами, либо субъектами действия. Однако мы не можем сконструировать сложную систему одновременно двумя способами, тем более что эти способы, по сути, ортогональны. Мы должны начать разделение системы либо по алгоритмам, либо по объектам, а затем, используя полученную структуру, попытаться рассмотреть проблему с другой точки зрения.

Опыт показывает, что полезнее начинать с объектной декомпозиции. Такое начало поможет нам лучше справиться с приданием организованности сложным программным системам. Этот объектный подход помогает нам при описании таких непохожих систем, как компьютеры, растения, галактики и общественные институты. Объектная декомпозиция имеет несколько чрезвычайно важных преимуществ перед алгоритмической декомпозицией. Объектная декомпозиция уменьшает размер программных систем за счет повторного использования общих механизмов, что приводит к существенной экономии выразительных средств. Объектно-ориентированные системы более гибки и проще эволюционируют со временем, потому что их схемы базируется на устойчивых промежуточных формах. Объектная декомпозиция существенно снижает риск при создании сложной программной системы, так как она развивается из меньших систем, в которых мы уже уверены. Более того, объектная декомпозиция помогает нам разобраться в сложной программной системе, предлагая нам разумные решения относительно выбора подпространства большого пространства состояний.

Многое из сказанного можно отнести и к алгоритмической декомпозиции систем. Приведем еще одно соображение в пользу объектной декомпозиции. Объектная декомпозиция более близка человеческому пониманию окружающего мира и в процессе эволюции непрерывно совершенствовалась. Поэтому не случайно технология проектирования программных систем пришла к использованию объектной технологии в качестве основной.

Роль абстракции. Выше мы ссылались на эксперименты психологов, в которых было установлено, что обычно человек может одновременно воспринять лишь 7±2 единицы информации. Это число, по-видимому, не зависит от содержания информации. Размер нашей памяти накладывает жесткие ограничения на количество информации, которое мы можем воспринять, обработать и запомнить. Организуя поступление входной информации одновременно по нескольким различным каналам и в виде последовательности отдельных порций,

мы можем прорвать этот информационный затор. В современной терминологии это называют выделением *абстракций.*

Люди развили чрезвычайно эффективную технологию преодоления сложности: мы абстрагируемся от нее. Будучи не в состоянии полностью воссоздать сложный объект, мы просто игнорируем не слишком важные детали и, таким образом, имеем дело с обобщенной, идеализированной моделью объекта. Например, изучая процесс фотосинтеза у растений, мы концентрируем внимание на химических реакциях в определенных клетках листа и не обращаем внимание на остальные части — черенки, жилки и т. д. И хотя мы по-прежнему вынуждены охватывать одновременно значительное количество информации, благодаря абстракции мы пользуемся единицами информации существенно большего семантического объема. Это особенно верно, когда мы рассматриваем мир с объектно-ориентированной точки зрения, поскольку объекты как абстракции реального мира представляют собой отдельные насыщенные связные информационные единицы.

Роль иерархии. Другим способом, расширяющим информационные единицы, является организация внутри системы иерархий классов и объектов. Объектная структура важна, так как она иллюстрирует схему взаимодействия объектов друг с другом, которая осуществляется с помощью *механизмов* взаимодействия. Структура классов не менее важна: она определяет общность структур и поведения внутри системы. Зачем, например, изучать фотосинтез каждой клетки отдельного листа растения, когда достаточно изучить одну такую клетку, поскольку мы ожидаем, что все остальные ведут себя подобным же образом. И хотя мы рассматриваем каждый объект определенного типа как отдельный, можно предположить, что его поведение будет похоже на поведение других объектов того же типа. Классифицируя объекты по группам родственных абстракций (например, типы клеток растений в противовес клеткам животных), мы четко разделяем общие и уникальные свойства разных объектов, что помогает нам, затем, справляться со свойственной им сложностью.

Определить иерархии в сложной программной системе не всегда легко, так как это требует разработки моделей многих объектов, поведение каждого из которых может отличаться чрезвычайной сложностью. Однако после их определения, структура сложной системы и, в свою очередь, наше понимание ее сразу во многом проясняются.

Контрольные вопросы

1. Какое противоречие заложено в сложности окружающих нас систем и возможностях человеческого мозга?
2. Какова роль декомпозиции в управлении сложными системами?
3. Чем отличаются алгоритмическая и объектная декомпозиции сложных систем?
4. Какова роль абстракции в понимании сложных систем?
5. Расскажите о роли иерархии в понимании сложных систем.

3.1.5. Жизненный цикл программного изделия

Понятие жизненного цикла программного изделия появилось в тот момент, когда программисты осознали необходимость перехода от кустарных методов создания программных систем к индустриальным методам. Самым распространенным мотивом использования понятия жизненного цикла изделия явилась потребность планирования работ в соответствии с технологическим процессом. Модели жизненного цикла программного изделия являются основой знания технологий программирования и используемых в работе инструментов: любая технология программирования связывает с этапами жизненного цикла специфические наборы профессиональных приемов и применяемые для успешной работы инструменты. С другой стороны, разбиение жизненного цикла программного изделия на этапы позволяет облегчить усилия по планированию работ и управлению работами. Укрупненно жизненный цикл программного продукта разбивается на две фазы: разработка и сопровождение.

Разработка начинается с осознания потребности в автоматизации выбранной области человеческого труда и заканчивается передачей программной системы в эксплуатацию. Более детально фаза разработки разбивается на этапы: определение требований, спецификация, проектирование, реализация, тестирование, сдача в эксплуатацию.

Определение требований включает общее описание задачи, описание основных функций автоматизированной системы, описание ограничений.

Содержанием *спецификации* программы является результат осмысления выдвигаемых заказчиком требований. Главным на этом этапе является ответ на вопрос: что нужно сделать? Назначением спецификации является описание внешнего поведения системы. Спецификация должна быть понятна как разработчикам системы, так и заказчикам. Прежде чем приступать к разработке проекта по спецификациям, они должны быть тщательно проверены на соответствие исходным целям, полноту, совместимость и однозначность.

Документально результаты первых двух этапов оформляются в виде (технического) задания на проектирование программного изделия.

Разработка проектных решений, отвечающих на вопрос, как должна быть реализована система, чтобы она могла удовлетворять специфицированным требованиям, выполняется на этапе *проектирование*. Главной задачей этого этапа является последовательная декомпозиция до уровня таких сущностей или модулей, которые могут быть реализованы локально. Документально этот этап оформляется в виде технического проекта.

На уровне *реализации*, или *кодирования*, каждый модуль программируется на подходящем языке программирования и отлаживается.

Этап *тестирования* включает разработку содержательных тестов и проверку поведения разработанной программы. Часто этот этап называют этапом тестирования и комплексной отладки. Иногда этап комплексной отладки выделяется в отдельный этап.

Документально последние два этапа оформляются в виде рабочего проекта.

Фаза разработки завершается этапом *передачи программы в эксплуатацию*.

Необходимость внесения изменений в действующую программу приводит к тому, что разработка программного обеспечения продолжается и после передачи его в эксплуатацию в течение всего жизненного цикла. Деятельность, связанная с решением многочисленных задач такой продолжающейся разработки, получила название «сопровождение программного обеспечения».

Фаза выполняющихся одновременно эксплуатации и сопровождения программы включает в себя всю деятельность по обеспечению нормального функционирования программных систем, в том числе фиксацию обнаруженных во время эксплуатации программы ошибок, поиск их причин и исправление, повышение эксплуатационных характеристик системы, адаптацию системы к окружающей среде, а также, при необходимости, и более существенные работы по развитию системы.

Большая программная система — это крупное капиталовложение, и мы не можем позволить себе выкидывать сделанное при каждом изменении внешних требований. Тем не менее даже большие системы имеют тенденцию к эволюции в процессе их использования. Чтобы быть более точными, введем несколько понятий:

— под *сопровождением* программы понимается устранение обнаруженных ошибок;

— под *развитием* — усовершенствование системы в ответ на изменившиеся требования к ней;

— под *сохранением* — использование всех возможных и невозможных способов для поддержания жизни в дряхлой и распадающейся на части системе, но при этом остающейся еще полезной, необходимой.

Опыт показывает, что на фазу эксплуатации и сопровождения программ затрачивается наибольшая часть средств, выделенных на обеспечение всего жизненного цикла программного изделия, из них существенный процент затрат тратится именно на сохранение программных систем.

Рассмотренная модель жизненного цикла программного изделия является идеальной. На практике, как правило, в нее вносится итеративность. Итеративность означает, что на каждом этапе при обнаружении недостатков возможен возврат к предыдущим этапам, с последующей переделкой проектных решений и самой программы. Итеративность неизбежна при разработке сложных программных изделий, а поэтому ее целесообразно внести в основу жизненного цикла изделия и технологию разработки программ. Однако вплоть до появления объектно-ориентированного программирования подходы к развитию проектов не пытались использовать итеративность в качестве метода проектирования и стремились лишь к минимизации возвратов. Объектно-ориентированное программирование сразу поставило итеративность в основу своих технологий и своих моделей жизненного цикла, более того, по этой причине в настоящее время итеративные методы часто отождествляют с объектно-ориентированными методами. Однако поддержка итеративности разработки не создается автоматически. Программист должен постоянно осознавать, что программа будет переделываться, и должен постоянно заботиться о придании программе свойства простой модифицируемости.

Модифицируемость программы означает:

а) по возможности легкое понимание исходного текста программы;

б) локализация программных блоков (объектов) таким образом, чтобы любые изменения программы требовали пересмотра как можно меньшей части исходного текста программы.

Модифицируемость как свойство программы не появляется автоматически, его нужно заложить в программу во время проектирования и реализации. Легкости понимания программы в немалой степени способствуют внимательное отношение к структурному оформлению текста программы и использование правил именования идентификаторов (см. Приложение 2). Правила именования идентификаторов становятся в настоящее время стандартом фирм-разработчиков программного обеспечения. Локализация программных блоков заложена в самой объектной организации программы при условии квалифицированной объектной декомпозиции системы. Специальным средством обеспечения модифицируемости программы является полиморфизм.

Контрольные вопросы

1. Охарактеризуйте понятие жизненного цикла программного изделия и причины введения этого понятия.
2. Перечислите основные этапы жизненного цикла программного изделия.
3. Сформулируйте содержание каждого этапа жизненного цикла программного изделия.
4. Что означает итеративность в модели жизненного цикла программного изделия?
5. Охарактеризуйте понятие модифицируемости программы.

3.2. Проектирование программных систем

3.2.1. Инженерное дело как наука, искусство и ремесло

На практике любая инженерная дисциплина, будь то строительство, механика, химия, электроника или программирование, содержит в себе элементы и науки, и искусства. Разработка новых структур предполагает и полет фантазии, и синтез опыта и знаний: все то, что необходимо художнику для реализации своего замысла на холсте или бумаге. После того, как этот замысел созрел в голове инженера-художника, он обязательно должен быть проанализирован с точки зрения применимости данного научного метода инженером-ученым со всей тщательностью, присущей настоящему ученому. Аналогично, программная постановка задачи является упражнением в применении абстракции и требует способностей как инженера-математика, так и компетентного инженера-системотехника.

Когда разрабатывается совершенно новая система, роль инженера как художника выдвигается на первый план. Это происходит постоянно при проектировании программ, а тем более при работе с системами, обладающими обратной связью, и особенно в случае систем управления и контроля, когда нам приходится создавать программное обеспечение, требования к которому нестандартны.

В других случаях, например при создании прикладных научных средств, инструментов для исследований в области искусственного интеллекта или даже для систем обработки информации, требования к системе могут быть хорошо и точно определены, но определены таким образом, что соответствующий им технический уровень разработки выходит за пределы существующих технологий. Нам, например, могут предложить создать систему, обладающую большим быстродействием, большей вместимостью или имеющей гораздо более мощные функциональные возможности по сравнению с существующими системами. Во всех этих случаях мы будем стараться использовать знакомые абстракции и механизмы (устойчивые промежуточные формы) как основу новой системы. При наличии большой библиотеки повторно используемых программных компонентов инженер-программист должен их по-новому скомпоновать, чтобы удовлетворить всем явным и неявным требованиям к системе точно так же, как художник или музыкант находит новые возможности своего инструмента.

Контрольные вопросы

1. Почему инженерное дело рассматривается одновременно как наука, искусство, ремесло?

3.2.2. Смысл проектирования

В любой инженерной дисциплине под проектированием обычно понимается некий унифицированный подход, с помощью которого мы ищем пути решения определенной проблемы, обеспечивая выполнение поставленной задачи. В контексте инженерного проектирования целью проектирования является получение описания искусственной системы (артефакта), которое:

— удовлетворяет заданным (возможно, неформальным) функциональным спецификациям;

— согласовано с ограничениями, накладываемыми оборудованием;

— удовлетворяет явным и неявным требованиям по эксплуатационным качествам и ресурсопотреблению;

— удовлетворяет явным и неявным критериям дизайна продукта;

— удовлетворяет требованиям к самому процессу разработки, таким, например, как продолжительность и стоимость, а также привлечение дополнительных инструментальных средств.

Цель проектирования — выявление ясной и относительно простой внутренней структуры, называемой архитектурой. Описание архитектуры называется проектом. Проект есть окончательный продукт процесса проектирования. Проектирование подразумевает учет противоречивых требований. Его результатами являются модели, позволяющие нам понять структуру будущей системы, сбалансировать требования и наметить схему реализации.

Контрольные вопросы

1. Какая цель инженерного проектирования?
2. Каким требованиям должен удовлетворять искусственный объект (артефакт)?

3. В данном параграфе инженерное проектирование рассмотрено абстрактно. Порассуждайте, как перечисленные в параграфе требования интерпретируются в случае, когда искусственным объектом является программное изделие.

3.2.3. Важность построения модели

Моделирование широко распространено во всех инженерных дисциплинах, в значительной степени из-за того, что оно реализует принципы декомпозиции, абстракции и иерархии. Каждая модель описывает определенную часть рассматриваемой системы, а мы, в свою очередь, строим новые модели на базе старых, в которых более или менее уверены. Модели позволяют нам доказывать свою правоту и контролировать наши неудачи. Мы оцениваем поведение каждой модели в обычных и необычных ситуациях, а затем проводим соответствующие доработки, если нас что-то не удовлетворяет.

Чтобы понять во всех тонкостях поведение сложной системы, приходится использовать не одну модель. Например, проектируя компьютер на одной плате, инженер-электронщик должен рассматривать систему как на уровне отдельных элементов схемы (микросхем), так и на уровне схемы. Схема помогает инженеру разобраться в совместном поведении микросхем. Другая схема представляет собой план физической реализации системы микросхем, в котором (плане) учтены размер платы, потребляемая мощность и типы имеющихся интегральных микросхем. С этой точки зрения инженер может независимо оценивать такие параметры системы, как температурное распределение и технологичность изготовления. Проектировщик платы может также рассматривать динамические и статические особенности системы. Аналогично, инженер-электронщик использует диаграммы, иллюстрирующие статические связи между различными микросхемами, и временные диаграммы, отражающие поведение элементов во времени. Затем инженер может применить осциллограф или цифровой анализатор для проверки правильности и статической, и динамической моделей.

Общим свойством всех моделей является их подобие оригинальной системе. Важность построения моделей заключается в возможности их использования для получения информации о свойствах или поведении системы-оригинала. При этом процесс построения и последующего применения моделей для получения информации о системе-оригинале получил название «моделирование». Общая модель системы содержит некоторую важную информацию о функциональных особенностях данной системы, которые дают представление о ее дальнейшем поведении.

Контрольные вопросы

1. В чем вы видите полезность моделирования систем в процессе проектирования?

3.2.4. Элементы программного проектирования

Ясно, что не существует такого универсального метода, который бы провел инженера-программиста по пути от требований к сложной программной системе до их выполнения. Проектирование сложной программной системы

отнюдь не сводится к слепому следованию некоему набору рецептов. Скорее, это постепенный и итеративный процесс. Использование методологии проектирования вносит в процесс разработки определенную организованность. Инженеры-программисты разработали десятки различных приемов, которые мы можем классифицировать по трем категориям. Несмотря на различия, эти приемы имеют что-то общее. Их, в частности, объединяет следующее:

— условные обозначения — язык для описания каждой модели;

— процесс — правила проектирования модели;

— инструменты — средства, которые ускоряют процесс создания моделей и в которых уже воплощены законы функционирования моделей. Инструменты помогают выявлять ошибки в процессе разработки.

Хороший метод проектирования базируется на прочной теоретической основе и при этом дает программисту известную степень свободы самовыражения.

Объектно-ориентированные модели. Существует ли наилучший метод проектирования? На этот вопрос нет однозначного ответа.

Объектно-ориентированный анализ и проектирование — это метод, логически приводящий нас к объектно-ориентированной декомпозиции. Применяя объектно-ориентированное проектирование, мы создаем гибкие программы, написанные экономными средствами. При разумном разделении пространства состояний мы добиваемся большей уверенности в правильности нашей программы. В итоге мы уменьшаем риск ошибиться при разработке сложных программных систем.

В основе любой технологии проектирования программ лежат:

— представления о *логической* и *физической*, *статической* и *динамической* моделях;

— принципы *абстрагирования*, *инкапсуляции*, *модульности*, *иерархичности*, *типизации*, *параллелизма* и *сохраняемости*.

Понятия моделей появились еще в технологии нисходящего проектирования программ. Принципы проектирования развивались постепенно. В объектно-ориентированной технологии проектирования программных систем перечисленные принципы впервые были применены в полном объеме.

Так как построение моделей крайне важно при проектировании сложных систем, объектно-ориентированное проектирование предлагает богатый выбор моделей, которые представлены на рисунке 3.2. Эти модели покрывают весь спектр важнейших конструкторских решений, которые необходимо рассматривать при разработке сложной системы, и таким образом вдохновляют нас на создание проектов, обладающих всеми пятью атрибутами хорошо организованных сложных систем.

Особенности объектной модели. Методы структурного проектирования помогают упростить процесс разработки сложных систем за счет использования алгоритмов как готовых строительных блоков. Аналогично, методы объектно-ориентированного проектирования созданы, чтобы помочь разработчикам корректно и рационально применять мощные выразительные средства объектно-ориентированного программирования, использующего в качестве блоков классы и объекты.

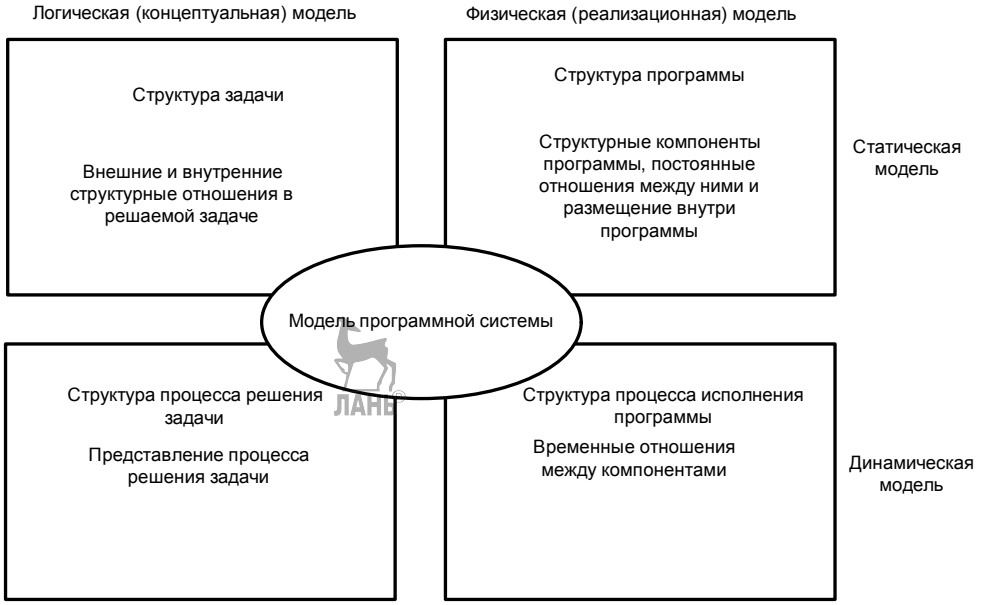


Рис. 3.2
Объектно-ориентированные модели

Но в объектной модели отражается и множество других факторов. Объектный подход зарекомендовал себя как унифицирующая идея всей компьютерной науки, применимая не только в программировании, но также в проектировании интерфейса пользователя, баз данных и даже архитектуры компьютеров. Причина такой широты в том, что ориентация на объекты позволяет нам справляться со сложностью систем самой разной природы.

Объектно-ориентированный анализ и проектирование отражают эволюционное, а не революционное развитие проектирования; новая методология не порывает с прежними методами, а строится с учетом предшествующего опыта. К сожалению, многие программисты в настоящее время формально и неформально натренированы на применение только методов структурного проектирования. Разумеется, многие хорошие проектировщики создали и продолжают совершенствовать большое количество программных систем на основе этой методологии. Однако алгоритмическая декомпозиция помогает только до определенного предела, и обращение к объектно-ориентированной декомпозиции необходимо. Более того, при попытках использовать объектно-ориентированные языки C++ или Object Pascal в качестве алгоритмически ориентированных мы не только теряем их внутренний потенциал — скорее всего результат будет даже хуже, чем при использовании обычных языков C или Pascal. Дать электродрель плотнику, который не слышал об электричестве, значит использовать ее в качестве молотка. Он согнет несколько гвоздей и разобьет себе пальцы, потому что электродрель малопригодна для замены молотка. Скажем более жестко: техника в руках дикаря — это груда металла.

Технология объектно-ориентированного проектирования программ создавалась для решения задач автоматизации проектирования *сложных программ*. Поэтому к ней нужно обращаться только в таких случаях.

Контрольные вопросы

1. Сформулируйте своими словами смысл программного проектирования.
2. Какие основные приемы используются в программном проектировании?
3. Какие представления и принципы используются в объектной технологии программного проектирования?
4. Какие модели программных систем используются в объектно-ориентированном подходе?

3.2.5. Объектно-ориентированные анализ, проектирование и программирование

Объектно-ориентированный анализ направлен на создание моделей реальной действительности на основе объектно-ориентированного мировоззрения.

Объектно-ориентированный анализ — это методология анализа предметной области, при которой выявляются классы и объекты, а требования к системе воспринимаются с точки зрения полученных классов и объектов.

Объектно-ориентированное проектирование основное внимание уделяет правильному и эффективному структурированию сложных систем. Определим объектно-ориентированное проектирование следующим образом.

Объектно-ориентированное проектирование — это методология проектирования, соединяющая в себе процесс объектной декомпозиции и объектные приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

В данном определении содержатся две важные части объектно-ориентированного проектирования:

1) основывается на объектно-ориентированной декомпозиции;

2) использует многообразие приемов представления моделей, отражающих логическую (классы и объекты) и физическую (модули и процессы) структуру системы, а также ее статические и динамические аспекты.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного проектирования программ; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором — алгоритмами.

Объектно-ориентированное программирование определим следующим образом.

Объектно-ориентированное программирование — это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

В данном определении можно выделить три части объектно-ориентированного программирования.

1. Использует в качестве базовых элементов *объекты,* а не алгоритмы.
2. Каждый объект является экземпляром какого-либо определенного класса.
3. Классы организованы иерархически.

Программа будет объектно-ориентированной только при соблюдении всех трех указанных требований.

В соответствии с этим определением не все языки программирования являются объектно-ориентированными. Понятие *объектно-ориентированный язык* включает средства хорошей поддержки объектно-ориентированного стиля программирования. Обеспечение такого стиля, в свою очередь, означает, что в языке удобно пользоваться этим стилем. Если написание программ в стиле объектно-ориентированного программирования требует специальных усилий или оно невозможно совсем, то этот язык не отвечает требованиям объектно-ориентированного программирования. Теоретически возможна имитация объектно-ориентированного программирования в обычных языках, таких как Pascal, C или *язык ассемблера*, но это крайне затруднительно. Язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

— поддерживаются объекты, т. е. абстракции данных, имеющие интерфейс в виде именованных операций и собственные данные, с ограничением доступа к ним;

— объекты относятся к соответствующим классам (типам);

— классы (типы) могут наследовать атрибуты суперклассов (супертипов).

Поддержка наследования в таких языках означает возможность установления отношения «is-a», например, красная роза — это цветок, а цветок — это растение. Языки, не имеющие таких механизмов, нельзя отнести к объектно-ориентированным языкам.

Как соотносятся анализ, проектирование и программирование? На результатах анализа формируются модели, на которых основывается проектирование; проектирование, в свою очередь, создает фундамент для окончательной реализации системы с использованием методологии программирования.

Контрольные вопросы

1. Сформулируйте своими словами смысл проектирования.
2. Сформулируйте определение объектно-ориентированного анализа.
3. Сформулируйте определение объектно-ориентированного проектирования.
4. Сформулируйте определение объектно-ориентированного программирования.
5. Какой язык программирования можно называть объектно-ориентированным языком?

3.3. Принципы и инструменты объектно-ориентированного анализа

Классификация — средство упорядочения знаний. В объектно-ориентированном анализе определение общих свойств объектов помогает найти общие ключевые абстракции и механизмы, что, в свою очередь, приводит нас к более простой архитектуре системы. К сожалению, пока не разработаны строгие методы классификации и нет правила, позволяющего выделять классы и объекты. Нет таких понятий, как «совершенная структура классов», «правильный выбор объектов». Как и во многих технических дисциплинах, выбор классов является компромиссным решением. К счастью, имеется богатый опыт классификации в других науках, на основе которого разработаны методики объектно-ориентированного анализа. Каждая такая методика предлагает свои правила (эвристики) идентификации классов и объектов. Они и будут предметом этого параграфа.

3.3.1. Классификация и объектно-ориентированный анализ

Выделение в решаемой задаче классов и объектов — одна из самых сложных задач объектно-ориентированного анализа. Как показывает опыт, эта работа обычно содержит в себе элементы открытия и изобретения. С помощью открытий человек распознает ключевые понятия и механизмы, которые образуют словарь предметной области. С помощью изобретения мы конструируем обобщенные понятия, а также новые механизмы, которые определяют правила взаимодействия объектов. Поэтому открытие и изобретение — неотъемлемые части успешной классификации. Целью классификации является нахождение общих свойств объектов. Классифицируя, мы объединяем в одну группу объекты, имеющие одинаковое строение или одинаковое поведение.

Классификация есть средство упорядочения знаний.

Разумная классификация, несомненно, часть любой науки. Неотъемлемой задачей науки является построение содержательной классификации наблюдаемых объектов или ситуаций. Такая классификация существенно облегчает понимание основной проблемы и дальнейшее развитие научной теории. Например, проектная деятельность архитектора управляется целиком образами, которые он держит в своем сознании в данный момент, и его способностью комбинировать эти образы при создании нового проекта.

Неудивительно, что классификация затрагивает многие аспекты объектно-ориентированного проектирования. Она помогает определить иерархии обобщения, специализации и агрегации. Найдя общие формы взаимодействия объектов, мы вводим механизмы, которые станут фундаментом реализации нашего проекта. Классификация помогает правильно определить модульную структуру. Мы можем расположить объекты в одном или разных модулях, в зависимости от степени схожести объектов.

Классификация играет большую роль при распределении процессов между процессорами. Мы направляем процессы на один процессор или на разные в зависимости от того, как эти процессы связаны друг с другом.

Контрольные вопросы

1. Сформулируйте своими словами понятие «классификация».
2. Какова роль классификации в процессе познания?

3.3.2. Трудности классификации

В главе 1 объект определен как нечто, имеющее четкие границы. На самом деле это не всегда так. Границы предметов часто неопределенны. Например, посмотрим на нашу ногу. Попытаемся определить, где начинается и кончается колено. В разговорной речи трудно понять, почему именно эти звуки определяют слово, а не являются частью какого-то более длинного слова. Представим себе, что мы проектируем текстовый редактор. Что считать классом — буквы или слова? Как понимать отдельные словосочетания, фразы, предложения, параграфы, документы? Как обращаться с произвольными, не обязательно осмысленными, блоками текста? Что делать с предложениями, абзацами и целыми документами — соответствуют ли такие классы нашей задаче?

То, что разумная классификация — трудная проблема, новостью не назовешь. И поскольку есть параллели с аналогичными трудностями в объектно-ориентированном проектировании, рассмотрим примеры классификации в двух других научных дисциплинах: биологии и химии.

Вплоть до XVIII века идея о возможности классификации живых организмов по степени сложности была господствующей. Мера сложности была субъективной, поэтому неудивительно, что человек оказался в списке на первом месте. В середине XVIII века шведский ботаник Карл Линней предложил более подробную таксономию (классификацию) организмов: он ввел понятия рода и вида. Век спустя Дарвин выдвинул теорию, по которой механизмом эволюции является естественный отбор и ныне существующие виды животных — продукт эволюции древних организмов. Теория Дарвина основывалась на разумной классификации видов. Как утверждает Дарвин, «натуралисты пытаются расположить виды, роды, семейства в каждом классе в то, что называется натуральной системой. Что подразумевается под этой системой? Некоторые авторы понимают некоторую простую схему, позволяющую расположить наиболее похожие живые организмы в один класс и различные — в разные классы». В современной биологии термин «классификация» обозначает установление иерархической системы категорий на основе предположительно существующих естественных связей между организмами. Наиболее общее понятие в биологической таксономии — царство, затем, в порядке убывания общности: тип (отдел), класс, отряд (порядок), семейство, род и, наконец, вид. Исторически сложилось так, что место каждого организма в иерархической системе определяется на основании внешнего и внутреннего строения тела и эволюционных связей. В современной классификации живых существ выделяются группы организмов, имеющих общую генетическую историю, т. е. организмы, имеющие сходные ДНК, включаются в одну группу. Классификация по ДНК полезна, чтобы различить организмы, которые похожи внешне, но генетически сильно

отличаются. По современным воззрениям дельфины ближе к коровам, чем к форели.

Возможно, для программиста биология представляется зрелой, вполне сформировавшейся наукой с определенными критериями классификации организмов. Но это не так: на сегодняшний день классифицировано менее чем 2 млн видов, в то время как возможное число видов оценивается от 5 до 50 млн. Более того, различные критерии классификации одних и тех же животных приводят к разным результатам.

Аналогичная ситуация сложилась и в химии. В древние времена считалось, что все вещества суть комбинации земли, воздуха, огня и воды. В настоящее время такая классификация не может считаться сколько-нибудь удовлетворительной. В середине XVII века Роберт Бойль предложил элементы как примитивные химические абстракции, из которых составляются более сложные вещества. Век спустя, в 1789 году, Лавуазье опубликовал первый список, содержащий 23 элемента, хотя впоследствии было открыто, что некоторые из них таковыми не являются. Но открытие новых элементов продолжалось, список увеличивался. Наконец, в 1869 году Менделеев предложил периодический закон, который давал точные критерии для классификации известных элементов и даже мог предсказывать свойства еще не открытых элементов. Но даже периодический закон не был концом истории о классификации элементов. В начале XX века были открыты элементы с одинаковыми химическими свойствами, но с разными атомными весами — изотопы.

Почему же классификация так сложна? Это можно объяснить двумя причинами. Во-первых, отсутствием «совершенной» классификации, хотя, естественно, одни классификации лучше других. Существует столько способов деления мира на объектные системы, сколько специалистов принимается за эту задачу. Любая классификация зависит от точки зрения субъекта. Возьмем любое государство. Экономисты могут рассматривать его как экономический институт, социологи — как общество, защитники окружающей среды — как гибнущий уголок природы, туристы — как достопримечательность, руководители других государств — как военную угрозу, наконец, наиболее романтичные из нас — как зеленые луга родины. Во-вторых, разумная классификация требует изрядной доли творческого озарения. Иногда ответ очевиден, иногда он — дело вкуса, а бывает, что все зависит от умения заметить главное. Все это напоминает загадку: почему лазерный луч похож на золотую рыбку? Потому, что ни тот, ни другая не умеют свистеть! Надо быть творческим мыслителем, чтобы найти общее в настолько не связанных предметах.

Вывод прост. Открытие порядка — нелегкая задача, но, если он найден, понять его совсем не трудно. Лучшие программистские решения выглядят просто, но, как показывает опыт, добиться простой архитектуры очень трудно.

Контрольные вопросы

1. Какие трудности возникают при попытке классификации предметной области?

3.3.3. Итеративная суть классификации

Все эти сведения приведены здесь не для того, чтобы оправдать «долгострой» в программном обеспечении, хотя на самом деле многим программистам и пользователям кажется, что необходимы века, чтобы закончить начатую работу. Приведенные сведения подчеркивают, что разумная классификация — работа интеллектуальная и лучший способ ее ведения — последовательный, итеративный процесс. Это становится очевидным при анализе разработки таких продуктов, как графический интерфейс, база данных, язык программирования. В разработке программного обеспечения развитие какой-либо абстракции часто следует общей схеме. Вначале проблема решается «как-нибудь», для каждого частного случая. По мере накопления опыта некоторые решения оказываются более удачными, чем другие, и возникает род фольклора, переходящего от человека к человеку. Удачные решения изучаются систематически, они программируются и анализируются. Это позволяет развить модели, осуществить их реализацию и разработать теорию, обобщающую найденное решение[9]. Это, в свою очередь, поднимает практику на более высокий уровень и позволяет взяться за еще более сложную задачу, к которой, в свою очередь, мы подходим «как-нибудь», тем самым начиная новый виток спирали.

Итеративный подход к классификации накладывает соответствующий отпечаток и на процедуру конструирования иерархии классов и объектов при разработке сложного программного обеспечения. На практике обычно за основу берется какая-то определенная структура классов, которую постепенно совершенствуют.

Разные наблюдатели классифицируют один и тот же объект по-разному.

И только на поздней стадии разработки, когда уже получен некоторый опыт использования такой структуры, можно критически оценить качество получившейся классификации. Основываясь на полученном опыте, мы можем создать новый подкласс из уже существующих (вывод) или разделить большой класс на много маленьких (факторизация), или, наконец, слить несколько существующих в один (композиция). Возможно, в процессе разработки будут найдены новые общие свойства, ранее не замеченные, и мы сможем определить новые классы (абстракция).

Со времен Платона проблема классификации занимала умы бесчисленных философов, лингвистов, когнитивистов, математиков. Поэтому было бы разумно изучить накопленный опыт и применить его в объектно-ориентированном проектировании. Исторически известны три подхода к классификации:

— классическая категоризация;
— концептуальная кластеризация;
— теория прототипов.

[9] Примером могут служить известные алгоритмы поиска слова в таблице: поиск перебором, бинарный поиск, хэш-поиск.

Контрольные вопросы

1. В чем заключается итеративная суть классификации?

3.3.4. Классическая категоризация

В классическом подходе все вещи, обладающие схожей совокупностью свойств, формируют некоторую категорию. Причем наличие этих свойств является необходимым и достаточным условием, определяющим категорию. Например, холостые люди — это категория: каждый человек или холост, или женат, и этот признак достаточен для решения вопроса, к какой категории принадлежит тот или иной индивидуум. С другой стороны, высокие люди не определяют категории, если, конечно, мы специально не уточним критерий, позволяющий четко отличать высоких людей от невысоких.

Классическая категоризация пришла к нам от Платона и Аристотеля. Последний в своей классификации растений и животных пользовался техникой рассуждений, напоминающей современную детскую игру в вопросы (Это минерал, животное или растение? Это покрыто мехом или перьями? Может ли оно летать? Пахнет ли оно? и т. д.).

Принципы классической категоризации отражены в современной теории развития ребенка. После первого года жизни ребенок осознает существование объектов и затем начинает приобретать навыки их классификации, вначале пользуясь базовыми категориями, такими как собаки, кошки и игрушки. Позднее ребенок осознает, с одной стороны, более общие категории (животные), а с другой стороны, более частные категории (колли, доги, овчарки).

Таким образом, классический подход в качестве критерия похожести объектов использует родственность их свойств. В частности, объекты можно разбивать на непересекающиеся множества в зависимости от наличия или отсутствия некоторого признака. В основе классификации можно взять такие наборы свойств, элементы которых мало взаимодействуют между собой. Этим объясняется всеобщая любовь к таким критериям, как размер, цвет, форма и материал. Так как эти критерии не пересекаются, про какой-нибудь предмет можно утверждать, что он большой, серый, круглый и деревянный. Вообще говоря, свойства не обязательно должны быть измеряемыми, в качестве них можно использовать наблюдаемое поведение. То обстоятельство, что птицы летают, а рыбы нет, позволяет отличить орла от стерляди.

Какие конкретно свойства надо принимать во внимание? Это зависит от обстановки. Например, цвет автомобиля надо зафиксировать в задаче учета продукции автомобилестроительного завода, но он неинтересен программе, управляющей уличным светофором. Вот почему мы говорим, что нет абсолютного критерия классификации, одна и та же структура классов может подходить для одной задачи и не годиться для другой. Нельзя утверждать, что некоторая схема классификации лучше других отражает структуру и порядок вещей в природе. Природе безразличны наши попытки в ней разобраться. Некоторые классификации действительно важнее других, но только в связи с нашими интересами, а не потому, что они вернее или полнее отражают реальность.

Современное мышление по большей части насквозь пропитано классической категоризацией, однако, как показывает пример с высокими и низкими людьми, этот подход не всегда работает. Часто естественные категории не четко отграничены друг от друга. Большинство птиц летает, но не все. Стул может быть деревянным, металлическим или пластмассовым, а количество ног у него целиком зависит от прихоти конструктора. Практически невозможно перечислить определяющие свойства естественной категории так, чтобы не было исключений. Это, действительно, коренные пороки классической категоризации, которые и попытались исправить в последующих подходах.

Контрольные вопросы

1. Какой критерий похожести используется в методе классификации «классическая категоризация»?
2. Почему нет абсолютного критерия классификации?

3.3.5. Концептуальная кластеризация

Это современный вариант классического подхода. Он возник из попыток формального представления знаний. При таком подходе сначала формируются концептуальные описания классов (кластеров объектов), а затем мы классифицируем сущности в соответствии с этими описаниями. Например, возьмем понятие «любовная песня». Это именно понятие, а не признак или свойство, поскольку степень любовности песни едва ли можно измерить. Но если можно утверждать, что песня, скорее, про любовь, чем про что-то другое, то мы помещаем ее в эту категорию.

Концептуальную кластеризацию можно связать с теорией нечетких множеств, в которой объект может принадлежать к нескольким категориям одновременно с разной степенью точности. Концептуальная кластеризация делает в классификации абсолютные суждения, основываясь на наилучшем согласии.

Контрольные вопросы

1. Охарактеризуйте метод классификации «концептуальная кластеризация».

3.3.6. Теория прототипов

Классическая категоризация и концептуальная кластеризация — достаточно выразительные методы, вполне пригодные для проектирования сложных программных систем. Но все же есть ситуации, в которых эти методы не работают. Рассмотрим более современный метод классификации — теорию прототипов.

Существуют некоторые абстракции, которые не имеют ни четких свойств, ни четкого определения. Существуют категории (например, игры), которые не соответствуют классическим образцам, так как нет признаков, свойственных всем играм. По этой причине их можно объединить так называемой семейной схожестью. У категории игр нет четкой границы. Категорию можно расширить и включить новые виды игр при условии, что они напоминают уже известные игры. Вот почему этот подход называется теорией прототипов: класс определяется одним объектом-прототипом, и новый объект мож-

но отнести к классу при условии, что он наделен существенным сходством с прототипом.

Попытаемся применить классификацию на основе прототипов к упомянутой выше проблеме стульев. Будем считать мягкий пуф, парикмахерское кресло и складной стул стульями не потому, что они удовлетворяют некоторому фиксированному набору признаков прототипа, но потому, что они имеют достаточное фамильное сходство с прототипом. Не требуется никакого общего набора свойств прототипа, которое годилось бы и для пуфика, и для парикмахерского кресла, но они оба — стулья, так как каждый из них в отдельности похож на прототипный стул, пусть даже каждый по-своему. Свойства, определяемые при взаимодействии с объектом (свойства взаимодействия), являются главными при определении семейного сходства.

Понятие свойств взаимодействия — центральное для теории прототипов. В концептуальной кластеризации мы группируем в соответствии с различными концепциями. В теории прототипов классификация объектов производится по степени их сходства с конкретным прототипом.

Контрольные вопросы

1. Какой прием используется в теории прототипов для образования класса?

3.3.7. Применение классических и новых теорий

Разработчику, озабоченному постоянно меняющимися требованиями к системе и вечно сражающемуся с напряженным планом при ограниченных ресурсах, предмет нашего обсуждения может показаться далеким от реальности. В действительности три рассмотренных подхода к классификации имеют непосредственное отношение к объектно-ориентированному проектированию.

На практике мы идентифицируем классы и объекты сначала по свойствам, важным в данной ситуации, т. е. стараемся выделить и отобрать структуры и типы поведения с помощью словаря предметной области. Потенциально возможных абстракций, как правило, очень много. Если таким путем не удалось построить удобоваримой структуры классов, мы пробуем концептуальный подход. В этом случае в центре внимания находится поведение объектов, когда они взаимодействуют друг с другом. Наконец, мы пробуем выделить прототипы и ассоциировать с ними объекты.

Эти три способа классификации составляют теоретическую основу объектно-ориентированного подхода к анализу, предлагающего много практических советов и правил, которые можно применить для идентификации классов и объектов при проектировании сложной программной системы.

Контрольные вопросы

1. В каком порядке рекомендуется применять методы классификации?

3.3.8. Объектно-ориентированный анализ

Границы между стадиями анализа и проектирования размыты, но решаемые ими задачи определяются достаточно четко. В процессе анализа мы моделируем проблемную область, *обнаруживая* классы и объекты. Результатом яв-

ляется словарь предметной области. При объектно-ориентированном проектировании мы *изобретаем* абстракции и механизмы, обеспечивающие поведение, требуемое моделью. Теперь мы рассмотрим несколько проверенных практикой подходов к анализу объектно-ориентированных систем.

Классические подходы. Разные ученые находят различные источники классов и объектов, согласующихся с требованиями предметной области. Рассмотрим три подхода, которые можно назвать классическими, поскольку они опираются на классическую категоризацию.

Первый подход

Осязаемые предметы	Автомобили, телеметрические данные, датчики давления
Роли	Мать, учитель, политик
События	Посадка, прерывание, запрос
Взаимодействие	Заем, встреча, пересечение

Второй подход

Люди	Человеческие существа, выполняющие некоторые функции
Места	Области, связанные с людьми или предметами
Предметы	Осязаемый материальный объект или группа объектов
Организации	Формально организованная совокупность людей, ресурсов, оборудования, которая имеет определенную цель и существование которой в целом не зависит от индивидуумов
Концепции	Принципы и идеи, сами по себе неосязаемые, но предназначенные для организации деятельности и/или общения, или же для наблюдения за ними
События	Нечто случающееся с чем-то в заданное время или последовательно

Третий подход

Структуры	Отношения «целое-часть» и «общее-частное»
Внешние системы	Системы, с которыми взаимодействует приложение
Устройства	Устройства, с которыми взаимодействует приложение
События	Происшествия, которые должны быть запомнены
Разыгрываемые роли	Роли, которые исполняют пользователи, работающие с приложением
Места	Здания, офисы и другие места, существенные для работы приложения
Организационные единицы	Группы, к которым принадлежат пользователи

Анализ поведения. В то время как классические подходы концентрируют внимание на осязаемых элементах предметной области, другая школа объектно-ориентированного анализа сосредотачивается на динамическом поведении объектов. Это напоминает концептуальную кластеризацию, рассмотренную выше: мы формируем классы, основываясь на группах объектов, демонстрирующих сходное поведение.

Предлагается понятие ответственности объекта, под которыми следует понимать его знания и умения. Ответственность — это способ выразить цель объекта и его место в системе. Ответственность объекта есть совокупность всех

услуг, которые он может предоставлять по всем его контрактам. То есть мы объединяем вместе те объекты, которые имеют сходные ответственности и строим иерархию классов, в которой каждый подкласс, выполняя обязательства суперкласса, привносит свои дополнительные услуги.

Предлагается идентифицировать классы и объекты, анализируя функционирование системы. Такой подход основан на изучении поведения системы. Формы поведения сопоставляются с частями системы, при этом пытаются понять, какая часть инициирует поведение и какие части в нем участвуют. Инициаторы и участники, играющие существенные роли, опознаются как объекты, которые назначаются ответственными за эти роли.

Другой взгляд основан на анализе функций объекта. Функция определяется как отдельное действие конечного пользователя: организация ввода/вывода данных или обслуживание запросов, структурирование данных или поддержка интерфейса. Очевидно, что эта концепция происходит из области информационных систем. Однако она может быть применена к любой автоматизированной системе. По существу, функция — это любое достоверно видимое извне и имеющее отношение к решаемой задаче поведение системы.

Анализ других приложений из предметной области. До сих пор мы неявно имели в виду единственное разрабатываемое нами приложение. Но иногда в поисках полезных и уже доказавших свою работоспособность идей полезно обратиться сразу ко всем приложениям в рамках данной предметной области, как, например, ведение историй болезни пациентов, торговля ценными бумагами, разработка компиляторов или систем управления ракетами. Если мы находимся в середине разработки и застряли, анализ какой-нибудь узкой предметной области может помочь, указав нам на ключевые абстракции, оказавшиеся полезными в сходных системах. Анализ приложений из предметной области работает очень хорошо, исключая, разве что, лишь очень специальные ситуации, так как уникальные программные системы встречаются крайне редко.

Мы определим такой анализ как попытку выделить те объекты, операции и связи, которые эксперты данной области считают наиболее важными. Выделяются следующие этапы в анализе области:

— построение скелетной модели предметной области при консультациях с экспертами в этой области;

— изучение существующих в данной области систем и представление результатов в стандартном виде;

— определение сходства и различий между системами при участии экспертов;

— уточнение общей модели для приспособления к нуждам конкретной системы.

Анализ приложений из предметной области можно вести относительно аналогичных приложений (горизонтально) или относительно аналогичных частей одного и того же приложения (вертикально). Например, начиная проектировать систему учета пациентов, имеет смысл рассмотреть уже имеющиеся подобные системы, чтобы понять, какие ключевые абстракции и механизмы, использованные в них, будут вам полезны, а какие нет. Аналогично система

бухгалтерского учета должна представлять различные виды отчетов. Если считать отчеты некой предметной областью, ее анализ может привести разработчика к пониманию ключевых абстракций и механизмов, которые обслуживают все виды отчетов. Полученные таким образом классы и объекты представляют собой множество ключевых абстракций и механизмов, отобранных с учетом цели исходной задачи: создания системы отчетов. Поэтому окончательный проект будет проще.

Анализ вариантов. По отдельности классический подход, поведенческий подход и изучение предметной области, рассмотренные выше, сильно зависят от индивидуальных способностей и опыта аналитика. Для большинства реальных проектов одновременное применение всех трех подходов неприемлемо, так как процесс анализа становится недетерминированным и непредсказуемым.

Анализ вариантов — это подход, который можно успешно сочетать с первыми тремя, делая их применение более упорядоченным. Вариант применения — это частный пример или образец использования, сценарий, начинающийся с того, что пользователь системы инициирует операцию или последовательность взаимосвязанных событий.

Коротко говоря, этот вид анализа можно начинать вместе с анализом требований. В этот момент пользователи, эксперты и разработчики перечисляют сценарии, наиболее существенные для работы системы (пока не углубляясь в детали). Затем они тщательно прорабатывают сценарии, раскладывая их по кадрам, как делают телевизионщики и кинематографисты. При этом они устанавливают, какие объекты участвуют в сценарии, каковы обязанности каждого объекта и как они взаимодействуют в терминах операций. Тем самым группа разработчиков вынуждена четко распределить области влияния абстракций. Далее набор сценариев расширяется, чтобы учесть исключительные ситуации и вторичное поведение. В результате появляются новые или уточняются существующие абстракции.

Неформальное описание. Радикальная альтернатива классическому анализу была предложена в чрезвычайно простом методе. Согласно этому методу надо описать задачу или ее часть на естественном языке, а потом подчеркнуть существительные и глаголы. Существительные — кандидаты на роль классов, а глаголы могут стать именами операций. Этот подход полезен, так как он прост и заставляет разработчика заниматься словарем предметной области. Однако он весьма приблизителен и непригоден для сколько-нибудь сложных проблем. Человеческий язык — ужасно неточное средство выражения, потому список объектов и операций зависит от умения разработчика записывать свои мысли. Тем более что для многих существительных можно найти соответствующую глагольную форму, и наоборот.

Поиск и выбор ключевых абстракций. Ключевая абстракция — это класс или объект, который входит в словарь проблемной области. Самая главная ценность ключевых абстракций заключена в том, что они определяют границы нашей проблемы: выделяют то, что входит в нашу систему и поэтому важно для нас, и устраняют лишнее. Задача выделения таких абстракций спе-

цифична для проблемной области. Правильный выбор объектов зависит от назначения приложения и степени детальности обрабатываемой информации.

Как мы уже отмечали, определение ключевых абстракций включает в себя два процесса: открытие и изобретение. Мы открываем абстракции, слушая специалистов по предметной области: если эксперт про нее говорит, то эта абстракция обычно действительно важна. Изобретая, мы создаем новые классы и объекты, не обязательно являющиеся частью предметной области, но полезные при проектировании или реализации системы. Например, пользователь банкомата говорит «счет, снять, положить»; эти термины — часть словаря предметной области. Разработчик системы использует их, но добавляет свои, такие как «база данных», «диспетчер экрана», «список», «очередь» и т. д. Эти ключевые абстракции созданы уже не предметной областью, а проектированием.

Наиболее мощный способ выделения ключевых абстракций — сводить задачу к уже известным классам и объектам. При отсутствии таких повторно используемых абстракций мы рекомендуем пользоваться сценариями, чтобы вести процесс идентификации классов и объектов.

Уточнение ключевых абстракций. Определив кандидатов на роли ключевых абстракций, мы должны оценить их по некоторым критериям. Программист должен задаваться вопросами: Как создаются объекты? Как можно копировать и/или уничтожать объекты данного класса? Какие операции могут быть выполнены над этим объектом? Если ответы на эти вопросы туманны, то, возможно, общая концепция не ясна и лучше еще подумать, чем бросаться программировать.

Определив новые абстракции, мы должны найти их место в контексте уже существующих классов и объектов. Не стоит пытаться делать это строго сверху-вниз или снизу-вверх. Нет особой необходимости строить иерархию классов, начиная с самого верхнего класса, и потом дополнять ее подклассами. Удобнее создать несколько независимых иерархий, осознать их общие черты и выделить один или несколько суперклассов. Требуется несколько проходов вверх и вниз по иерархии, чтобы создать программный проект. Это наблюдение, основанное на опыте и подтверждающее тот факт, что объектно-ориентированное проектирование — процесс последовательных приближений.

Классы и объекты должны быть на надлежащем уровне абстракции: не слишком высоко и не слишком низко.

Трудно сразу расположить классы и объекты на правильных уровнях абстракции. Иногда, найдя важный класс, мы можем передвинуть его вверх в иерархии классов, тем самым увеличивая степень повторности использования кода. Это называется продвижением класса. Аналогично можем прийти к выводу, что класс слишком обобщен, и это затрудняет наследование.

Программисты часто легкомысленно относятся к правильному наименованию классов и объектов, но на самом деле очень важно отразить в обозначении классов и объектов сущность описываемых ими предметов. Программы необходимо писать тщательно, как художественную литературу, думая и о читателях, и о компьютере. При идентификации одного только объекта нам нужно придумать имена: для него, для его класса и для модуля, в котором класс

объявлен. Умножьте на тысячу объектов и сотни классов, и вы поймете, как остра проблема.

Часто используются следующие правила:

— объекты следует называть существительными: **Sensor** (датчик) или **Shape** (форма);

— классы следует называть обобщенными существительными: **Sensors**, **Shapes**;

— операции-модификаторы следует называть активными глаголами: **Draw**, **MoveLeft**;

— у операций-селекторов в имя должен включаться запрос или форма глагола «to be»: **ExtentOf**, **IsOpen**;

— подчеркивание и использование заглавных букв — на ваше усмотрение, постарайтесь лишь не противоречить сами себе.

Идентификация механизмов. В предыдущем обсуждении мы называли механизмами структуры, посредством которых объекты взаимодействуют друг с другом и ведут себя так, как требуется. Так же как при разработке класса фактически определяется поведение отдельных объектов, так же и механизмы служат для задания поведения совокупности объектов. Таким образом, механизмы представляют шаблоны поведения.

Рассмотрим требование, предъявляемое к автомобилю: нажатие на «газ» (акселератор) должно приводить к увеличению оборотов двигателя, а отпускание — к их уменьшению. Как это происходит, водителю совершенно безразлично. Может быть использован любой механизм, обеспечивающий нужное поведение, и его выбор — дело вкуса разработчика. Например, допустимо любое из предложенных ниже инженерных решений:

— механическая связь между акселератором и карбюратором (обычное решение);

— под педалью ставится датчик давления, который соединяется с компьютером, управляющим карбюратором (механизм управления по проводам);

— карбюратора нет; бак с горючим находится на крыше автомобиля, и топливо свободно течет в двигатель. Поток топлива регулируется зажимом на трубке. Нажатие на педаль акселератора ослабляет зажим (очень дешевое решение).

Какую именно реализацию выберет разработчик, зависит от таких параметров, как стоимость, надежность, технологичность и т. д.

Подобно тому, как было бы недопустимой невежливостью со стороны клиента нарушать правила пользования сервером, также и выход за пределы правил и ограничений поведения, заданных механизмом, социально неприемлем. Водитель был бы удивлен, если бы, нажав на педаль акселератора, увидел зажегшиеся фары.

Ключевые абстракции определяют словарь проблемной области, механизмы определяют суть проекта. В процессе проектирования разработчик должен придумать не только начинку классов, но и то, как объекты этих классов будут взаимодействовать друг с другом. Но конкретный механизм взаимодействия все равно придется разложить на методы классов. В итоге протокол клас-

са будет отражать поведение его объектов и работу механизмов, в которых они участвуют.

Механизмы, таким образом, представляют собой стратегические решения в проектировании, подобно проектированию структуры классов. С другой стороны, проектирование интерфейса какого-то одного класса — это скорее тактическое решение. Стратегические решения должны быть выполнены явно, иначе у нас получится неорганизованная толпа объектов, кидающихся выполнять работу, расталкивая друг друга. В наиболее элегантных, стройных и быстрых программах воплощены тщательно разработанные механизмы.

Механизмы суть средства, с помощью которых объекты взаимодействуют друг с другом для достижения необходимого поведения более высокого уровня.

Механизмы представляют только один из шаблонов, которые мы находим в структурированных системах. Так, на нижнем конце пирамиды находятся идиомы[10]. Это обороты, специфические для языков программирования или программистских культур и отражающие общепринятые способы выражаться. Определяющей характеристикой идиомы является то, что ее игнорирование или нарушение влечет немедленные социальные последствия: вы превращаетесь в чужака, не заслуживающего уважения. Например, в одних языках программирования не принято использовать подчеркивание в именах функций или переменных, хотя в Object Pascal это дело обычное. Изучая язык, приходится учить его идиомы, которые обычно передаются в форме фольклора. Однако идиомы играют важную роль в кодификации шаблонов низкого уровня. Многие программистские действия идиоматичны, и поэтому распознание таких идиом позволяет выражать функциональности вне самого этого языка с сохранением иллюзии, что они являются частью языка.

При разработке пользовательского интерфейса, напротив, не рекомендуется использовать профессиональные идиомы. Из опыта автора: на экранную форму была помещена кнопка «Переиндексация файла» (базы данных). Сразу же посыпались просьбы пользователей объяснить, что представляет собой «переиндексация файла». После нескольких малоуспешных попыток объяснить указанную идиому далеким от программирования пользователям автор написал на кнопке «Ремонт файла». Вопросы больше не возникали.

Примеры механизмов. Рассмотрим механизм рисования, применяемый обычно в графических интерфейсах пользователя. Для того чтобы представить какой-либо рисунок на экране, необходимы несколько объектов: окно, вид, модель, которую надо показать, и, наконец, клиент, который знает, когда надо нарисовать модель, но не знает, как это сделать. Сначала клиент дает окну команду нарисовать себя. Так как окно может включать в себя ряд видов, оно, в свою очередь, приказывает каждому из них нарисовать себя. Каждый вид посылает

[10] Идиома — присущий только данному языку нерасчленимый оборот речи, значение которого не совпадает со значением составляющих его слов, взятых в отдельности, например, русские выражения: спустя рукава, попасть впросак; профессиональные выражения: сортировка пузырьком, магазинный автомат; наконец: взять с потолка — в проектировании классов применять не рекомендуем.

сообщение своей модели нарисовать себя, в результате чего и появляется изображение на экране. В этом механизме модель полностью отделена от окна и вида, в котором она представлена: виды могут посылать сообщения моделям, но модели не могут посылать сообщения видам.

Механизм, таким образом, представляет уровень повторного использования в проектировании, более высокий, чем повторное использование индивидуальных классов. Библиотека визуальных компонентов (VCL — Visual Compinents Library), например, является основой интерфейса пользователя в языке C++Builder. Эта парадигма, в свою очередь, строится на базе механизма зависимостей, который вложен в поведение базовых классов языка C++ (**TObject, TComponent, TControl** и др.).

Примеры механизмов можно найти во многих системах. Структуру операционной системы, например, можно описать на высоком уровне абстракции по тем механизмам, которые используются для диспетчеризации программ. В системах искусственного интеллекта использованы разнообразные механизмы принятия решений. Одним из наиболее распространенных является механизм рабочей области, в которую каждый индивидуальный источник знаний независимо заносит свои сведения. В таком механизме не существует центрального контроля, но любое изменение в рабочей области может явиться толчком для выработки системой нового пути решения поставленной задачи. Имеется ряд общих механизмов в объектно-ориентированных системах, включая шаблоны временных ассоциаций, обработку событий и рассылку сообщений. Во всех случаях эти механизмы проявляются не как индивидуальные классы, а как структуры сотрудничающих классов.

Контрольные вопросы

1. Какая основная задача стадии анализа предметной области?
2. Какая основная задача объектно-ориентированного проектирования?
3. Какое назначение приема «анализ поведения»?
4. Почему полезно использовать «анализ других приложений из предметной области»?
5. В чем заключается полезность приема «анализ вариантов»?
6. В чем заключена полезность приема «неформальное описание»?
7. В чем ценность ключевых абстракций?
8. Что понимается под «идентификацией механизмов»?

3.4. Принципы и инструменты объектно-ориентированного проектирования

Для объектно-ориентированного программирования концептуальной основой является *объектная модель.* Она включает семь элементов:

— абстрагирование;
— инкапсуляция;
— модульность;
— иерархия;

— типизация;
— параллелизм;
— сохраняемость.

Без такой концептуальной основы использовать объектно-ориентированные языки можно. Однако выразительная способность объектно-ориентированного языка будет либо потеряна, либо искажена. Но еще более существенно, что при этом будет мало шансов справиться со сложностью решаемых задач.

3.4.1. Абстрагирование

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

В качестве примера абстракции рассмотрим модель студента в деканатской информационной системе. В нее обязательно войдут атрибуты: номер зачетной книжки, фамилия, имя, отчество, специальность; свойства: быть отличником, быть старостой группы; возможности: сдавать экзамен, получать стипендию. Наверняка в модель студента не войдут атрибуты: цвет волос, рост; свойства: обладать автомобилем, иметь друга; возможности: иметь дополнительный заработок, уметь садиться на шпагат.

Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Существует целый спектр абстракций, начиная с объектов, которые почти точно соответствуют реалиям предметной области, и кончая объектами, не имеющими право на существование. Перечислим в таблице некоторые абстракции, от наиболее полезных к наименее полезным:

Абстракция сущности	Объект представляет собой полезную модель некой сущности в предметной области
Абстракция поведения	Объект состоит из обобщенного множества операций
Абстракция виртуальной машины	Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня
Произвольная абстракция	Объект включает в себя набор операций, не имеющих друг с другом ничего общего

Клиентом называется любой объект, использующий ресурсы другого объекта (называемого *сервером*). Мы будем характеризовать поведение объекта услугами, которые он оказывает другим объектам, и операциями, которые он выполняет над другими объектами. Такой подход концентрирует внимание на внешних проявлениях объекта и приводит к идее *контрактной модели* программирования: внешнее проявление объекта рассматривается с точки зрения его контракта с другими объектами, в соответствии с этим должно быть выполнено и его внутреннее устройство (часто во взаимодействии с другими объектами). Контракт фиксирует все обязательства, которые объект-сервер имеет перед объектом-клиентом. Другими словами, этот контракт определяет *ответственность* объекта — то поведение, за которое он отвечает.

Центральной идеей абстракции является понятие инварианта. *Инвариант* — это некоторое логическое условие, значение которого (истина или ложь) должно сохраняться. Для каждой операции объекта можно задать *предусловия* (инварианты, предполагаемые операцией) и *постусловия* (инварианты, которым удовлетворяет операция). Предусловию должны отвечать значения исходных данных операции, постусловию должны отвечать формируемые операцией результаты. Изменение инварианта нарушает контракт, связанный с абстракцией. В частности, если нарушено предусловие, то клиент не соблюдает свои обязательства и сервер не может выполнить свою задачу правильно. Если же нарушено постусловие, то свои обязательства нарушил сервер и клиент не может более ему доверять. В случае нарушения какого-либо условия возбуждается исключительная ситуация.

Все абстракции обладают как статическими, так и динамическими свойствами. Например, файл как объект требует определенного объема памяти на конкретном устройстве, имеет имя и содержание. Эти атрибуты являются статическими свойствами. Конкретные же значения каждого из перечисленных свойств динамичны и изменяются в процессе использования объекта: файл можно увеличить или уменьшить, изменить его имя и содержимое. Объектно-ориентированный стиль программирования связан с воздействием на объекты. Так, операция над объектом порождает некоторую реакцию этого объекта. Операции, которые можно выполнить по отношению к данному объекту, и реакция объекта на внешние воздействия определяют поведение этого объекта.

Клиенту нет никакого дела до реализации класса, который его обслуживает, до тех пор, пока тот соблюдает свои обязательства. На самом деле, абстракция объекта всегда предшествует его реализации. А после того, как решение о реализации принято, оно должно трактоваться как секрет абстракции, скрытый от большинства клиентов (инкапсулирован). Никакая часть сложной системы не должна зависеть от внутреннего устройства какой-либо другой части. В то время как абстракция помогает людям думать о том, что они делают, инкапсуляция позволяет легко перестраивать программы.

Контрольные вопросы

1. Сформулируйте своими словами понятие «абстрагирование».
2. Как вы понимаете понятие «контрактная модель программирования»?
3. Какие условия называются инвариантом?

3.4.2. Инкапсуляция

Понятие «инкапсуляция» связано с понятием «капсула» — оболочка, в которую помещается объект. Открывая часть оболочки, мы получаем возможность доступа к открытой части содержимого объекта. Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается устройством объекта. Чаще всего инкапсуляция выполняется посредством скрытия информации, т. е. маскировкой всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрываются и внутренняя структура объекта, и реализация его методов.

Инкапсуляция, таким образом, определяет четкие границы между различными абстракциями. Например, при проектировании базы данных принято писать программы так, чтобы они не зависели от физического представления данных; вместо этого сосредотачиваются на схеме, отражающей логическое строение данных. В обоих случаях объекты защищены от деталей реализации объектов более низкого уровня.

Абстракция работает только вместе с инкапсуляцией. Практически это означает наличие двух частей в классе: интерфейса и реализации. *Интерфейс* отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса. Внутренняя *реализация* описывает представление этой абстракции и механизмы достижения желаемого поведения объекта. Принцип разделения интерфейса и реализации соответствует сути вещей: в интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами; реализация скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов.

Итак, инкапсуляцию можно определить следующим образом.

Инкапсуляция — это средство отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Разумная инкапсуляция локализует те особенности проекта, которые могут подвергнуться изменениям. По мере развития системы разработчики могут решить, что какие-то операции выполняются несколько дольше, чем допустимо, а какие-то объекты занимают больше памяти, чем приемлемо. В таких ситуациях часто изменяют внутреннее представление объекта, чтобы реализовать более эффективные алгоритмы или оптимизировать алгоритм по критерию памяти, заменяя хранение данных вычислением. Важным преимуществом ограничения доступа является возможность внесения изменений в объект без изменения других объектов.

Скрытие информации — понятие относительное: то, что спрятано на одном уровне абстракции, обнаруживается на другом уровне. Забраться внутрь объектов можно, правда, обычно требуется, чтобы разработчик класса-сервера об этом специально позаботился, а разработчики классов-клиентов не поленились в этом разобраться.

Контрольные вопросы

1. Опишите своими словами понятие «инкапсуляция».
2. Как инкапсуляция используется совместно с абстракцией?

3.4.3. Модульность

Разделение программы на модули до некоторой степени позволяет уменьшить ее сложность. Однако гораздо важнее тот факт, что внутри модульной программы создаются множества хорошо определенных и документированных интерфейсов. Эти интерфейсы неоценимы для исчерпывающего понимания программы в целом. Классы и объекты составляют логическую структуру системы, они помещаются в *модули,* образующие физическую структуру

системы. Это свойство становится особенно полезным, когда система состоит из многих сотен классов.

В большинстве языков, поддерживающих принцип модульности как самостоятельную концепцию, интерфейс модуля отделен от его реализации. Таким образом, модульность и инкапсуляция ходят рука об руку. Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций.

Модули играют роль физических контейнеров, в которые помещаются определения классов и объектов при логическом проектировании системы. Структура модуля должна быть достаточно простой для восприятия; реализация каждого модуля не должна сильно зависеть от реализации других модулей; должны быть приняты меры для облегчения процесса внесения изменений там, где они наиболее вероятны. Таким образом, программист должен находить баланс между двумя противоположными тенденциями: стремлением скрыть информацию и необходимостью обеспечения видимости тех или иных абстракций в нескольких модулях. Существует следующее правило: особенности системы, подверженные изменениям, следует скрывать в отдельных модулях; в качестве межмодульных можно использовать только те элементы, вероятность изменения которых мала. Все структуры данных должны быть обособлены в модуле; доступ к ним будет возможен для всех процедур этого модуля и закрыт для всех других. Доступ к данным из модуля должен осуществляться только через процедуры данного модуля. Другими словами, следует стремиться построить модули так, чтобы объединить логически связанные абстракции и минимизировать взаимные связи между модулями. Исходя из этого, приведем определение модульности.

Модульность — это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули.

Таким образом, принципы абстрагирования, инкапсуляции и модульности являются взаимодополняющими. Объект логически определяет границы определенной абстракции, а инкапсуляция и модульность делают их физически незыблемыми.

На выбор разбиения на модули могут влиять и некоторые внешние обстоятельства. При коллективной разработке программ распределение работы осуществляется, как правило, по модульному принципу и правильное разделение проекта минимизирует связи между участниками. При этом более опытные программисты обычно отвечают за интерфейс модулей, а менее опытные — за реализацию. На более крупном уровне такие же соотношения справедливы для отношений между субподрядчиками. Абстракции можно распределить так, чтобы быстро установить интерфейсы модулей по соглашению между компаниями, участвующими в работе. Изменения в интерфейсе вызывают много отрицательных эмоций, все эти факторы делают интерфейс крайне консервативным. Что касается документирования проекта, то оно строится, как правило, также по модульному принципу — модуль служит единицей описания и администрирования. Десять модулей вместо одного потребуют в десять раз больше описаний, и поэтому, к сожалению, иногда требования по документированию

влияют на декомпозицию проекта (в большинстве случаев негативно). Могут сказываться и требования секретности: часть кода может быть несекретной, а другая — секретной. Последняя часть тогда выполняется в виде отдельного модуля (модулей).

Свести воедино столь разноречивые требования довольно трудно, но главное уяснить: вычленение классов и объектов в проекте и организация модульной структуры — *независимые* действия. Процесс вычленения классов и объектов составляет часть процесса логического проектирования системы, а деление на модули — этап физического проектирования. Разумеется, иногда невозможно завершить логическое проектирование системы, не завершив физическое проектирование, и наоборот. Два этих процесса выполняются итеративно.

Контрольные вопросы

1. Сформулируйте своими словами понятие «модульность».
2. Как взаимодействуют абстрагирование, инкапсуляция и модульность?

3.4.4. Иерархия

Абстракция — вещь полезная, но всегда, кроме самых простых ситуаций, число абстракций в системе намного превышает наши умственные возможности. Инкапсуляция позволяет в какой-то степени устранить это препятствие, убрав из поля зрения внутреннее содержание абстракций. Модульность также упрощает задачу, объединяя логически связанные абстракции в группы. Но этого оказывается недостаточно.

Значительное упрощение в понимании сложных задач достигается за счет иерархического упорядочивания абстракций. Определим иерархию следующим образом.

Иерархия — это упорядочение абстракций по уровням.

Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия наследования, «is-a») и структура объектов (иерархия целое-часть, «part of»).

Важным элементом объектно-ориентированных систем и основным видом иерархии является концепция наследования. Наследование означает такое отношение между классами, когда один класс заимствует структурную или функциональную часть одного или нескольких других классов (соответственно, *одиночное* и *множественное наследование*). Иными словами, наследование создает такую иерархию абстракций, в которой подклассы наследуют строение от одного или нескольких суперклассов (отношение родитель/потомок). Часто подкласс достраивает или переписывает компоненты вышестоящего класса.

Семантически наследование описывает отношение типа «is-a». Например, медведь есть млекопитающее, дом есть недвижимость и «быстрая сортировка» есть сортирующий алгоритм. Таким образом, наследование порождает иерархию «обобщение-специализация», в которой подкласс представляет собой специализированный частный случай своего суперкласса. «Лакмусовая бумажка»

наследования — обратная проверка; так, если *B* не есть *A*, то *B* не стоит производить от *A*.

Пример. В деканатской информационной системе модель студента целесообразно строить в два этапа. Вначале строим объект «человек» с атрибутами: фамилия, имя, отчество, дата рождения. Затем создаем потомка «студент» с атрибутами: номер зачетной книжки, номер группы, при этом все атрибуты «человека» «студент» получит благодаря наследованию. Аналогично будут наследоваться и операции. «Студент» является специализированным «человеком». Заметим, что «студент» является «человеком», а обратное — неверно. В дальнейшем потомков объекта «человек» можно будет использовать для других целей.

В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общем суперклассе. По этой причине говорят о наследовании как об иерархии *обобщение-специализация.* Суперклассы при этом отражают наиболее общие, а подклассы более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы и даже скрыты. Принцип наследования позволяет упростить выражение абстракций, делает проект менее громоздким и более выразительным. В отсутствие наследования каждый класс становится самостоятельным блоком и должен разрабатываться «с нуля». В этом случае классы лишаются общности, поскольку каждый программист реализует их по-своему. Стройность системы достигается тогда только за счет дисциплинированности программистов. Наследование позволяет вводить в обращение новые программы, как мы обучаем новичков новым понятиям — сравнивая новое с чем-то уже известным.

Принципы абстрагирования, инкапсуляции и иерархии находятся между собой в некоем здоровом конфликте. Абстрагирование данных создает непрозрачный барьер, скрывающий состояние и функции объекта; принцип наследования требует открыть доступ и к состоянию, и к функциям объекта для производных объектов. Для любого класса обычно существуют два вида клиентов: объекты, которые манипулируют с экземплярами данного класса, и подклассы-наследники. Существуют три способа управления инкапсуляцией через наследование: подкласс может получить доступ к переменным экземпляра своего суперкласса, вызвать закрытую функцию и, наконец, обратиться напрямую к суперклассу своего суперкласса. Различные языки программирования по-разному находят компромисс между наследованием и инкапсуляцией. Например, интерфейс класса может быть разделен на три части: закрытую (private), видимую только для самого класса; защищенную (protected), видимую также и для подклассов; и открытую (public), видимую для всех.

Множественное наследование — вещь нехитрая, но оно осложняет реализацию языков программирования. Есть две проблемы: конфликты имен между различными суперклассами и повторное наследование. Первый случай — это когда в двух или большем числе суперклассов определено поле или операция с одинаковым именем. Возникает конфликт имен. Повторное наследование — это когда класс наследует двум классам, а они порознь наследуют одному и тому же четвертому. Получается ромбическая структура наследования, и надо

решить, должен ли самый нижний класс получить одну или две отдельные копии самого верхнего класса? В некоторых языках повторное наследование запрещено, в других конфликт решается «волевым порядком», а в-третьих, это оставляется на усмотрение программиста.

Если иерархия «is-a» определяет отношение наследования (обобщение/специализация), то отношение «part of» (целое/часть) вводит иерархию агрегации.

Агрегация есть во всех языках, использующих структуры или записи, состоящие из разнотипных данных. Но в объектно-ориентированном программировании она обретает новую мощь: агрегация позволяет физически сгруппировать логически связанные структуры, а наследование с легкостью копирует эти общие группы в различные абстракции.

Не следует путать агрегацию классов и агрегацию объектов. Отношение агрегации, определенное на структуре абстракций, допускает как композиционные, так и ассоциативные (фактические) связи между объектами.

Концептуально агрегация может означать, а может и не означать включение одного объекта в другой. Сравним два примера. Первый пример раскрывает связь самолета с фюзеляжем, двигателем, шасси и прочими частями. Самолет полностью владеет своими компонентами. Во втором примере приведем связь акционера с акциями, связь автомобиля с шофером. Физическая связь здесь отсутствует, тем не менее мы имеем отношение агрегации.

Логически агрегация определяет отношение целое-часть. При этом физически она может быть реализована как с помощью композиции, так и с помощью ассоциации.

При композиционном отношении между объектами существует зависимость во времени жизни. Предполагается, что часть не может существовать без целого. При уничтожении объекта-целого уничтожаются и объекты-части. При ассоциативном отношении объекты могут существовать независимо друг от друга по время жизни. Укажем на средства обеспечения безусловной композиции. Если в программе один объект объявлен как обычная переменная (поле) другого объекта, то эти объекты будут находиться в отношении композиции. Связь по времени жизни обеспечивается в этом случае программистом: при создании (удалении) объекта-целое должен быть создан (удален) объект-часть. Часто это выглядит так, что связь по времени жизни обеспечивается компилятором. Например, мы не задумываемся над созданием элементов управления (кнопок, полей редактирования, меток и т. д.), которые переносим на экранную форму во время процесса проектирования. Ассоциативную связь сложно обеспечить таким простым способом. Например, между некоторым объектом-стеком и объектом-элементом стека существует ассоциативная связь. В этом случае предполагается, что объект-элемент стека может существовать и вне объекта-стека. Если мы уничтожим заполненный объект-стек, то объекты-элементы стека окажутся неорганизованными и недоступными. Правильной стратегией является в этом случае следующая: сначала нужно уничтожить объекты-элементы стека (опустошить стек), а затем — сам стек.

Контрольные вопросы

1. Сформулируйте своими словами понятие «иерархичность».
2. Как взаимодействуют между собой абстрагирование, инкапсуляция и иерархия?

3.4.5. Типизация

Понятие *тип* интенсивно используется в теории абстрактных типов данных. Для наших целей достаточно считать, что термины *тип* и *класс* взаимозаменяемы. Тем не менее типы стоит обсудить отдельно, поскольку они выставляют смысл абстрагирования в совершенно другом свете. В частности, мы утверждаем, что *типизация* — это способ защититься от использования объектов одного класса вместо другого или по крайней мере управлять таким использованием.

Типизация заставляет нас выражать наши абстракции так, чтобы язык программирования, используемый в реализации, поддерживал соблюдение принятых проектных решений.

Идея согласования типов занимает в понятии типизации центральное место. Например, возьмем физические единицы измерения. Деля расстояние на время, мы ожидаем получить скорость, а не вес. В умножении температуры на силу смысла нет, а в умножении расстояния на силу — есть. Все это примеры сильной типизации, когда прикладная область накладывает правила и ограничения на использование и сочетание абстракций.

Отметим следующие важные преимущества строго типизированных языков:

— контроль типов позволяет заранее исключить многие загадочные сбои в программах во время их выполнения;

— в большинстве систем процесс «редактирование-компиляция-отладка» утомителен, и раннее обнаружение ошибок просто незаменимо;

— объявление типов улучшает документирование программ;

— многие компиляторы генерируют более эффективный объектный код, если им явно известны типы.

Подробному рассмотрению абстрактных типов данных посвящена глава 4.

Контрольные вопросы

1. Сформулируйте своими словами понятие «типизация».

3.4.6. Параллелизм

Есть задачи, в которых автоматические системы должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном компьютере. Процесс (*поток управления*) — это фундаментальная единица действия в системе. Каждая программа имеет по крайней мере один поток управления, параллельная система имеет много таких потоков: век одних недолог, а другие живут в течение всего сеанса работы системы. Реальная параллельность достигается только

на многопроцессорных системах, а системы с одним процессором имитируют параллельность за счет алгоритмов разделения времени.

Кроме этого, «аппаратного», различия, мы будем различать «тяжелую» и «легкую» параллельность по потребности в ресурсах. «Тяжелые» процессы управляются операционной системой независимо от других, и под них выделяется отдельное защищенное адресное пространство. «Легкие» сосуществуют в одном адресном пространстве. «Тяжелые» процессы общаются друг с другом через операционную систему, что обычно медленно и накладно. Связь «легких» процессов осуществляется гораздо проще, часто они используют одни и те же данные.

Многие современные операционные системы предусматривают прямую поддержку параллелизма, и это обстоятельство очень благоприятно сказывается на возможности обеспечения параллелизма в объектно-ориентированных системах.

В то время как объектно-ориентированное программирование основано на абстракции, инкапсуляции и наследовании, параллелизм главное внимание уделяет абстрагированию и синхронизации процессов. Объект есть понятие, на котором эти две точки зрения сходятся: каждый объект (полученный из абстракции реального мира) может представлять собой отдельный поток управления (абстракцию процесса). Такой объект называется *активным.* Для систем, построенных на основе объектно-ориентированного программирования, мир может быть представлен как совокупность взаимодействующих объектов (агентов), часть из которых является активной и выступает в роли независимых вычислительных центров. На этой основе дадим следующее определение параллелизма: *Параллелизм* позволяет различным объектам действовать одновременно.

Параллелизм — это свойство, отличающее активные объекты от пассивных.

Контрольные вопросы

1. Сформулируйте своими словами понятие «параллелизм».

3.4.7. Сохраняемость

Сохраняемость — это не только проблема сохранения *данных.* В объектно-ориентированном программировании имеет смысл сохранять и *классы,* так, чтобы программы могли правильно интерпретировать данные. Это создает большие трудности по мере увеличения объема данных, особенно если класс объекта вдруг потребовалось изменить.

До сих пор мы говорили о сохранении объектов во времени. В большинстве систем объектам при их создании отводится место в памяти, которое не изменяется и в котором объект находится всю свою жизнь. Однако для распределенных систем желательно обеспечивать возможность перенесения объектов в пространстве так, чтобы их можно было переносить с машины на машину и даже при необходимости изменять форму представления объекта в памяти.

Определим сохраняемость следующим образом.

Сохраняемость — способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.

Контрольные вопросы

1. Сформулируйте своими словами понятие «сохраняемость».

3.4.8. Преимущества объектной модели

Как уже говорилось выше, объектная модель принципиально отличается от моделей, которые связаны с более традиционными методами структурного анализа, проектирования и программирования. Это не означает, что объектная модель требует отказа от всех ранее найденных и испытанных временем методов и приемов. Скорее, она вносит некоторые новые элементы, которые добавляются к предшествующему опыту. Объектный подход обеспечивает ряд существенных удобств, которые другими моделями не предусматривались. Наиболее важно, что объектный подход позволяет создавать системы, которые удовлетворяют пяти признакам хорошо структурированных сложных систем. Есть преимущества, которые дает объектная модель.

Во-первых, объектная модель позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования. Значительным достижением является введение иерархии классов в процессе проектирования. Опыт показал, что при использовании объектных языков вне объектной модели их наиболее сильные стороны либо игнорируются, либо применяются неправильно.

Во-вторых, использование объектного подхода существенно повышает уровень унификации разработки и пригодность для повторного использования не только программ, но и проектов, что, в конце концов, ведет к созданию среды разработки и индустриализации процесса проектирования. Объектно-ориентированные системы часто получаются более компактными, чем их не объектно-ориентированные эквиваленты. А это означает не только уменьшение объема кода программ, но и удешевление проекта за счет использования предыдущих разработок, что дает выигрыш в стоимости и времени.

В-третьих, использование объектной модели приводит к построению систем на основе стабильных промежуточных описаний, что упрощает процесс внесения изменений. Это дает системе возможность развиваться постепенно и не приводит к полной ее переработке даже в случае существенных изменений исходных требований.

В-четвертых, объектная модель уменьшает риск неудачной разработки сложных систем, прежде всего потому, что процесс интеграции растягивается на все время разработки, а не превращается в единовременное событие. Объектный подход состоит из ряда хорошо продуманных этапов проектирования, что также уменьшает степень риска и повышает уверенность в правильности принимаемых решений.

Наконец, объектная модель ориентирована на человеческое восприятие мира, многие люди, не имеющие понятия о том, как работает компьютер, находят вполне естественным объектно-ориентированный подход к системам. Этот фактор очень важен для общения с заказчиками.

Контрольные вопросы

1. Какие преимущества дает объектно-ориентированная модель?

3.4.9. Принципы объектно-ориентированного проектирования

Объектно-ориентированные программы представляют собой набор взаимодействующих объектов некоторых классов, большей частью являющихся абстрактами понятий, обнаруженных в процессе анализа предметной области, для которой создается программа. Правильно найденные и реализованные абстракции — ключевой фактор для достижения необходимой при создании сложного приложения гибкости кода и снижения затрат на сопровождение. В этом параграфе мы рассмотрим принципы, которым необходимо следовать при проведении объектной декомпозиции, т. е. создания иерархии классов.

Принцип единственности абстракции. Правило единственности абстракции было сформулировано Робертом Мартином (R. Martin). Класс должен реализовывать одну и только одну абстракцию, и реализовывать ее полностью, чтобы программист, ее использующий, с одной стороны, не попадал в положение кэрролловской Алисы: «Кот с улыбкой — и то редкость, но уж улыбка без Кота — это я прямо не знаю, что такое!», а с другой — не был бы поставлен перед необходимостью иметь дело с монстрообразными классами, реализующими несколько абстракций одновременно. Очевидно, что реализация в одном классе нескольких абстракций снижает возможность повторного использования класса, однако этот принцип является одним из наиболее часто нарушаемых принципов. Признаками нарушения принципа единственности абстракции являются следующие факторы:

— затруднения с выбором подходящего имени класса;
— большое количество методов в классе;
— большое количество входящих и/или исходящих зависимостей;
— систематическое нарушение других принципов дизайна.

Действительно, если мы затрудняемся дать абстракции имя, ясно говорящее о том, что она делает или чем является, то стоит задуматься: а существует ли такой класс вообще? Большое количество методов в классе чаще всего говорит о том, что разные их подмножества предназначены для совершенно разных целей и, возможно, описывают поведение разных абстракций, собранных по ошибке в один класс. Большое количество зависимостей также должно наводить на мысль о том, что класс использует или используется разными группами других классов для совершенно различных целей и может быть расщеплен на несколько составляющих. Число методов или зависимостей, превышающее 5–7, должно как минимум заставить поискать обоснование.

Принцип инверсии зависимости. Принцип инверсии зависимости был также сформулирован Робертом Мартином. Модули высокого уровня не долж-

ны зависеть от модулей низкого уровня. И те, и другие должны зависеть от абстракций. Абстракции не должны зависеть от деталей реализации. Проиллюстрируем сказанное примером на рисунке 3.3.

На этой диаграмме классов (рис. 3.3) абстракция высокого уровня (алгоритм копирования одного устройства на другое) зависит от таких низкоуровневых понятий, как клавиатура и принтер. Проблемы такого решения очевидны: мы не можем легко добавить поддержку ни новых устройств ввода, ни новых устройств вывода информации, хотя сам алгоритм копирования с одного символьного устройства на другое для всех таких устройств совершенно одинаков. Правильное решение этой задачи могло бы выглядеть так, как показано на диаграмме классов 3.4. Здесь, как мы видим, абстракции высокого (класс **TCopier**) и низкого (**TPrinter** и **TKeyboard**) уровней зависят от введенных высокоуровневых абстракций **Reader** и **Writer**, что повышает гибкость нашего кода, так как теперь мы можем легко расширить программу добавлением новых устройств ввода-вывода, не модифицируя уже реализованные классы.

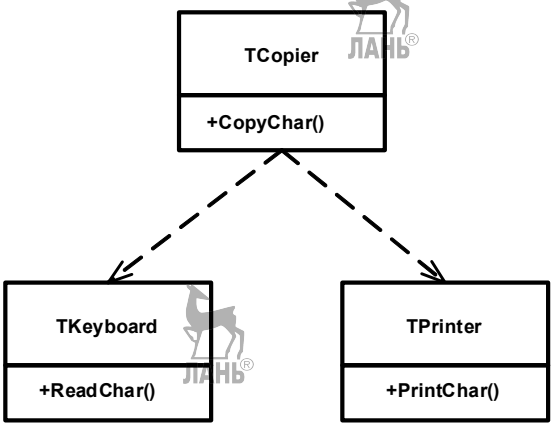


Рис. 3.3
Нарушение принципа инверсии

Принцип Деметера. Одной из основных проблем, снижающих возможности повторного использования программного кода, является сильное *сцепление*, определяемое как мера связанности классов между собой. Чем больше у класса связей (зависимости, наследование, агрегация) с другими классами, тем сильнее его сцепление и тем сложнее его модифицировать и повторно использовать в другой программе.

Рассматриваемый далее принцип Деметера позволяет, при его соблюдении, уменьшить сцепление и, соответственно, повысить гибкость и возможность повторного использования. Оригинальная формулировка этого принципа, возникшая в процессе работы над одноименным проектом (Demeter) в Northeastern University осенью 1987 года, принадлежит Яну Холланду (Ian Holland): каждый элемент программы должен обладать ограниченным знанием о других элементах и использовать только тесно связанные с ним элементы. Как видно из формулировки, принцип Деметера не ограничивается классами, но претендует на правильность в применении ко всем элементам программного

обеспечения, таким как пакеты, модули, процедуры. Здесь мы рассмотрим применение принципа Деметера к иерархиям классов, а поиск аналогий оставим читателю.

В контексте классов под элементом Холланд понимает метод класса. Методами, тесно связанными (closely related) с методом f класса C, называются:

— методы класса C;
— методы классов, являющихся параметрами f;
— методы полей класса C;
— методы классов объектов, созданных в f.

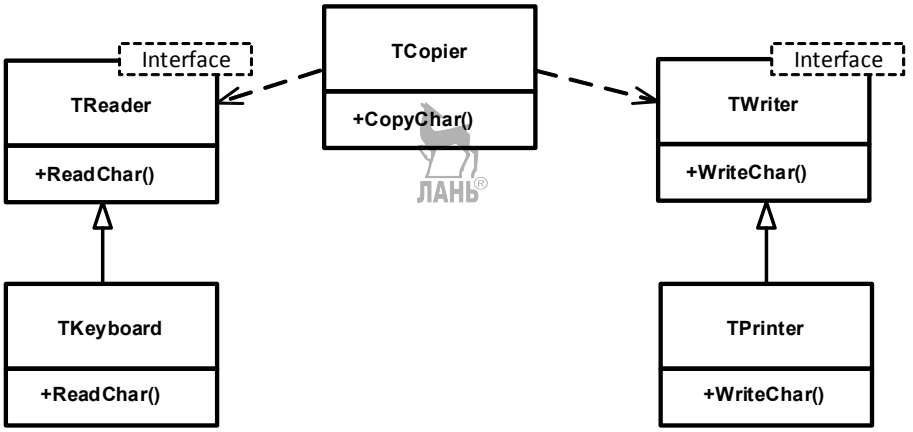


Рис. 3.4
Соблюдение принципа инверсии

Рассмотрим пример (диаграмма на рис. 3.5): отношение между классами является отношением ассоциации, по сути дела — это отношение между экземплярами классов.

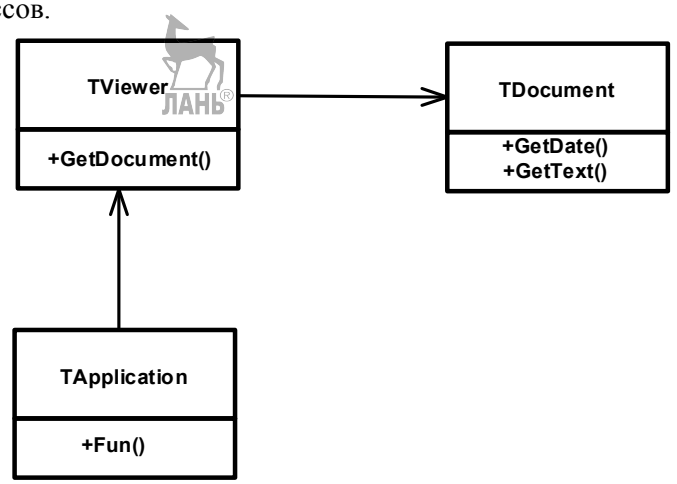


Рис. 3.5
Нарушение принципа Деметера

Попытаемся реализовать методы классов диаграммы на рисунке 3.5.

```
class TApplication
{
   public:
     Viever * TViewer;
      void Fun( )
      {
        Viewer =new TViewer();
        Viewer–>Document–>GetDate();
      }
};
class TViewer
{
 public:
     Document * TDocument;
      void Fun( )
      {
        Document =new TDocument();
        Document–>GetText();
      }
};
```

Обратим внимание на то, что в этом фрагменте программы код класса **TApplication** неявно зависит от методов класса **TDocument**, тем самым порождая две проблемы. Первая состоит в том, что возможные изменения кода не локализованы: попытка изменить класс **TDocument** (а также любой из тех, от которых он зависит) может отразиться на классе **TViewer**, а через него и на **TApplication**. Вторая, самая неприятная для архитектора проблема состоит в том, что связь между **TApplication** и **TDocument** может не попасть на диаграмму классов, так как появляется в процессе написания кода, а не во время построения модели. Формальная проверка программистом кода на соответствие принципу Деметера легко выявляет потенциальную проблему, поскольку метод **Fun** класса **TApplication** обращается к методу **GetDate** класса **TDocument**, который:

— не является методом ни самого класса **TApplication**, ни классов-аргументов метода **Fun**;

— не является методом никакого из полей класса **TApplication**;

— не является методом класса объекта, созданного в **Fun**.

Корень зла содержится, конечно же, не в методе **Fun**, а немножко раньше — в классе **TDocument**, беззаботно открывающем секреты своей реализации незнакомцам. Пример показывает, что нарушение принципа Деметера ведет к неконтролируемому распространению зависимостей между классами, что в конечном счете неизбежно приводит к снижению гибкости кода и увеличению стоимости сопровождения. Действительно, изменение, например, способа доступа к документу в классе **TViewer** становится затруднительным, так как может вызвать каскадное распространение изменений на классы, использующие метод **GetDocument**. Неконтролируемые, а стало быть, потенциально слишком длинные цепочки зависимостей между классами, безусловно, снизят переиспользуемость, поскольку слишком много классов придется переиспользовать «в нагрузку», а также могут

стать причиной появления циклических зависимостей между классами и пакетами. Гради Буч по этому поводу заметил: «Основной эффект от применения закона Деметера — создание слабо связанной иерархии классов, детали реализации которых скрыты друг от друга. Такие классы не должны быть загромождены: чтобы понять один класс, мы не должны понимать детали многих других классов».

Принцип подстановки Лисковой. Следующее правило, предложенное Барбарой Лисковой (Barbara Liskov) в 1988 году, имеет вид: функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Таким образом, идея Лисковой о «подтипе» дает определение понятия **замещения** — если S является подтипом T, тогда объекты типа T в программе могут быть замещены объектами типа S без каких-либо изменений желательных свойств этой программы (например, корректность).

Более простыми словами можно сказать, что поведение наследуемых классов не должно противоречить поведению, заданному базовым классом, т. е. поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа. Практический прием, реализующий данный принцип, выглядит так: функции, которые используют ссылки на базовые классы, должны иметь возможность использовать объекты производных классов, не зная об этом. Скажем по-другому: подкласс не должен требовать от вызывающего кода больше, чем базовый класс, и не должен предоставлять вызывающему коду меньше, чем базовый класс.

Чтобы все вышесказанное не выглядело голословно, приведем пример, в котором принцип подстановки Лисковой нарушен. Пусть в программе имеется базовый класс **Птица**. Естественно, главное функциональное свойство птицы — это возможность летать, выражаемое функцией **Летать()**. Предположим, что от класса **Птица** производятся два класса: **Журавль** и **Страус**, в которых возникает необходимость переопределения метода **Летать()**. Если в классе **Журавль** трудностей не возникает, то в классе **Страус** мы сталкиваемся с фактом, что данный вид птиц не умеет летать.

Рассмотренный пример нарушения принципа Лисковой является достаточно очевидным. Приведем теперь пример, в котором принцип Лисковой нарушается вопреки традиционным представлениям.

Пусть имеется класс **Прямоугольник**:

```
class TRectangle
 {
    private:
        int intHeight; int intWidth;  // соответственно, высота и ширина прямоугольника
    public:
        TRectangle( int w, int h ) { intHeight=h; intWidth=w; }
        void SetHeight( int h ) { intHeight=h; }
        int GetHeight() { return intHeight; }
        void SetWidth( int w ) { intWidth=w; }
        int GetWidth () { return intWidth; }
 };
```

Известно, что квадрат является частным случаем прямоугольника, введем класс **Квадрат** (**Square**) как потомка класса **TRectangle**:

```
class TSquare : public TRectangle
  {
     public:
       TSquare( int s ) { SetHeight(s); SetWidth(s); }     // s — сторона квадрата
  };.
```

Применение наследования в этом случае достаточно типично для начинающих программистов, но не является правильным. Проблема такого решения заключается в потенциальной возможности для пользователя абстракции **TSquare** нарушить ее целостность самым легальным образом, через вызов, например, метода:

```
void F() { TSquare * sq=new TSquare( 6 ); sq–>SetHeight( 5 ); // sq уже не квадрат! }
```

Действительно, после вызова метода **SetHeight** с аргументом 6 объект типа **TSquare**, первоначально инициализированный со значением стороны 5, попросту перестает быть квадратом. Напрашивающийся способ устранения указанной проблемы через переопределение методов **SetHeight** и **SetWidth** в производном классе только на первый взгляд является хорошим решением. Рассмотрим модифицированный пример некорректного исправления нарушения принципа Лисковой. Пусть класс **Square** выглядит теперь так:

```
class TSquare: public TRectangle
 {
  public:
   TSquare( int s ) { ((TRectangle)this)–>SetHeight (s); ((TRectangle)this)–>SetWidth(s); }
   void SetSize (int s) { ((TRectangle)this)–>SetHeight(s); ((TRectangle)this)–>SetWidth(s); }
   void SetHeight( int h ) { SetSize(h); }
   void SetWidth ( int w ) { SetSize(w); }
 };
```

В реализации конструктора класса **TSquare** и метода **SetSize** (установить размер стороны квадрата) используется приведение типа **TSquare** текущего объекта (**this**) к типу предка **TRectangle**. Переопределенные методы предка **SetHeight**, **SetWidth** реализуются теперь с помощью введенного в классе метода **SetSize**.

Рассмотрим теперь метод:

```
void F( TRectangle * r )
  {
     r–>SetHeight(4); r–>SetWidth(5);
    if( r–>GetHight() * r–>GetWidth() != 20 ) throw Exception( “Сюрприз!” );
 }.
```

В методе **F** сначала задаются значения высоты и ширины прямоугольника, а затем проверяются.

Предположим, что некто пытается отладить код, использующий вышеприведенный метод **F**, разработанный другим разработчиком. Иногда метод будет отрабатывать как предполагалось, а иногда выбрасывать исключение. Причем

для того, чтобы понять, что же именно произойдет в каждой конкретной ситуации, разработчик должен знать, чем именно является переданный в метод аргумент — прямоугольником или квадратом. Но ведь метод **F** проектировался как работающий с прямоугольником, и его разработчик мог даже не догадываться, что некогда другой член команды напишет производный класс! Обратно, разработчик класса **TSquare** может не знать, что где-то в программе существует метод, подобный **F**. На самом деле сама возможность написать такой метод показывает, что дизайн класса **TSquare** нарушает принцип подстановки Лисковой, так как поведение метода **F** существенным (и весьма неожиданным для программиста, его написавшего) образом, зависит от того, какому типу принадлежит аргумент — базовому или производному. Таким образом, задать Квадрат наследованием Прямоугольника, не нарушив принцип подстановки, нельзя. Читатель, надо полагать, уже недоумевает: общеизвестно, что квадрат — частный случай прямоугольника, а наследование — как раз способ задания отношения частное-общее, не так ли? Следовательно, либо принцип подстановки неверен, либо наследование — не есть способ задания отношений «частное-общее». Кто прав? Правда, как это часто бывает, находится посередине: принцип верен и наследование прекрасно работает как способ выражения отношения «частное-общее», но не в нашем случае. Дело в том, что принцип подстановки требует от нас при принятии решения о наследовании исходить из общности поведения; мы же, говоря, что Квадрат — частный случай Прямоугольника, неявно исходим из общности их структуры. При том наборе методов, который мы выбрали для Прямоугольника, Квадрат действительно не является его частным случаем в смысле поведения, так как при таком свойстве (поведении) Прямоугольника как способность изменять стороны, Квадрат и Прямоугольник — существенно различные абстракции. Принцип Лисковой, таким образом, может использоваться для доказательства неправильности дизайна отношений типа класс-суперкласс. Доказать правильность в общем случае затруднительно. Набор открытых методов класса, как мы обсуждали ранее, называют контрактом класса. Если развивать эту тему дальше, то можно заметить, что вышеописанный Прямоугольник и Квадрат могут быть связаны отношением, не имеющим явного способа выражения в объектно-ориентированных языках, а именно: **Квадрат** может быть представлен как потомок понятия **Правильный четырехугольник**, который может быть определен как прямоугольник с равными сторонами или ромб с равными углами[11].

Принцип разделения интерфейсов. В следующем принципе речь пойдет о дизайне ассоциаций между классами. Классы не должны зависеть от контрактов, которых они не используют. Принцип достаточно очевиден, ведь никто из нас не захотел бы платить за ненужный ему сервис. Любой из нас как минимум удивится, если ему предложат переоформить договор страхования автомобиля на том основании, что страховая компания изменила политику страхования жилья от пожара. В программировании, однако, люди достаточно легко делают подобные предложения своим товарищам по работе, предлагая им иметь дело с избыточными контрактами классов.

[11] Упражнение для самостоятельного решения.

Пример. Предположим, что мы проектируем систему управления автомобилем (наши рассуждения можно распространить с некоторыми изменениями на автомобиль-беспилотник). В нашей системе управления имеется класс **TControlSystem**, в который, среди прочих, мы включаем поля и методы: **CarNumber** — номер автомобиля, **CurrentSpeed()** — текущая скорость, **StartEngine()** — пуск двигателя, **Accelerate()** — управление мощностью (газ), **Brake()** — тормоз, **LightsOn()** — управление фарами, **ChangeGear()** — управление коробкой передач, **Radio()** — управление магнитолой, **JockeyBox** — бардачок.

Пусть теперь у нас имеются пользователи класса **TControlSystem**: **TDriver** — водитель, **TInspector** — инспектор ГИБДД, **TPassenger** — пассажир. Пользователи могут смотреть на нашу систему управления автомобилем **TControlSystem** по-разному, например, водителю нужно отслеживать текущую скорость, запускать двигатель и т. д., но в процессе управления его не интересует номер автомобиля; инспектора больше интересует номер автомобиля, показания скорости, управление фарами, но безразличны пуск двигателя и бардачок; пассажира будет интересовать управление магнитолой, бардачок и, скорее всего, больше ничего. Мы не можем разделить систему управления на части, так как они пересекаются. Тем не менее класс **TControlSystem** может выступать в разных ролях.

Для решения возникших проблем достаточно создать несколько специализированных классов интерфейса, как показано на рисунке 3.6. Интерфейсы следует передавать соответствующим пользователям. Исходный класс останется при этом неизменным.

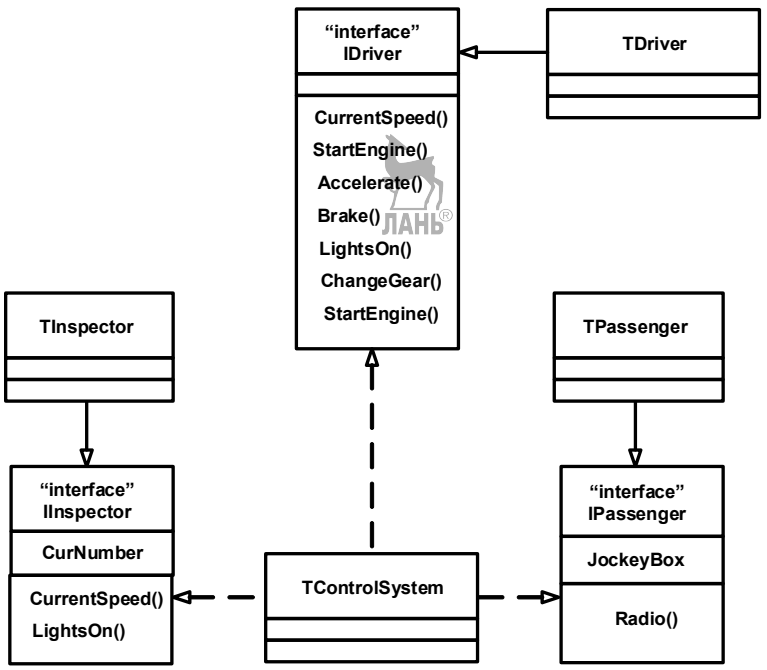


Рис. 3.6
Разделение интерфейса

Принцип ацикличности зависимостей. Структура зависимостей между элементами (классами, пакетами, методами) должна представлять собой направленный ациклический граф (R. Martin, 1996). Хотя Р. Мартин здесь говорит только о классах, можно легко заметить, что этот принцип в равной степени применим ко всем элементам программного обеспечения. Действительно, любые циклически связанные между собой элементы не могут быть использованы отдельно друг от друга, а значит, нет никаких преимуществ в их разделении на отдельные составляющие, недостатки же очевидны: любое изменение элемента цикла влечет перекомпиляцию всех остальных, а минимально возможной единицей переиспользования является весь цикл. Например, в приложении, согласно рисунку 3.7, в результате нарушения принципа ацикличности может возникнуть ситуация, когда какому-либо классу из пакета MyDialogs потребуется доступ к глобальным данным, находящимся в пакете Application.

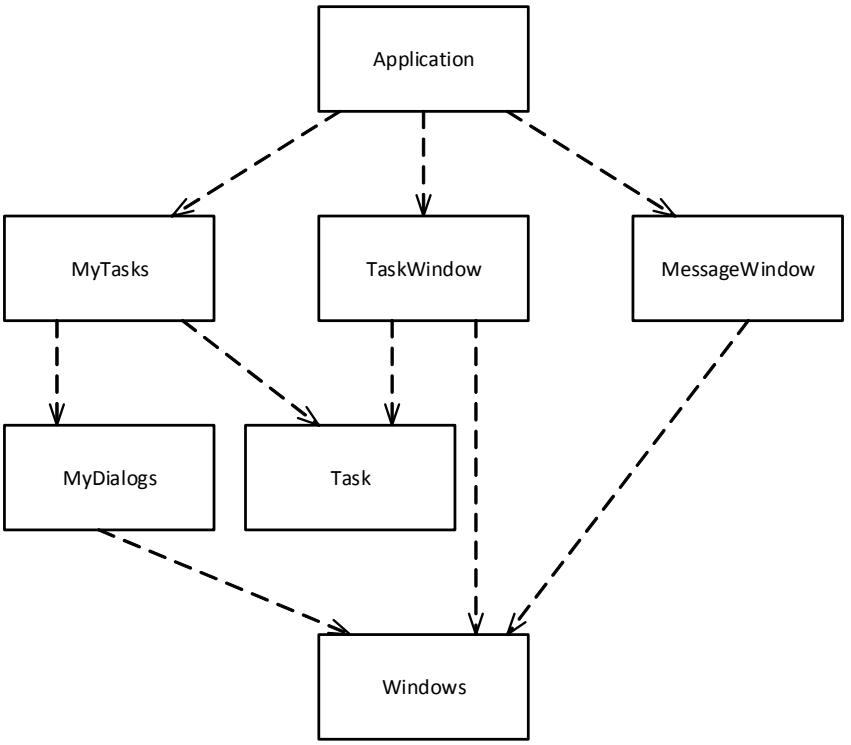


Рис. 3.7
Нарушение принципа ацикличности зависимостей

В этом случае появится зависимость пакета MyDialogs от пакета Application и образуется цикл Application — MyTasks — MyDialogs. Вся цепочка пакетов при этом становится не переиспользуемой иначе как целиком. Неприятным эффектом является то, что циклическую зависимость создает пакет MyDialogs, а переиспользуемость теряет ничего не подозревающий MyTasks. Правильным решением в этом случае будет выделение глобальных данных в отдельный пакет, как это показано на диаграмме (рис. 3.8).

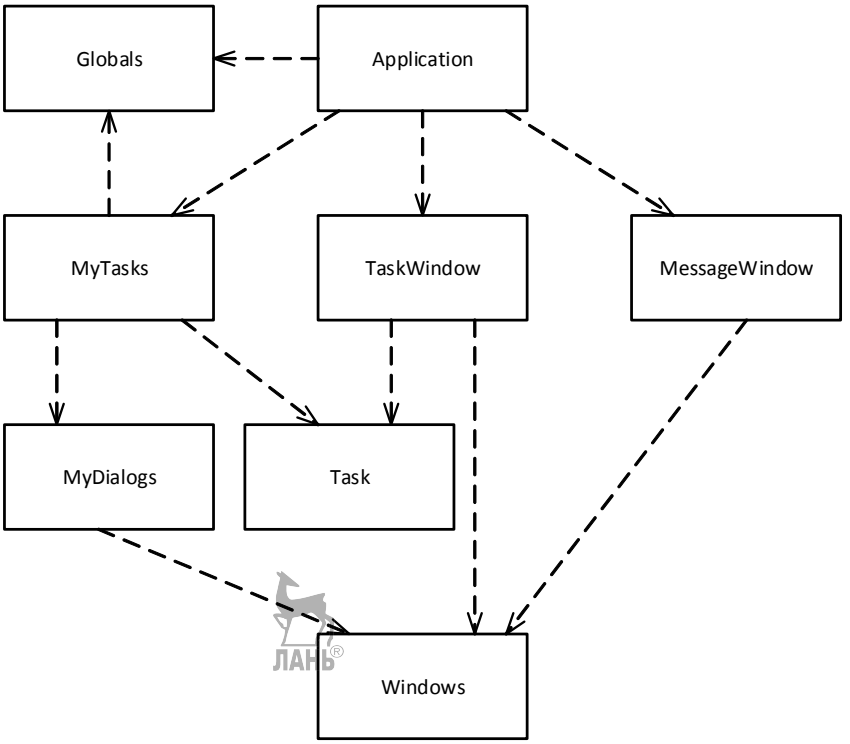


Рис. 3.8
Соблюдение принципа ацикличности зависимостей

Контрольные вопросы

1. Сформулируйте принцип единственности абстракции.
2. Опишите суть принципа инверсии зависимости.
3. Как должны взаимодействовать классы в соответствии с принципом подстановки Лисковой?
4. С какой целью был введен принцип разделения интерфейсов?
5. Какой должна быть структура зависимости между элементами в соответствии с принципом ацикличности зависимостей?

Задачи

Для приведенных формулировок задач выполнить анализ предметной области, выделить абстракции, разработать диаграмму классов, соблюдая принципы объектно-ориентированного проектирования.

1. Каталог ресурсов Интернет

Требуется разработать модель программного обеспечения каталога ресурсов сети Интернет. В каталоге хранится следующая информация о ресурсах: название ресурса, URL, раздел каталога, в котором содержится ресурс, список ключевых слов, краткое описание, дата последнего обновления, контактная информация. Пользователи каталога могут добавлять новые ресурсы, информация о которых не была внесена ранее. Поскольку содержание ресурсов Интернет со

временем изменяется необходимо следить за датой последнего обновления, периодически опрашивая Web-сайты.

Ресурсы в каталоге классифицируются по разделам. Полный список ресурсов каждого раздела должен быть доступен пользователям.

Пользователям каталога должны быть предоставлены возможности по поиску ресурсов. Поиск осуществляется по ключевым словам. Если пользователь недоволен результатами поиска, он может уточнить запрос (осуществить поиск среди результатов предыдущего поиска). Должна быть возможность выдавать результаты поиска в разной форме (вывод всей информации о ресурсах или частичной). Пользователь может отсортировать список ресурсов по релевантности (соответствию ключевым словам из запроса) или по дате обновления.

2. Программное обеспечение микропроцессора табло

Требуется разработать модель программного обеспечения системы отображения информации для станций метрополитена, далее в тексте — табло.

Табло должно отображать текущее время (часы, минуты) и время, прошедшее с момента отправления последнего поезда (минуты, секунды). Дополнительная часть табло должна отображать конечную станцию назначения прибывающего поезда. Предусмотреть также возможность выдачи рекламной информации в «бегущей строке» табло.

Все табло должны быть синхронизованы, т. е. текущее время отсчитывается (и устанавливается) из центральной службы времени.

Каждая станция самостоятельно определяет момент прибытия и отправления поездов при помощи датчиков, устанавливаемых на путях.

3. Система автоматизации для пункта проката видеокассет

Требуется разработать модель системы автоматизации для пункта проката видеокассет. Пункт проката содержит каталог кассет, имеющихся в наличии в данный момент времени. Клиент, обратившийся в пункт, выбирает кассету по каталогу и забирает кассету на определенный срок. Срок проката оговаривается при выдаче кассеты. Клиент возвращает кассету и оплачивает прокат в зависимости от времени возврата и срока проката. Дополнительно нужно предусмотреть возможность принимать заказы от постоянных клиентов на пополнение ассортимента, т. е., если какой-либо кассеты нет в каталоге, постоянный клиент может оставить соответствующую заявку.

Пункт проката должен периодически обновлять ассортимент кассет, предлагаемых для просмотра (с учетом заявок). Пункт должен регулярно сдавать выручку инкассаторам.

4. Программное обеспечение микропроцессора мини-АТС

Требуется разработать модель программного обеспечения встроенного микропроцессора учрежденческой мини-АТС (автоматическая телефонная станция). АТС должна выполнять следующие функции. Абоненты АТС имеют коды 1–9. Код 0 зарезервирован. Каждый абонент подключен к АТС линией связи. АТС соединяет линии абонентов (осуществляет коммутацию линий).

Когда абонент поднимает трубку телефона, АТС получает сигнал «Трубка». В ответ АТС должна послать сигнал «Тон» абоненту. Абонент набирает номер абонента, с которым он хочет связаться. АТС проверяет готовность абонента.

Если абонент готов, АТС посылает ему сигнал «Вызов». При этом телефон абонента начинает звонить. Когда вызываемый абонент снимает трубку, АТС осуществляет коммутацию линии. При этом абоненты обмениваются сигналами «Данные», которые АТС должна передавать от одного абонента к другому. Когда один из абонентов опускает трубку, АТС получает сигнал «Конец» и посылает другому абоненту сигнал «Тон». Если абонент не готов (его линия занята), АТС посылает вызывающему абоненту сигнал «Занято». Когда абонент кладет трубку, АТС получает сигнал «Конец».

5. Программное обеспечение микропроцессора торгового автомата

Требуется разработать модель программного обеспечения встроенного процессора автомата по продаже банок с напитком. Автомат должен выполнять следующие функции.

Автомат имеет отделение для хранения запаса банок с напитком. Автомат должен принимать к оплате бумажные купюры и монеты (установленных образцов). Специальный индикатор должен высвечивать текущую сумму денег, принятых автоматом к оплате. После достижения заданной стоимости клиент должен нажать кнопку и получить банку. Должна быть предусмотрена кнопка «Возврат», при нажатии на которую клиенту возвращаются все принятые от клиента к оплате деньги.

6. Программное обеспечение микропроцессора бытового холодильника

Требуется разработать модель программного обеспечения встроенного микропроцессора бытового холодильника.

Холодильник имеет камеру для хранения продуктов. Доступ в камеру осуществляется через дверцу. На передней панели холодильника имеется регулятор температуры, при помощи которого можно устанавливать максимальную допустимую температуру в холодильной камере.

Холодильник имеет мотор, используемый для понижения температуры в холодильной камере. С камерой связан индикатор температуры, который постоянно выдает текущее значение температуры в камере. При повышении температуры выше определенного предела, определяемого текущим положением регулятора, должен включаться мотор. При снижении температуры ниже некоторого другого значения, связанного с первым, мотор должен отключаться.

Если дверь холодильника открыта в течение слишком долгого времени, должен подаваться звуковой сигнал. Дополнительно звуковой сигнал должен подаваться в любых нештатных ситуациях (например, при невозможности обеспечить требуемую температуру в течение интервала, больше чем некоторое максимальное значение, либо при поломке индикатора температуры).

7. Программное обеспечение встроенного микропроцессора стиральной машины

Требуется разработать программное обеспечение встроенного микропроцессора стиральной машины. Машина должна выполнять следующие функции.

В машине имеется бак для белья, клапаны для забора и слива воды, мотор, дверца для доступа в бак и емкость для моющего средства. Для управления машиной имеется единственная кнопка «Пуск». Для использования машины необходимо открыть дверцу, поместить белье в бак, поместить моющее средство в емкость, закрыть дверцу и нажать кнопку «Пуск». Машина автоматически определяет необходимое количество воды, моющего средства и время стирки.

Машина должна открыть клапан для забора воды, набрать необходимое количество воды, затем закрыть клапан. Далее машина должна впустить необходимое количество моющего средства. Затем в течение установленного времени должен работать мотор. Затем машина должна открыть сливной клапан и выпустить воду.

8. Программное обеспечение микропроцессора телефонного аппарата

Требуется разработать программное обеспечение встроенного микропроцессора для телефонного аппарата учрежденческой мини-АТС. Аппарат подключен к линии связи, ведущей к мини-АТС. Телефонный аппарат должен выполнять следующие функции.

При включении трубки аппарат посылает сигнал «Трубка». При включенной трубке аппарат реагирует на сигналы «Тон» и «Занято» из линии (от АТС).

При получении сигнала «Тон» от АТС аппарат должен воспроизвести звуковой тон «Готов» в трубку. При получении сигнала «Занято» аппарат должен воспроизвести звуковой тон «Занято» в трубку. При отключенной трубке эти сигналы игнорируются.

На трубке имеются цифровые кнопки 0–9. После получения сигнала «Тон» нажатие кнопки с цифрой 1–9 осуществляет вызов абонента с данным кодом. После того как АТС произвела коммутацию линии, два аппарата могут обмениваться данными.

При получении сигнала «Данные» из линии аппарат должен воспроизвести данные, передаваемые с сигналом, в трубку. При получении данных из трубки аппарат должен преобразовать их в сигнал «Данные» и передать их в линию.

После выключения трубки аппарат посылает в линию сигнал «Конец». При получении сигнала «Вызов» из линии аппарат должен произвести звуковой сигнал «Вызов» (независимо от положения трубки).

4. АЛГЕБРАИЧЕСКИЕ ОСНОВЫ ТИПОВ ДАННЫХ

Программа = Алгоритм + Структура данных.
Никлаус Вирт

Тип данных — одно из ключевых понятий современного программирования. Введение этого понятия в программирование обусловлено практикой восприятия действительности человеком. Сталкиваясь в жизни с объектами различной природы, мы привыкли классифицировать их, объединять в одну группу объекты со сходными признаками и применять к ним одни и те же способы обработки. Появление компьютера привело к еще большей формализации такого процесса, представлению его в виде, понятном не только человеку, но и некоторому формальному вычислителю. Влияние человека на типы данных заключается в способах образования понятий и работы с ними. Компьютер — это сложившиеся к настоящему времени шаблоны вычислений и использования вычислительных ресурсов. Понятие типа данных отражает оба эти аспекта. Действительно, моделирование некоторой реальности будет осуществляться проще и естественнее, если инструмент моделирования будет позволять разбивать моделируемые объекты на группы (типы) и манипулировать с объектами каждой группы наиболее адекватными для них средствами. Поэтому нет ничего удивительного в том, что языки программирования, как один из инструментов моделирования реального мира, интенсивно используют понятие типа данных.

Интуитивно ясно, что тип данных характеризует некоторую группу (множество) однородных данных. На начальном этапе развития программирования популярным было определение: тип данных — это множество значений. Более детальное изучение проблемы типов данных навело на мысль: в программе, как и в реальной жизни, при работе с данными с каждым типом связывается некоторый набор операций по созданию и анализу значений данных. Постепенно в трактовке типа данных стал утверждаться принцип конструктивизма: тип данных характеризуется не столько множеством значений, сколько множеством операций. В результате дальнейших исследований появилось операционное определение: тип данных — это множество операций, задающих интерпретацию значений, принадлежащих универсальному для этого типа данных пространству значений. Операции, определяющие тип данных, разбиваются на два класса: конструкторы и анализаторы. Конструкторы, их также называют внутренними операциями, вырабатывают в качестве результата значения определяемого типа. Анализаторы, по-другому — внешние операции, вырабатывают значения, не принадлежащие к определяемому типу (т. е. выдают информацию о значениях типа данных). Каждое значение данного типа является результатом работы последовательности конструкторов (не допускается существование во множестве значений некоторого типа таких значений, которые не вычисляются его операциями). В качестве примера рассмотрим тип натуральных чисел Nat с двумя конструирующими операциями nil и succ и одной анализирующей операцией eqv.

Операция nil является константой, задающей первое значение типа (первичный конструктор). Операция succ использует в качестве аргумента значение данного типа и вырабатывает следующее значение этого типа (комбинированный конструктор). Анализатор eqv сравнивает два значения этого типа и вырабатывает значение логического типа (false или true). Сопоставляя операции с общепринятым изображением натуральных чисел, можно сказать, что знак 0 — это значение, вырабатываемое операцией nil; 1 — значение, вырабатываемое композицией операций succ(nil); 2 — succ(succ(nil)) и т. д. Все возможные последовательности операций типа образуют множество его значений. При этом, как правило, мало кого интересует, что является строительным материалом для значений типа (мы можем знать, конечно, что в компьютере непосредственным строительным материалом для целых или вещественных чисел являются биты, но в типизированных языках мы очень редко используем такое знание). Рассмотрение значений как результатов тех или иных последовательностей конструкторов позволяет утверждать, что каждое значение принадлежит только некоторому определенному типу, операциями которого оно произведено. Формулировка типа данных в таком случае выглядит следующим образом: тип данных определяет множество значений посредством множества операций.

Поскольку при создании типа необходимо указать для пользователя синтаксис и семантику операций, а для компьютера — способ их реализации, любой тип данных строится как комплекс из двух разделов: спецификации и реализации. Спецификация указывает синтаксис (имена, типы операндов и результатов) и семантику (смысл и условия применимости) операций. Реализация указывает представление типа (т. е. данные каких типов являются строительным материалом для значений данного типа) и алгоритмы операций в терминах типов представления. Пользователю типа данных всегда известен первый раздел, второй раздел обычно скрыт от него.

Для того чтобы операции использовались правильно (т. е. чтобы программист понимал, что делает данная операция и в каких условиях она применима, а компьютер мог проконтролировать правильность применения операции), синтаксис и семантика операций должны быть изложены на некотором формальном языке. К настоящему времени предложено несколько формальных методов спецификации программ вообще и типов данных в частности. Наибольшую популярность получил алгебраический подход. В главе представлена алгебраическая теория абстрактных типов данных, которая является математическим обоснованием процессов конструирования пользовательских данных исходя из существующего множества базовых данных. Любая система программирования строится с учетом теории типов данных.

Каждый программист должен хорошо представлять принципы построения и использования типов данных. Рассматривается алгебраический подход, который используется для изучения процессов типообразования в языках программирования.

4.1. Сигнатуры

Пусть имеется множество имен. В типизированных языках программирования принято каждое имя относить к некоторому типу, указывая тем самым, в операциях какого типа это имя можно использовать. Такое связывание имен с типами называется обычно *определением имени*. Предположим, что имеется следующая совокупность определений имен:

bool, nat, setofnat : **тип**;
true, false : **функция** bool;
not : **функция** (bool) bool;
or, and : **функция** (bool, bool) bool;
zero : **функция** nat;
succ : **функция** (nat) nat;
≅, ≤ : **функция** (nat, nat) bool;
empty : **функция** setofnat;
insert : **функция** (setofnat, nat) setofnat;
has : **функция** (setofnat, nat) bool.

На каждой строке этого примера вводится одно или несколько имен (до двоеточия) и специфицируется их тип (после двоеточия). Выделенное слово ***тип*** указывает, что введенные на данной строке имена обозначают какие-то типы данных. Выделенное слово ***функция*** обозначает некоторый тип *функции*, задающей отображение из области определения (аргументов) в область результата. Имена true, false, zero, empty будем называть *константами*. Для единства записи определений имен эти константы будем рассматривать как функции без аргументов. Заметим, что в приведенных определениях указываются только имена типов и ничего не сообщается, что они собой представляют (биты, ячейки, массивы или что-либо еще). Не приводятся также тела функций. Все это на данном (синтаксическом) уровне рассмотрения имен не имеет значения.

В соответствии с терминологией, принятой в математике, будем употреблять термин «основа» вместо «тип» и «знак операции» вместо «имя функции». Тогда можно ввести понятие *сигнатуры* Σ как пары <S, Ω>, состоящей из множества S имен основ и множества Ω знаков операций, для каждого из которых указываются имена основ операндов и основы результата, например:

< {bool, nat, setofnat};
{true, false : → bool;
not : bool → bool;
or, and : bool, bool → bool;
zero : → nat;
succ : nat→ nat;
≅, ≤ : nat, nat→ bool;
empty : → setofnat;
insert : setofnat, nat→ setofnat;
has : setofnat, nat→ bool}> .

Основы и знаки операций выделены в отдельные множества, поэтому необходимость применения терминов «тип» и «функция» отпала. Второй компонент пары можно рассматривать как отображение различных последовательностей имен основ операндов и основы результата на множество знаков операций, что подчеркивается введением знака → *вместо скобок*. Это приводит к понятию I-индексированного семейства множеств. Будем считать, что I-*индексированное семейство множеств* — это функция из I множества. Предположим для примера, что nat обозначает множество {0, 1, 2, …}. Тогда функция, отображающая 0 в {a, b}, 1 в {a, c, d}, 2 в пустое множество и т. д. представляет nat-индексированное семейство множеств букв, изображаемое кратко следующим образом: $\{a, b\}_0$, $\{a, c, d\}_1$, $\varnothing_2$, …. Теперь можно сказать, что второй компонент сигнатуры является по сути дела семейством множеств знаков операций, индексированных своей «арностью», т. е. своей последовательностью имен основ операндов и основы результата. Для рассмотренного ранее примера сигнатуры имеет место: $\{true, false\}_{bool}$, $\{not\}_{bool,bool}$, $\{or, and\}_{bool,bool,bool}$, $\{zero\}_{nat}$, $\{succ\}_{nat,nat}$, $\{\cong, \leq\}_{nat,nat,bool}$, $\{empty\}_{setofnat}$, $\{insert\}_{setofnat,nat,setofnat}$, $\{has\}_{setofnat,nat,bool}$.

Определение 4.1. *Сигнатура* Σ есть пара <S, Ω>, где S — множество (имен основ), $\Omega - (S^* \times S)$ — индексированное семейство множеств (знаков операций), а S^* — множество всех последовательностей элементов множества S, включая пустую последовательность.

Непосредственным следствием данного определения сигнатуры, важным для языков программирования, является то, что в одной сигнатуре (и тем более в разных) допускаются одинаковые имена знаков операций при условии их разной индексации (т. е. разных типов операндов и/или результатов). Например, $\{+\}_{целое,целое,целое}$, $\{+\}_{вещественное,вещественное,вещественное}$, $\{+\}_{строка,строка,строка}$ и т. д.

Далее под обозначением Σ будет пониматься пара <S, Ω>. Для двух сигнатур Σ и Σ′ запись $\Sigma \subseteq \Sigma'$ означает, что $S \subseteq S'$ и $\Omega \subseteq \Omega'$. В этом случае Σ называется *редуктом сигнатуры* Σ′, а Σ′ — *расширением сигнатуры* Σ. Совокупность основ и знаков операций, преобразующих Σ в Σ′, будем называть Σ′ — *приращением сигнатуры* Σ.

Контрольные вопросы

1. Что называется определением имени?
2. Сформулируйте понятие I-индексированного семейства множеств.
3. Сформулируйте формальное определение понятия «сигнатура».
4. Что называется редуктом сигнатуры?
5. Что является расширением сигнатуры?
6. Что понимается под приращением сигнатуры?

4.2. Алгебры

Как было показано в параграфе 4.1, все операции могут быть представлены как функции:

$$T_1, T_2, \ldots, T_n \rightarrow T_p \text{ или } \textbf{функция } (T_1, T_2, \ldots, T_n)\ T_p.$$

В отличие от математической трактовки функций, при которой способ получения результата функции T_p по набору аргументов T_1, T_2, …, T_n лишь подразумевается и не является основным объектом изучения, при программистском рассмотрении функции процесс переработки аргументов-данных в результат выдвигается на передний план. Употребление типов данных в качестве аргументов функций влечет за собой зависимость от них типа результата и/или зависимость типов одних аргументов от других. Поэтому перед нами будут стоять: задача реализации функций и задача анализа функций (рассуждения о функциях). Вследствие этого при указании области определения функции каждый из типов-аргументов при необходимости будем сопровождать соответствующим именем. При таких исходных посылках обозначение типа функции приобретает вид:

функция ($p_1 : T_1, p_2 : T_2, \ldots, p_n : T_n$) T_p ,

где T_i и p_i, i={1, 2, …, n}, — тип, определяющий i-й аргумент (i-е множество значений из области определения), а T_p — тип, определяющий результат функции (множество значений области результата). В дальнейшем в зависимости от потребности будем выбирать любой из перечисленных способов представления функции.

Сигнатуры являются чисто синтаксическим образованием, и от них мало пользы до тех пор, пока мы не придадим смысл применяемым в них именам (т. е. пока не будет задана семантика имен): каждому имени основы мы сопоставим конкретное множество, а каждому имени операции — конкретную функцию. «Осмысление» сигнатуры приводит к алгебре. Задание конкретной алгебры для сигнатуры будем называть *реализацией* сигнатуры. Например, реализация основ и операций нашего примера средствами «компьютерного языка» могла бы выглядеть следующим образом:

реализация основ
bool={**0**, **1**};
nat={***множество комбинаций битов одной ячейки памяти***};
setofnat={***множество ячеек памяти***};
реализация операций
false : ***функция*** bool=**0**;
true : ***функция*** bool=**1**;
not : ***функция*** (p : bool) bool=***если*** **p=0** ***то*** **1** ***иначе*** **0**;
or : ***функция*** (p : bool, q : bool) bool=***если*** **p=0** ***то*** **(*если* q=0 *то* 0 *иначе* 1)** ***иначе*** **1**;
and : ***функция*** (bool, bool) bool bool=***если*** **p=1** ***то*** **(*если* q=1 *то* 1 *иначе* 0)** ***иначе*** **0**;
zero : nat=***заполненная нулями ячейка памяти***;
succ : ***функция*** (n : nat) nat=***машинное сложение*** **(n, 1)**;
≅ : ***функция*** (n1 : nat, n2 : nat) bool=***машинное сравнение*** **(n1, n2)**;
≤ : ***функция*** (n1 : nat,n2 : nat) bool=***машинное вычитание (*n1, n2)**;
empty : setofnat=***пустая последовательность ячеек памяти***;
insert : ***функция*** (s : setofnat, n : nat) setofnat=***если*** **s** ***не содержит*** **n**
то добавить ячейку памяти с **n**;
has : ***функция*** (s : setofnat, n : nat) bool=***если*** **s** ***содержит*** **n** ***то*** **1** ***иначе*** **0**.

Здесь с основой bool связывается множество из двух значений **0** и **1**, одно из которых сопоставляется затем константе false, а другое — константе true. Основе nat сопоставляется множество комбинаций битов ячейки памяти, из которых комбинация с нулями во всех битах выбирается в качестве значения константы zero. Операция succ реализуется посредством команды сложения числа n с единицей, а операции ≅ и ≤ реализуются командами сравнения и вычитания двух чисел, записывающими в определенный регистр признак результата с помощью сопоставляемого основе bool множества значений (1 — плюс, 0 — минус). Основа setofnat связывается с множеством ячеек памяти, а соответствующие операции empty, insert и has реализуются командами над ячейками памяти. Заметим, что в реализации операций как аргументы, так и результаты операций являются объектами сопоставленных основам множеств. Таким образом, можно сказать, что алгебра — это сигнатура с парой функций, одна из которых отображает имена основ во множества, а другая — знаки операций в функции. Если множество S содержит одну основу, алгебра называется *одноосновной*, в противном случае — *многоосновной*.

Ранее мы рассмотрели многоосновную алгебру. Простым примером одноосновной алгебры может служить алгебра целых чисел, взятых по модулю 4 с сигнатурой

<{number}, {zero, one : → number; plus, times : number, number → number}>,

которая может быть реализована следующим образом:

number={0, 1, 2, 3}; zero=0; one=1;

Plus =

	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Times =

	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

У этой алгебры всего одна основа number, представляющая множество из четырех объектов. Реализация операций plus и times задана таблицами: аргументами операции является пара чисел, первое число берется из первого столбца, второе число — из первой строки, на пересечении столбца и строки берется результат операции.

Семейство множеств, сопоставленных именам основ сигнатуры, называется *носителем* алгебры, для алгебры A носитель обычно обозначается как $|A|$.

Определение 4.2. Пусть Σ=<S, Ω> есть сигнатура. Тогда Σ-*алгебра* A — это S-индексированное семейство множеств $|A|$, называемое *носителем* A, вместе с $(S^*\times S)$ — индексированным семейством функций α_{us}: $\Omega_{us} \to \Gamma_{us}$, где us — индекс знака операции, $u \in S^*$, $u = u_1 \dots u_n$, $s \in S$, $\Gamma_{us}=\{|A|_u \to |A|_s\}$ — множество сопоставленным знакам операций Ω_{us} функций реализации, и $|A|_u=|A|_{u1 \dots un} = |A|_{u1}\times\dots\times|A|_{un}$.

Таким образом, каждая α_{us} сопоставляет некоторому знаку операции $\omega_{us} \in \Omega_{us}$ конкретную функцию реализации $\gamma_{us} \in \Gamma_{us}$.

Определение 4.3. Σ-алгебра *A* называется Σ-*редуктом* Σ′-алгебры A′, если $S^A \subseteq S^{A'}$, $\alpha^A_{us}(\omega_{us}) \subseteq \alpha^{A'}_{us}(\omega_{us})$ для всех $s \in S$, $\omega \in \Omega$; соответственно A′ называется *расширением* A. Совокупность основ и операций, добавление которых превращает алгебру A в алгебру A′, будем называть A′ — *приращением* алгебры A.

Рассматривая пример алгебры множеств, можно выделить в качестве редукта алгебру натуральных чисел (с основами nat и bool), сама она является расширением обеих этих алгебр.

Контрольные вопросы

1. Расскажите своими словами, что представляет собой алгебра.
2. Что является носителем алгебры?
3. Чем отличается одноосновная алгебра от многоосновной?
4. Сформулируйте формальное определение алгебры.
5. Что такое редукт, расширение, приращение алгебры?

4.3. Алгебры слов

Для любой сигнатуры Σ существует особая алгебра W_Σ, называемая *алгеброй слов*, или *алгеброй термов*. Ее элементы принято называть *базисными термами*. В соответствии с количеством основ сигнатуры алгебры все множество базисных термов разбивается на подмножества термов каждой из этих основ. Множества базисных термов основы $s \in S$ определяются следующим образом:

— все нуль-арные знаки операций (константы) основы s называются базисными термами основы s;

— для всех знаков операции $\omega \in \Omega_{us}$ и всех базисных термов $t_1, t_2, \ldots, t_n$, принадлежащих, соответственно, основам $s_1 \in S, s_2 \in S, \ldots, s_n \in S$, $\omega(t_1, t_2, \ldots, t_n)$ является базисным термом основы s.

Термы алгебры слов часто называют *выражениями*, а входящие в них знаки операций — *конструкторами выражений*. Для сигнатуры алгебры множеств базисными термами основы setofnat будут empty, insert (empty, zero), insert(insert(empty, zero), succ(zero)), …. В той же сигнатуре базисными термами основы nat будут zero, succ(zero), succ(succ(zero)), …. Для сигнатуры алгебры чисел множество «чисел» алгебры слов будет состоять из zero, one, plus(zero, zero), plus(zero, one), …. Каждое применение операции образует выражение «большей длины» из выражения «меньшей длины». Например, применение операции times к аргументам zero и plus(zero, one) приводит к терму times(zero, plus(zero, one)). Такие выражения можно рассматривать как строки символов, а их конструкторы — как операции над строками, использующими операцию конкатенации строк.

Пусть *X* является S — индексированным множеством, элементы которого называются *переменными*. Тогда можно ввести расширение алгебры слов, с которым постоянно имеют дело программисты, а именно алгебру слов на множестве переменных X_s основы s. Множество элементов этой алгебры составляют термы, использующие данные переменные. Например, если *x* — переменная из множества X_{number}, то наряду с plus(zero, zero) термом алгебры будет и plus(x, zero). И в этой алгебре все термы по-прежнему связываются с определенными

основами, а знаками операций по-прежнему являются конструкторы выражений. Однако в алгебре слов на множестве переменных X множество базисных термов сигнатуры расширено множеством *термов с переменными*. Определение терма с переменными основы s, или просто терма, получается расширением определения базисного терма:

— все базисные термы основы s являются термами основы s;

— все переменные $x \in X_s$ являются термами основы s;

— для всех знаков операции $\omega \in \Omega_{us}$ и всех термов $t_1, t_2, \ldots, t_n$, основы s $\omega(t_1, t_2, \ldots, t_n)$ является термом основы s.

Пусть A есть некоторая Σ-алгебра, а $V=(V_s : X_s \to |A|_s)s \in S$ — семейство функций оценки, сопоставляющих элемент $V_s(x) \in |A|_s$ каждой переменной основы s. Тогда каждый терм алгебры слов может быть «проинтерпретирован», или «вычислен» в этой алгебре. Это означает, что посредством последовательного исполнения функций V_s, осуществляющих привязку переменных к конкретным значениям, и функций γ_{us}, сопоставленным знакам операций ω_{us}, входящим в данный терм, вырабатывается некоторый объект соответствующей основы. Порождение элемента носителя $|A|_s$ для терма t основы s в алгебре A будем называть интерпретацией, или вычислением терма t в алгебре A, и обозначать t^A. Например, пусть имеется алгебра: number={0, 1, 2, 3}, succ : *функция* (n : number) number=n+1 и функции оценки {V(x)=0, V(x)=1, V(x)=2}. Пусть имеется терм t=succ(x). Тогда применение пар операций приведет к следующим интерпретациям

а) V(x)=0 succ(x)=1 б) V(x)=1 succ(x)=2 в) V(x)=2 succ(x)=3.

Контрольные вопросы

1. Что такое алгебра слов и как она конструируется?
2. Чем характерна алгебра на множестве переменных и чем она отличается от обычной алгебры слов?
3. Сформулируйте определение терма алгебры слов.

4.4. Теории

Пусть имеется сигнатура $\Sigma=\langle S, \Omega\rangle$. Тогда сигнатуре Σ можно сопоставить две различные алгебры A и A', с носителями, соответственно, $|A|$, $|A'|$ и функциями сопоставления α_{us}, α'_{us} для каждого символа операции. В общем случае таких алгебр может быть бесчисленное множество. Различные алгебры одной и той же сигнатуры получили название *подобных алгебр*.

При наличии множества подобных алгебр, естественно, возникает необходимость выделения среди них таких алгебр, которые отвечают требованиям полезности, отвечают определенным законам. Эти законы могут быть выражены в виде *равенств*. Равенство на множестве переменных в данной сигнатуре представляет собой не что иное, как пару термов с переменными. Например, равенство plus(x, y)=plus(y, x) на множестве переменных {x, y} может рассматриваться как тройка <{x, y}, plus(x, y), plus(y, x)>.

Определение 4.4. **Σ-*равенство*** — это тройка <X, t_1, t_2>, где *X* есть S — индексированное множество (переменных), а t_1, $t_2 \in |W_\Sigma(X)|$ суть термы одной основы на множестве (переменных) X (обычно уравнение имеет вид: «для всех X, $t_1=t_2$»).

Равенства дают основу для классификации алгебр. Если в алгебре выполняется, скажем, равенство e1(x, y)=e2(x, y), то это означает, что, каковы бы ни были значения, приданные переменным x и y, вычисление выражения левой части дает тот же результат, что и вычисление выражения правой части. Таким образом, если A есть алгебра и v : {x, y} $\rightarrow |A|$ есть функция привязки переменных, тогда требуется, чтобы расширение $v^{\#}$ этой функции подчинялось закону $v^{\#}(e1(x, y))=v^{\#}(e2(x, y))$.

Пусть теперь E будет множеством равенств. Некоторая алгебра называется моделью E, если она удовлетворяет каждому из равенств E. Множество всех моделей будем обозначать E^*.

При наличии некоторого множества моделей мы можем рассматривать множество равенств, выполняющихся в каждой модели этого множества. Обозначим через E^{**} такое множество равенств. E^{**} называется замыканием E.

Рассмотрим пример теории:

- **основы** *{государственное_лицо}*;
- **операции** *{президент, хранитель_ядерного_чемоданчика: государственное_лицо}*;
- **равенства** *{президент=хранитель_ядерного_чемоданчика}*.

Если в государстве есть президент, и если государство обладает ядерным оружием, то всегда можно построить модель, в которой указанное равенство выполняется. Следовательно, в этом случае мы имеем множество моделей E^* и замыкание E^{**} множества равенств.

Рассмотрим еще один пример:

- **основы** *{должностное_лицо}*;
- **операции** *{начальник_отдела, главный_инженер, главный_конструктор: должностное_лицо}*;
- **равенства** *{начальник_отдела=главный_инженер, главный_инженер=главный_конструктор}*.

Предположим теперь, что имеются алгебры с различными значениями для указанных констант (примером такой алгебры может служить малое предприятие с сотрудниками Иванов, Петров, Сидоров и т. д.). В тех алгебрах, в которых *начальник_отдела* и *главный инженер* имеют одно и то же значение, а *главный_инженер* и *главный_конструктор* также имеют одно и то же значение, выполняются приведенные равенства. Но если в алгебре выполняется данная пара равенств, то в ней выполняется и равенство *начальник_отдела=главный_конструктор*.

Если мы выделим из всего множества предприятий такие, в которых предусматривается совмещение рассмотренных должностных обязанностей, то на этом подмножестве мы будем иметь модель E^* и замыкание E^{**}. В общем случае замыкание в этом примере не существует. В этом и заключается смысл введения равенств: получение полезных подмножеств.

Сигнатура с множеством равенств называется *представлением*.

Определение 4.5. *Представление* — это пара <Σ, E>, где Σ есть сигнатура, а E — множество равенств.

Определение 4.6. *Теория* — это такое представление <Σ, E>, в котором E замкнуто.

Для двух теорий T=<Σ, E> и T′=<Σ′, E′> будем обозначать T≤T′, если Σ≤Σ′ и E≤E′. В этом случае T называется *редуктом* теории T′, а T′ — *расширением* T. Совокупность основ, знаков операций и равенств, преобразующих T в T′, будем называть T′ — *приращением* теории *T*.

Следует обратить внимание, что теория и представление являются синтаксическими объектами, в то время как алгебры — это уже семантические объекты. У каждой теории может быть несколько удовлетворяющих ей алгебр.

В качестве примера рассмотрим теорию множества натуральных чисел, соответствующую приведенной в параграфе 4.1 сигнатуре:

основы: < {bool, nat, setofnat};
операции: {true, false : → bool;
not : bool → bool;
or, and : bool, bool → bool;
zero : → nat;
succ : nat→ nat;
≅, ≤ : nat, nat→ bool;
empty : → setofnat;
insert : setofnat, nat→ setofnat;
has : setofnat, nat→ bool}>;
равенства: {not(true)= false;
not(false)=true;
∀ p : bool and(false, p)=false;
and(true, p)=p;
or(false, p)=p;
or(true, p)=true;
∀ m, n : nat zero ≤ n=true;
succ(m) ≤ zero=false;
succ(m) ≤ succ(n)=m ≤ n;
m ≅ n=and(m ≤ n, n ≤ m);
∀ x, y : nat, s : setofnat
has(empty, x) =false;
has(insert(s, x), y)=or(x ≅ y, has(s, y))}.

Алгебрами данной теории будут все алгебры, в которых выполняется каждое из приведенных равенств. Заметим, что вследствие ограничений, накладываемых равенствами, множество этих алгебр будет уже, чем множество алгебр соответствующей сигнатуры. Алгебры некоторой теории часто называют *моделями* этой теории.

Бывают случаи, когда равенств недостаточно для выражения всех нюансов использования сигнатуры и алгебры. В этом случае система равенств расширяется логическими формулами с использованием логических операций «отрицание», «конъюнкция», «дизъюнкция», «импликация». В общепринятой практике формулы такого вида называются *аксиомами*.

Контрольные вопросы

1. Какие алгебры называются подобными?
2. Что такое Σ-равенство и в чем его назначение?
3. Какая алгебра называется моделью множества равенств?
4. Что является замыканием множества равенств?
5. Что называется представлением?
6. Сформулируйте понятие «теория».
7. Что называется моделью теории?

4.5. Абстрактные типы данных

Обсуждая алгебры и теории, мы постоянно соприкасались с двумя уровнями рассмотрения: конкретным и абстрактным. Конкретный уровень представляется конкретной алгеброй с конкретными носителями основ и конкретными функциями, задающими реализации знаков операций. Абстрактный уровень представляется сигнатурой и аксиомами теории.

По отношению к типам данных также можно выделять конкретный и абстрактный аспекты их рассмотрения. Типы данных, предоставляемые конкретным компьютером и конкретной системой программирования, конкретны: для каждого из них транслятор обеспечивает конкретный способ представления значений (в виде последовательностей битов памяти) и конкретные алгоритмы реализации операций. В то же время типы данных, предлагаемые языком программирования, абстрактны. Для каждого из них описание языка задает только синтаксис (имена основ, имена и типы операций) и семантику (смысл операций). В разных трансляторах одному и тому же абстрактному типу данных сопоставляются различные конкретные типы. Программы, полученные разными трансляторами, будут исполняться по-разному, но предъявляемый человеку результат должен быть семантически одинаков.

В самой развитой форме в определение абстрактного типа данных входят следующие четыре части:

1) *внешность*, содержащая имя определяемого типа, имена операций с указанием типов их аргументов и значений;

2) *абстрактное описание* операций и объектов, с которыми они работают, средствами некоторого языка спецификации;

3) *конкретное описание* этих же операций и объектов средствами «обычного» или «известного» языка программирования;

4) *описание связи* между пунктами (2), (3), объясняющее в каком смысле часть (3) корректно представляет часть (2).

Сопоставляя данное определение с уже знакомыми понятиями теории и алгебры, нетрудно видеть, что части (1) и (2) задают некоторую теорию, а части

(3) и (4) — конкретную алгебру этой теории. Таким образом, поскольку имеет место задание конкретной алгебры с конкретными носителями, здесь идет речь о вполне конкретном типе данной теории.

Основное пожелание к абстрактному типу — независимость от реализации. Абстракция в программировании определяется как «отдельная программная конструкция, использование которой связано с отвлечением от деталей ее реализации». Принцип абстракции дает программисту возможность работать с программной конструкцией, зная только синтаксис и семантику ее операций и не вдаваясь в детали реализации. Это, в свою очередь, позволяет определить структуру программного модуля таким образом, что конкретная реализация модуля скрывается от пользователя и может быть изменена, если статистика работы с модулем покажет недостаточную эффективность его реализации. При этом программист не должен ощущать какие-либо изменения в логическом построении типа данных. Рассмотренное различие между абстрактными и конкретными типами данных побуждает связать понятия абстрактного типа данных с понятиями теории, а конкретного типа данных — с понятиями алгебры. Алгебры данной теории, выбираемые для построения конкретных типов данных, будем называть *моделями абстрактного типа данных*.

Контрольные вопросы

1. Сформулируйте понятие «абстрактный тип данных».
2. Сопоставьте понятия «абстрактный тип данных», «теория», «модель теории».

4.6. Иерархические модели типов данных

В основе теории типов лежат две идеи — *типизация универсума* и *типизация языка*. Смысл первой в том, что интуитивно обозримый универсум (пространство значений) должен строиться обозримыми шагами, исходя из (одного или нескольких) базисных типов (классов) объектов, и на каждом шаге построения строительным материалом для новых типов служат типы, полученные на предыдущих шагах. Суть второй идеи в том, что эта иерархия типов должна явно выражаться в языке, на котором формулируются утверждения рассматриваемой теории. Эти идеи интенсивно используются при построении программ средствами типизированных языков программирования: каждый новый тип данных в программах строится на основе некоторой совокупности существующих.

Иерархический тип данных строится на основе одного или нескольких *примитивных* типов, которые, в свою очередь, могут быть иерархическими. Каждый примитивный тип может быть проанализирован и реализован отдельно, без использования информации о тех типах, для построения которых он потребуется. С другой стороны, и вновь созданный тип может рассматриваться как «черный ящик», поведение которого оценивается в терминах примитивных типов. Очень полезное правило: тип данных «не знает», где и как он будет использован.

Рассуждая о типах данных, необходимо различать понятие типа данных как отдельного индивидуума и понятие *системы типов данных* программы или

языка программирования. Дело в том, что, рассматривая алгебры типов данных, мы, по существу, всегда имеем дело с несколькими типами данных, которые можно выстроить в некоторую иерархию. Например, в отмеченной ранее алгебре множеств натуральных чисел явно выделяются типы bool, nat и setofnat с одноименными основами и соответствующими множествами операций. При этом алгебра nat получается расширением алгебры типа bool, а алгебра setofnat — расширением nat. Если теперь при наличии алгебры setofnat мы хотим построить типы integer и setofinteger, то должны взять ее редукт на алгебры bool и nat и расширить его до алгебр integer и setofinteger. В результате получится алгебра, изображенная на рисунке 4.1.

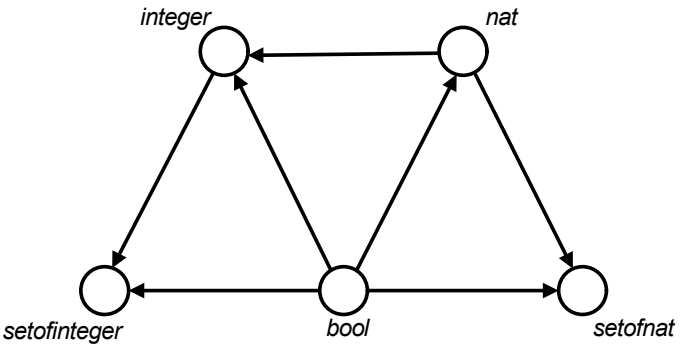


Рис. 4.1
Иерархическая модель построения системы типов данных

На рисунке 4.1 узлы обозначают типы данных, а стрелки — последовательность их построения. Система построенных типов образует алгебру setnatint. Последовательное отбрасывание узлов, из которых не выходит ни одной стрелки, позволяет получить различные ее редукты. Приведенные рассуждения дают основу для более формального сопоставления понятия типа данных с алгебраическими понятиями.

Определение 4.6. *Тип данных* — это либо одноосновная алгебра, либо такое приращение T, преобразующее Σ-алгебру A в Σ′-алгебру A′, которое состоит ровно из одной основы, называемой *конструируемой основой*, и непустого множества операций, в индексах каждой из которых имя конструируемой основы встречается хотя бы один раз. Расширенную алгебру A′, полученную при конструировании типа T, будем называть *алгеброй типа T*.

Определение 4.7. *Сигнатура типа данных* — это либо сигнатура одноосновной алгебры, либо такое приращение Σ'_T, преобразующее сигнатуру Σ в сигнатуру Σ′, которое состоит ровно из одного имени конструируемой основы и непустого множества знаков операций, в индексах каждого знака операции это имя основы встречается хотя бы один раз.

Определение 4.8. *Спецификация типа данных*, или *абстрактный тип данных*, — это сигнатура типа данных и множество аксиом, в каждой из которых хотя бы один раз встречается знак операции из данной сигнатуры. Другими словами, спецификация типа данных — это приращение теории, подчиняющее-

ся определенным ограничениям. Расширенную теорию, полученную при конструировании спецификации типа T, будем называть *теорией типа* T.

Таким образом, понятие абстрактного типа данных отождествляется с более знакомым программистам понятием спецификации (определения) типа данных, которое, в свою очередь, рассматривается как определенное расширение теории. На выбор модели теории типа данных (т. е. конкретной алгебры) не накладывается никаких ограничений. Главное — алгебра должна удовлетворять аксиомам теории и соответствовать той модели, которую разработчик имел в виду.

Пользователя типа данных, как правило, интересует не алгебра типа данных, а лишь его спецификация, знания которой вполне достаточно для корректного употребления операций типа. Поэтому на практике тип данных представляется как комплекс, состоящий из двух разделов:

— открытой пользователю спецификации типа;

— скрытой от пользователя реализации типа (т. е. соответствующего приращения алгебры).

Удобно считать, что каждому конструируемому типу сопоставляется имя, совпадающее с именем его основы. Тогда можно говорить, что знаки операций типа индексируются именами типов. Типы из расширяемой алгебры, встречающиеся в индексах операций данного типа, называются *примитивными типами*.

В качестве примера рассмотрим построение теории множеств натуральных чисел в виде иерархической последовательности абстрактных типов данных. При этом, хотя имена примитивных типов однозначно извлекаются из индексов операций, для более четкой индикации расширяемой теории они указываются явным образом в начале конструкции.

тип bool =
операции: {true, false : → bool;
not : bool → bool;
or, and : bool, bool → bool};
аксиомы {true ≠ false;
not(true)= false;
not(false)=true;
∀p : bool and(false, p)=false;
and(true, p)=p;
or(false, p)=p;
or(true, p)=true}.

Моделями данного абстрактного типа данных (т. е. данной теории одноосновных алгебр) могут быть любые одноосновные алгебры, носители которых содержат два различных элемента, образуемые операциями false и true.

тип nat =
примитивный bool;
операции { zero : → nat;
succ : nat→ nat;
≅, ≤ : nat, nat→ bool};

аксиомы {∀m, n *:* nat zero ≤n=true*;*
succ(m) ≤ zero=false;
succ(m) ≠ zero;
succ(m) ≤ succ(n)=m ≤ n;
m ≅ n=and(m ≤ n, n ≤ m)}.

Как видно из спецификации, данный абстрактный тип расширяет теорию одноосновных алгебр bool до теории двухосновных алгебр, скажем, boolnat. Нетрудно убедиться, что множество значений типа nat может представлять собой как бесконечное множество вида {0, 1, 2, …}, так и любое конечное множество вида {0, 1, 2, …n}.

тип setofnat =
примитивные {bool, nat};
операции {empty : → setofnat;
insert : setofnat, nat→ setofnat;
has : setofnat, nat→ bool};
аксиомы {∀x, y : nat, ∀s *:* setofnat
has(empty, x) =false;
has(insert(s, x), y)=or(x ≅ y, has(s, y))}.

Абстрактный тип setofnat расширяет теорию boolnat до теории, скажем, boolnatset. Алгебры, соответствующие этой теории, могут строиться расширением алгебр теории boolnat.

В заключение обратим внимание на то, что реализация одноосновных алгебр всегда относится к компетенции разработчиков системы программирования, а реализация абстрактного типа данных, строящегося как приращение существующей алгебры, часто является прерогативой программиста.

Контрольные вопросы

1. Что такое «иерархический тип данных»?
2. Чем отличается иерархический тип данных от примитивного?
3. Опишите свойства примитивного типа данных.
4. Расскажите своими словами, как строится иерархический тип данных.
5. Что представляет собой спецификация типа данных?

4.7. Типы и подтипы

Предположим, что имеются два типа T и П и две функции into*:* T→П и out*:* П→T с таким свойством, что ∀*x* : П имеет место into(out(x))=x. В этом случае говорят, что тип П является *подтипом* типа T, а тип T — образующим типом. Термин «подтип» означает, что область значений функции out представляет собой подмножество множества значений образующего типа и, соответственно, функция into определена только на этом подмножестве (множество значений подтипа является подмножеством значений образующего типа).

В общем случае типы T и П могут иметь собственные множества операций. Однако поскольку каждое значение подтипа может быть преобразовано в значение образующего типа и затем обработано посредством его операций, на практике подтипы обычно рассматриваются лишь как средство наложения ограничений на множество значений образующего типа. По этой причине мы будем считать, что при определении подтипа задаются только операции into и out. Аксиомы подтипа, очевидно, в таком случае специфицируют области определения и значения этих функций.

В качестве примера рассмотрим образование подтипа oneten, выделяющего в типе nat отрезок значений от 1 до 10. Соответствующее расширение теории типа nat будет выглядеть следующим образом:

тип oneten =
примитивные nat
операции {into : nat → oneten;
out : oneten → nat};
аксиомы {∀x : oneten into(out(x))=x,
out(x) ≤ 10 & out(x) ≥ 1=true}.

Первая из аксиом показывает, что речь идет об образовании подтипа, а вторая задает область определения функции into, т. е. критерий выделения значений типа для образования подтипа. Поскольку каждый подтип обладает ровно двумя операциями into и out с очевидными типами аргументов, их явное указание становится излишним, излишне также указание первой аксиомы, свойственной всем подтипам, и поэтому можно записать коротко: **подтип** oneten=nat(x : nat, 1 ≤ x ≤ 10).

Употребления значения подтипа в контекстах, в которых требуется значение образующего типа, всегда означает неявное использование операции out, а употребление значения образующего типа в контекстах, в которых требуется значение подтипа, всегда означает неявное использование операции into.

Контрольные вопросы

1. Сформулируйте понятие «подтип данных».
2. Какие аксиомы являются обязательными при определении подтипа данных?
3. Почему определение подтипа можно записать в краткой форме?

4.8. Типы типов

Формализация способа построения типов данных приводит к тому, что каждый тип может рассматриваться как отдельный объект, создаваемый специальной операцией конструирования типов. Это, в свою очередь, позволяет говорить о множестве типов, создаваемых указанной операцией, и тем самым — о типе всех типов данных **тип**.

Так же как и в отношении других типов данных, в типе данных **тип** можно выделить по тем или иным критериям различные подтипы. Каждый *подтип типов*,

или *ограниченный тип типов*, определяет множество типов данных, обладающих каким-либо общим свойством. Распространенный класс подтипов типов составляют такие подтипы, каждый из которых накладывает условие, что тип данных, входящий в его подмножество, должен содержать среди своих операций указанный список операций. Примерами могут быть подмножество типов данных, обладающих операциями сравнения на равенство; подмножество типов данных, обладающих операциями сложения и вычитания и т. д. Тип данных может иметь и другие операции, благодаря которым он может быть отнесен к другим классам.

Если β — некоторая спецификация операций типа данных, то будем обозначать **тип**(β) множество типов данных, каждый из которых содержит в своей спецификации операции и аксиомы из β. Чем уже β, тем мощнее **тип**(β). В предельном случае, когда β пусто, имеет место **тип**( )=**тип**, т. е. охватывается все множество типов (никаких ограничений). Если считать, что тип данных **тип** в своем множестве операций содержит операцию spec, выделяющую у данного типа его спецификацию, то согласно общему определению подтипа (см. п. 4.7) спецификация подтипа типов данных **тип**(β) должна иметь следующий вид:

тип *mun*(β) =
примитивные тип
операции {into : **тип** $\rightarrow$ *mun*(β*);*
out : *mun*(β) $\rightarrow$ **тип**};
аксиомы {$\forall$x : *mun*(β) into(out(x))=x,
$\beta \subset$ spec(out(x)) =true}.

Однако, как и в случае подтипов «обыкновенных» типов данных, для подтипов типов также употребление операций into*(*x*)*, out*(*x*)* и аксиомы into(out(x))=x подразумеваются неявно, а в определение подтипа подставляется лишь критерий выделения значений. Тогда краткая запись предыдущего определения будет иметь вид:

подтип *mun*(β)=**тип**($\forall$x : *mun*(β) $\beta \subset$ spec(out(x)) =true).

Рассмотрим теперь пример определения подтипа:

подтип *типсравно*= **тип**(**операции** $\cong$: *типсравно , типсравно* $\rightarrow$ bool
аксиомы $\forall$ x, y, z : *типсравно*
x $\cong$ x=true;
x $\cong$ y=y $\cong$ x;
and(x $\cong$ y, (y $\cong$ z))=(x $\cong$ z)).

Согласно определению подтипа типов **тип**(β) каждый тип со спецификацией β_k принадлежит (вернее, может быть преобразован к) подтипу типов **тип**(β_k) и каждому **тип**(β_l), если $\beta_l \subset \beta_k$.

Контрольные вопросы

1. Расскажите своими словами о понятии «тип всех типов данных».
2. Какой прием применяется при определении типа данных при наличии понятия «тип всех типов данных»?

4.9. Родовые типы данных

Согласно определению конкретного типа данных, приведенному в параграфе 4.5, операция конструирования типа данных в самой развитой форме должна содержать четыре аргумента: внешность, абстрактное описание, конкретное описание и описание связи. Первые два из них составляют спецификацию типа (приращение теории или сигнатуры системы типов в простейшем случае), а последние два — реализацию типа (приращение алгебры системы типов). Для дальнейших рассуждений операцию конструирования типов будем изображать в сокращенном виде с двумя аргументами — спецификацией и реализацией; саму операцию обозначим служебным словом **модуль**[12], за которым в квадратных скобках следует первый аргумент (спецификация типа) и затем в скобках **начало** … **конец —** второй аргумент:

модуль [*спецификация типа*] **начало** *реализация типа* **конец**.

Список операций в спецификации типа представляется в виде $r_1 : T_1$, $r_2 : T_2$, …, $r_n : T_n$, где r_i — имя, а T_i — тип (индекс) i-й операции.

Каждый из типов (индексов) операций T_i содержит обозначение конструируемого типа как типа одного или нескольких аргументов и/или типа результата. Ранее конструируемый тип в типах операций обозначался конкретным именем. Со сконструированным типовым объектом мы можем в принципе связать любое внешнее имя (например, мы можем описать типовою переменную и затем присваивать ей различные типовые значения). С другой стороны, во многих языках программирования существует прием, когда создаваемый тип данных никак не именуется. Поэтому требуется способ обозначения конструируемого типа внутри себя. Будем использовать для этой цели символ «*» (звездочка) и считать, что в каждом типе звездочка в его спецификации и реализации означает «я сам».

В соответствии с этими замечаниями конструкция определения абстрактного типа nat приобретает вид

тип nat=**модуль** [**операции** zero : $\rightarrow$ *;
succ : * $\rightarrow$ *;
$\cong, \leq$: * , * $\rightarrow$ bool
аксиомы $\forall$ m, n : *
zero $\leq$ n=true;
succ(m) $\leq$ zero=false;
succ(m) $\neq$ zero=true;
succ(m) $\leq$ succ(n)=m $\leq$ n;
m $\cong$ n=and(m $\leq$ n, n $\leq$ m)]
начало *реализация типа* **конец**.

Если предположить, что некоторые идентификаторы типов, используемые в спецификации и реализации конструируемого типа, являются свободными переменными, то тип становится функцией от значений этих переменных. Такая функция вырабатывает типы, характеризуемые одним и тем же множест-

[12] Для обозначения этой операции используются также термины «кластер», «класс».

вом операций с различными типами некоторых операндов и/или результатов. Подобную функцию можно назвать *типовой функцией* и изобразить следующим образом:

T_r=***функция*** (α) ***тип*** **; модуль** [β] **начало** *реализация типа* **конец,**

где $\alpha=\{p_1 : T_1, p_2 : T_2, \ldots, p_n : T_n,\}$ — спецификация параметров функции, β — спецификация операций типа.

Подобные типовые функции являются *генераторами*, или *схемами* типов. В языках программирования их принято называть *параметризованными*, или *родовыми* типами, или *шаблонами*, и изображать в упрощенном виде:

T_r=**модуль** (α) [β] **начало** *реализация типа* **конец.**

Порождение конкретного типа, заданного родовым типом, осуществляется операцией конкретизации, являющейся обычной операцией применения функции к своим аргументам:

$$Tk=Tr\ (q_1, q_2, \ldots, q_n).$$

В том случае, когда в α не зафиксировано значение ни одного формального параметра, тип, представленный конструктором T_r, называется *исходным родовым типом*, объединяющим некоторое максимальное множество типов данных со сходным поведением. Обращение к конструктору с фиксацией части параметров приводит к образованию конструктора редуцированного типа, представляющего частный родовой тип, являющийся потомком исходного множества типов данных (т. е. область значений этой функции является подмножеством исходной области значений). Фиксация всех параметров приводит к выработке конкретного типа данных с операциями из семейства T_r.

В качестве примера рассмотрим спецификацию родового типа множества:

тип set= **модуль** (elem : **тип**
< операции $\cong$: * , * $\rightarrow$ bool
аксиомы $\forall$ x, y, z : *
x $\cong$ x=true;
x $\cong$ y=y $\cong$ x;
and((x $\cong$ y), (y $\cong$ z))=(x $\cong$ z)>)
[**операции** empty : $\rightarrow$ * ;
insert : * , elem $\rightarrow$ * ;
has : * , elem $\rightarrow$ bool};
аксиомы $\forall$ s : *, $\forall$ x, y : elem,
has(empty, x) =false;
has(insert(s, x), y)=or(x $\cong$ y, has(s, y))]
начало *реализация типа* nat **конец**.

Заметим, что символ «*» в спецификации elem (в круглых скобках) обозначает тип этого elem, а в спецификации (в квадратных скобках) и в реализации (в скобках **начало** …**конец**) определяемого типа обозначает определяемый тип (set).

Аргументом данного родового типа может быть любой тип, имеющий в своем составе операцию сравнения на равенство, что постулируется типом па-

раметра elem. Необходимость этой операции вызвана тем, что она используется в спецификации типа set. Конкретизация введенного родового типа

подтип setofnat=set(nat)
порождает тип с множеством операций

empty : → setofnat ;
insert : setofnat , nat → setofnat ;
has : setofnat , nat → bool;

а конкретизация

подтип setofinteger=set(integer)
порождает тип с множеством операций

empty : → setofinteger ;
insert : setofinteger , integer → setofinteger ;
has : setofinteger , integer → bool.

Механизм непосредственного построения конкретных типов данных обеспечивает пошаговое расширение алгебры системы типов. Механизм родовых типов позволяет осуществить обобщение этого процесса в тех случаях, когда необходимо построить ряд «родственных» типов данных. Естественно, что каждый конкретный тип из семейства родового типа мог бы быть построен непосредственным образом.

Контрольные вопросы

1. Сформулируйте понятие «родовой тип данных».
2. Какой родовой тип данных называется исходным?
3. В чем полезность родового типа данных?
4. Почему параметризованный тип данных принято называть родовым?

4.10. Полиморфные операции

Возможность использования типа **тип** и родовых типов в спецификации параметров операций приводит к появлению нового качества — возможности составлять функции и процедуры, обрабатывающие данные разных типов. Такие функции и процедуры принято называть *родовыми*, или *полиморфными*. Поскольку, как было показано в предыдущем параграфе, описание параметра посредством родового типа служит на самом деле сокращением описания ряда типовых параметров, то основным признаком полиморфизма следует считать именно наличие понятия типа типов данных.

Существенным свойством полиморфной функции (процедуры) является наличие только одного тела функции (процедуры), независимо от сочетаний типов параметров. Строго говоря, следует считать ошибкой встречающееся иногда отождествление полиморфных функций с функциями, имеющие одинаковые имена. Последнее свойство называется совмещением имен и обеспечивается возможностью использования одних и тех же знаков операций при условии их разной индексации. Однако нужно заметить, что и в этом случае часто используется термин «полиморфизм». Распространенными примерами совмещения имен операций в языках программирования являются знаки операций

«+», «–», «=» и т. д., каждый из которых обозначает несколько операций, принадлежащих одному и тому же или разным типам данных. Например, знак операции «+» может изображать следующие операции:

+ : *целый, целый → целый*;

+ : *целый, вещественный → вещественный*;

+ : *вещественный, целый → вещественный*;

+ : *вещественный, вещественный → вещественный*;

+ : *строка, строка → строка*;

+ : *множество, множество → множество, и т. п.*

Хотя все эти операции имеют общие свойства (например, коммутативность), их не всегда можно реализовать посредством одной полиморфной функции (а при нашем подходе и невозможно, потому что первый знак операции «+» реализуется в типе данных «целое», следующие три — в типе данных «вещественное», последний — в типе данных «строка»).

В языках программирования первого поколения (*Фортран, Алгол-60*) возможность совмещения имен обеспечивалась только для знаков операций стандартных типов данных. В последующих языках (*ПЛ/1, Алгол-68*) эта возможность была распространена как на знаки операций конструируемых типов, так и на имена функций вообще.

Переход к использованию полиморфных операций по времени совпал с появлением объектно-ориентированного программирования. В качестве простого примера полиморфной операции рассмотрим функцию, вычисляющую в своем составе квадрат аргумента любого типа, имеющего в своем составе операцию умножения:

***квадрат*: функция (Т: тип (×: функция (Т; Т) Т), а: Т) Т начало а×а конец.**

При наличии описания переменных

перем *х: целое; у: вещественное*;

допустимы обращения к этой функции:

квадрат(*целое, х***), квадрат(***вещественное, у***).**

В первом случае функция *квадрат* выполнит целочисленное умножение, во-втором случае — умножение вещественных чисел. Подобным же образом допустимы все обращения к этой функции с параметром любого типа, обладающим операцией умножения.

Поясним смысл описания функции. Во-первых, параметром функции является тип данных. Во-вторых, являющийся параметром полиморфной функции тип данных должен содержать операцию умножения. В-третьих, при вызове функции операция умножения должна быть доступна для использования.

Наиболее существенной проблемой, связанной с полиморфными функциями, является способ компиляции (реализации) таких функций. Здесь вопрос ставится следующим образом: должно ли при компиляции программы сопоставляться полиморфной функции (процедуре) одно тело или для каждого сочетания типов аргументов должно компилироваться отдельное тело? Если типы делают доступным множество основных операций внутри процедуры, абст-

рактной по отношению к типу T, просто невозможен прямой доступ к операциям, связанным с типом T. Вместо этого должен обеспечиваться косвенный доступ к ним для каждого типа — аргумента. Реализация косвенного доступа к операциям аргумента полиморфной операции рассмотрена в главе 9.

Контрольные вопросы

1. Какие процедуры и функции следует считать полиморфными?
2. Чем отличаются полиморфные функции от функций, имеющих одинаковые имена?

4.11. Программы как данные

Согласно определению, каждая теория задает некоторый формальный язык, для которого алгебра слов определяет множество возможных предложений, а система аксиом — семантику, или смысл правильных предложений. Частным случаем формального языка является алгоритмический язык, или язык программирования. Каждая программа на таком языке представляет собой терм (с переменными) из данной теории, в котором традиционно выделяют две части: программу и данные. В то же время, если по отношению к данным давно применяется понятие типа, операции и т. п., то программа как объект обработки чаще всего представляется лишь в виде композиции лексических и синтаксических единиц, рассматриваемые как «операционные элементы», которые должны быть в конечном счете отображены в машинную программу.

Рассматривая систему типов языка программирования, необходимо понимать, что та часть типов, с которой имеет дело программист, использующий данный язык программирования, составляет только видимую часть айсберга системы типов. Она базируется на невидимой части — типах, необходимых для организации вычислений в целом (в частности, трансляции), но не представляемых языком программирования.

Если же говорить о системе типов языка программирования в целом, включая в нее пользовательские типы и типы, фигурирующие на уровне трансляции, то становится возможным указать родовые типы записей (структур) и объединений, которые конкретизируются при вычислении программы как данного. На самом деле известно, что каждый тип записи (структуры) характеризуется в языке программирования одним и тем же множеством операций (создание записи, выборка поля записи, сравнение записей). Более того, каждый, кто создавал транслятор языка программирования с типами записей, хорошо знает, что для обработки определений всех типов записей имеется одна процедура, воспринимающая в качестве параметров описания полей записи и конструирующая на их основе конкретный тип записи. Вычисления этой процедуры связаны с объектами таких «необычных» типов, как *описание имени*, *последовательность описаний имен*, которые выступают в качестве фактических параметров, конкретизирующих вырабатываемый результат. Осознание данного факта приводит к необходимости распространения понятий типов данных на все конструкции языка программирования. Такой взгляд на систему типов позволяет, в частности, считать средства задания типов записей и объе-

динений проекцией средств цельной системы типов на пользовательскую часть.

Скажем несколько слов о форме языка программирования. Поскольку с любым языком имеют дело пользователи двух категорий — разработчики системы программирования и программирующие на этом языке, то и описание языка должно иметь две формы: конструкторскую и потребительскую. Конструкторская форма должна быть предельно формализованной и максимально способствовать созданию корректной версии системы программирования. Потребительская форма должна быть объектом «дизайна», т. е. учитывать эстетические и психологические свойства людей, сложившиеся приемы программирования и т. п. Конструкторская форма больше ориентирована на автомат, чем на человека, а пользовательская — более «дружественная» к человеку. Необходимо лишь, чтобы между конструкторской и потребительской формами языка существовало строгое соответствие, так, чтобы потребительская форма с помощью определенных операций была бы сводима к конструкторской.

Для получения пользовательской формы языка программирования используются главным образом следующие приемы.

1. Опускание имен функций. В тех случаях, когда из контекста известно, о каких функциях идет речь, явно указываются только аргументы этих функций. В качестве примера можно привести функцию преобразования целого числа в вещественное, которая всегда используется неявно.

2. Использование специальных ограничителей. В стандартной форме обращения к функции ее аргументы заключаются в круглые скобки и разделяются запятыми. Для большинства стандартных функций языка программирования употребляется нестандартная форма обращения: вместо круглых скобок могут использоваться квадратные скобки или вообще стандартные символы (например, **начало** и **конец**), вместо разделительных запятых могут использоваться точки с запятой, точки или также специальные символы (например, **то**, **иначе**, **до**, **из** и т. д.). Именно благодаря специальным ограничителям чаще всего создается контекст, позволяющий опускать имена функций.

3. Неявная подстановка аргументов. Допускаются обращения к функциям с заданием части аргументов, в качестве остальных аргументов система программирования подставляет некоторые стандартные значения (например, в условном операторе может отсутствовать часть **иначе**).

Далее мы будем придерживаться в основном конструкторской формы, иллюстрируя ее возможными образами потребительской формы.

Контрольные вопросы

1. Какие формы должно иметь описание языка программирования?

4.12. Типовая полнота языка программирования

Понятие типовой полноты языка программирования связано с выполнением следующих условий:

1. Каждое имя (идентификатор или знак операции) и каждое выражение в языке имеют тип; тип любого сложного выражения образуется из типов его компонентов. Тип имени или выражения не зависит от контекста, в котором они используются (например, тип идентификатора не зависит оттого, используется ли он в левой или правой части оператора присваивания).

2. Для каждого типа данного языка программирования можно записать выражение, имеющее этот тип.

3. Любое выражение может быть параметризовано по отношению к любому из своих операндов с образованием функции от этого операнда. Отсюда следует, что функции могут иметь параметры любых типов и вырабатывать результаты любых типов.

4. Каждая конструкция языка является функцией (операцией) некоторого типа, и, соответственно, программа представляет собой композицию функций различных типов данных.

5. Типы данных являются полноправными объектами языка программирования, вследствие чего нет резкой границы между значениями и типами.

6. Любой тип данных представим посредством универсального конструктива типов данных, который доступен программисту для определения произвольных типов данных.

Благодаря указанным свойствам, мы можем рассматривать программу как некоторый терм алгебры слов, еще более сократить число базовых понятий языка программирования и обеспечить возможность единообразной трактовки стандартных и производных типов. Так, например, представление типов как значений дает возможность построения родовых модулей программ (функций, процедур, типов), при котором родовые типы рассматриваются как частные случаи типовых функций (т. е. функций, вырабатывающих типы). Вынесение операторов на уровень типов данных дает возможность трактовать процедуры как частные случаи функций (функции, вырабатывающие значение типа «оператор»). Из того, что тип является значением, следует, что определение типа данных есть частный случай определения константы. Например, возможность использовать для спецификации параметров функции родовые типы данных определяет единый взгляд на статические и динамические массивы. Объявив параметр функции родовым массивом, т. е. массивом, у которого не зафиксирован тип индекса, можно подставлять в качестве аргумента любой статический массив их компонентов того же типа.

Использование типовой полноты облегчает обучение языку программирования: изучать общий взгляд, а затем проецировать его на конкретные разработки и проще, и полезнее, чем синтезировать общую картину из разрозненной коллекции конкретных знаний.

Контрольные вопросы

1. Какие условия определяют типовую полноту языка программирования?

4.13. Стандартные и произвольные типы

В описании языков программирования обычно выделяют три категории типов данных: стандартные, конструируемые и абстрактные. К стандартным типам принято относить общепризнанные типы без параметров, например, целый или вещественный типы. К конструируемым типам относят общепризнанные типы с параметрами, например, отрезки, записи, массивы. К абстрактным относятся типы, создаваемые в данной программе специальным конструктором типов.

С точки зрения пользователя каждый тип данных надо считать абстрактным, так как не все программисты знают детали реализации даже стандартных типов. С другой стороны, при унифицированной трактовке типов данных все они представляются посредством одной и той же программной конструкции — унифицированным конструктором типов данных. По этим причинам правильнее делить все типы данных на два класса: стандартные и произвольные.

Стандартные типы характеризуют язык программирования, их спецификация осуществляется разработчиком языка, а реализация — разработчиками системы программирования. Произвольные типы (вместе со стандартными) характеризуют конкретную программу, их спецификацию и реализацию осуществляют пользователи средствами данного языка программирования, естественно, язык должен предоставлять средства такого типообразования. Можно сказать, что механизм произвольных типов данных является средством расширения языка, обеспечивая его специализацию для более удобного решения конкретной задачи автоматизации вычислений.

Как стандартные, так и произвольные типы делятся, в свою очередь, на две категории: конкретные и родовые (параметризованные) типы. Конкретный тип характеризует определенное множество значений, он не требует дальнейшего уточнения и готов к непосредственному употреблению. Родовой тип характеризует множество множеств значений и требует дальнейшего уточнения путем подстановки аргументов и получения конкретного множества значений. Переменная такого типа не может быть полностью обработана до завершения конкретизации типа.

Примеры конкретных типов: целый, литерный, отрезок от 1 до 10, массив из 100 целых, стек из 200 вещественных. Примеры родовых типов: отрезок, запись (структура), массив, стек.

Контрольные вопросы

1. Кто осуществляет спецификацию стандартных типов данных?
2. Кто осуществляет реализацию стандартных типов данных?
3. Кто осуществляет спецификацию произвольных типов данных?
4. Кто осуществляет реализацию произвольных типов данных?
5. На какие категории делятся стандартные и произвольные типы данных?

4.14. Примеры спецификации типов данных

Полная спецификация языка программирования должна определять иерархическую систему типов данных. В каноническом виде основанием иерархии является тип данных «бит». На более высоких уровнях иерархии располо-

жатся типы данных «целый», «вещественный», «логический» и т. д. На самых верхних уровнях иерархии будут находиться типы данных «оператор», «программа». Рассмотренный подход позволяет считать операторы и программу в целом типами данных.

В настоящем параграфе рассмотрены отдельные примеры спецификации типов данных. Мы сосредоточим внимание на стандартных типах языка программирования, однако все методы, используемые здесь, одинаково применимы и к произвольным типам. Сохраняется и выработанная ранее нотация: в круглых скобках указываются импортные параметры типа, в квадратных — экспортные атрибуты (операции), сам определяемый тип в пределах конструктива обозначается звездочкой (*).

При описании типов мы ограничимся лишь указанием имен и типов атрибутов, т. е. сигнатуры типа. Эту информацию мы называем спецификацией. Содержательный смысл определений раскрывается словесными пояснениями. Не приводится сведений о их реализациях, поскольку они в большинстве случаев неоднозначны. Оставаясь на чисто формальных позициях, можно считать, что любая реализация типа удовлетворяет приводимым спецификациям, если для каждого атрибута она представляет вычисления с аргументами указанных типов, которые приводят к результату указанного типа.

В приводимых спецификациях выделена только та информация, которая необходима для синтаксически правильной трансляции программы из элементов, удовлетворяющих требованиям соответствия типов. Эта информация вполне может быть проверена в ходе трансляции, рассматриваемой как часть вычисления пары «программа, данные». Если язык не ограничивает вычисления над типами статическим случаем, то можно считать, что часть трансляции, связанная с типовыми вычислениями, переносится в динамику вычисления программы. В соответствии с такой точкой зрения можно определить, какие процессы должны выполняться при трансляции, не прибегая к понятиям статических и динамических вычислений. Объектами оперирования при трансляции являются данные стандартных типов, представляющих конструкторы языка, и типов, фигурирующих в вычисляемой программе. Отнесение же трансляторных вычислений к статическому или динамическому уровню принадлежит реализационной стороне концептуально-целостного процесса трансляции.

Спецификация типов, используемая в данном параграфе, уже спецификаций, приводившихся в предыдущих параграфах данной главы, которые отражают семантическую сторону вычислений (т. е. описания соотношений между перерабатываемыми данными). Здесь предполагается, что семантический аспект вычислений задается реализацией. Приведенные пояснения используемого понятия спецификации позволяют охарактеризовать ее как трансляторную спецификацию.

В конце параграфа приведен список задач на построение спецификаций типов данных. В задачах можно использовать приведенные в настоящем параграфе спецификации.

Литеры. Литеры являются теми элементарными кирпичиками, из которых строится текст программы. Исчерпывающий набор литер представляет алфавит языка программирования. Будем считать, что каждая литера создается

отдельной операцией, имя которой совпадает с изображением литеры. В этом случае спецификация типа «литера» будет выглядеть следующим образом:

тип *литера*=**модуль**

[0, 1,2, …9, A, B, …Z, A, Б, …, Я, a, b, …, z, а, б, …, я, +, -, … : *].

Этот тип является внешним, или исходным, для любого языка программирования в том смысле, что язык не предоставляет никаких операций по работе с литерами, за исключением построения из литер более крупных конструкций — строк литер. Вследствие этого не имеет значения для конкретного языка программирования, представлены ли литеры знаками на бумаге или на экране терминала — главное, чтобы они образовали некоторый текст алгебры слов. Литеры группируются в несколько подмножеств, которые удобно представить в виде подтипов типа «литера»:

подтип *буква=литера* (A, B, …Z, A, Б, …, Я, a, b, …, z, а, б, …, я);

подтип *цифра=литера* (0, 1,2, 3, 4, 5, 6, 7, 8, 9);

подтип *буква или цифра=литера* (0, 1,2, …9, A, B, …Z, A, Б, …, Я, a, b, …, z, а, б, …, я);

подтип *специальный знак=литера* (+, -, !, ;, %, ?, *, …).

Строки литер можно представить в виде экземпляров типа данных со следующей спецификацией:

тип *строка литер*=**модуль**

[*создать строку* : **функция** (*первая литера* : *литера*)*;

наростить строку : **функция** (*строка* : *, *следующая литера* : *литера*) *].

Аналогичным способом образуются типы «строка букв», «строка цифр», «строка букв и цифр». Функции «создать строку» и «наростить строку» представляются в программах только своими аргументами, между которыми не ставится никаких разделительных знаков, например:

123+456 — строка литер,
87654321 — строка цифр,
АБВГД — строка букв,
25Е2 — строка букв и цифр.

Обычно идентификаторы типов «строка цифр», «строка букв», «строка букв и цифр» не принадлежат языку программирования, поэтому в программах невозможно использовать эти типы для описания переменных, спецификации параметров и результатов функций. При необходимости программист может самостоятельно ввести указанные типы данных в программу.

Идентификатор. Идентификаторы служат для именования различных объектов типов данных (функций, переменных, значений и т. д.), вследствие чего в большинстве языков программирования сами они остаются лишь объектами алгебры слов. Язык программирования Паскаль, по-видимому, первым продемонстрировал отход от этой практики, дав возможность определять идентификационные значения скалярных типов данных. Например, при наличии описаний переменных

var *день*1, *день*2 : *(пн, вт, ср, чт, пт, сб, вс);*

в операторе

день1 := день2;

оба идентификатора обозначают переменные, а в операторе

день1 := вт;

второй идентификатор обозначает «сам себя», т. е. идентификаторное значение.

В языке С эквивалентные описания переменных будут иметь вид:

enum *{ пн, вт, ср, чт, пт, сб, вс } день1, день2.*

Непосредственным следствием использования идентификаторных значений является введение типа идентификаторов с соответствующими операциями создания и анализа:

тип *идентификатор*=**модуль**

[create_ident : **функция** (*первый_символ : буква, следующие_символы : строка_букв_и_цифр*) *;

«=», «≠» : **функция** (*идентификатор*1*, идентификатор*2 : *) *логическое*].

Предполагается, что в этой спецификации использованы предварительно определенные типы данных «*буква*», «*строка_букв_и_цифр*», «*логическое*».

Операция create_ident вырабатывает идентификатор из буквы и строки букв и цифр. Ввиду частого использования идентификаторов в программах для облегчения работы пользователей во всех языках программирования принято соглашение явно указывать только аргументы этой операции, не отделяя их друг от друга никакими знаками. В то же время вычислитель постоянно пользуется ею при распознавании и формировании внутреннего представления идентификаторов. Поскольку аргументы данной операции являются базисными термами (константами) алгебры слов, она также порождает базисные термы алгебры слов. Благодаря такому свойству всегда возможно прямое вычисление идентификаторных значений, при котором функции α соответствует функция перевода идентификатора во внутреннее представление, а функции β — само внутреннее представление.

Анализирующие операции «=» и «≠» дают вычислителю возможность сравнивать идентификаторы между собой с целью нахождения определяющего вхождения, проверки уникальности и т. д.

Переменная. Модель вычислений практически любого существующего языка программирования основывается на понятии памяти, как пассивного хранилища информации и процессора как активного ее преобразователя. В языке программирования понятие ячейки памяти отражается в понятии *переменной*, а адреса — в имени (идентификаторе) переменной. В соответствии со сказанным переменные обычно рассматриваются как некоторые «контейнеры», в которых можно размещать различные объекты.

Практически все языки программирования встраивают переменные в свои системы типов: существуют понятия параметров-значений и параметров-переменных, делается различие между левой и правой частями оператора присваивания и т. п. Поддерживая линию рассмотрения переменных на уровне

типов данных, будем понимать под переменными объекты, с которыми могут связываться на определенное время объекты других типов. Множество переменных, с которыми могут связываться значения данного типа, образует конкретный тип переменной. Каждое значение типа переменной является парой, один компонент которой создается операцией образования переменной и всегда для данной переменной имеет постоянное значение, а второй компонент может посредством специальной операции присваивания менять свое значение. Все конкретные типы переменной являются конкретизациями родового типа переменной со следующей спецификацией:

тип *переменная*=**модуль** (*T : тип*)
[create_var : **функция** *;
assign : **процедура** (*идентификатор_переменной : * , значение_переменной : T*);
value : **функция** (*идентификатор_переменной* : *) *T*;
«=», «≠» : **функция** (*переменная*1*, переменная*2 : *) *логическое*].

Предполагается, что тип «логическое» определен ранее.

Операция create_var создает новую переменную с неопределенным вторым компонентом. Эта операция может использоваться как вычислителем, так и программистом. Переменные, создаваемые вычислителем, принято называть *статическими*, а переменные, создаваемые программистами, — *динамическими*. Часто в языках программирования операция create_var предоставляется программисту в виде стандартной процедуры new.

Операция assign устанавливает значение второй компоненты переменной равным некоторому значению ее типа. В языках программирования операция assign известна под названием «*оператор присваивания*». У этой операции первый операнд всегда имеет тип переменной, а второй — тип ее значений.

Отрезок. Тип отрезок определяет множество значений, формируемое на основе заданного подмножества значений некоторого перечислимого типа. Это подмножество задается в общем случае указанием родительского типа, также нижней и верхней границ отрезка его значений, вследствие чего спецификация родового типа отрезка приобретает следующий вид:

тип *отрезок*=**модуль** (*родительский_тип* : **тип** < succ : **функция** (*значение* : *) * ;
«=» : **функция** (*значение*1, *значение*2 : *) *логическое*>,
нижняя_грань, *верхняя_грань* : *)[13]
[into : **функция** (*аргумент* : *родительский_тип*) * ;
out : **функция** (*значение* : *) *родительский_тип*) * ;
first, last : * ;
pred, succ : **функция** (*значение* : *) *;
«=», «≠», «>», «<», «≥», «≤» : **функция** (*значение*1, *значение2* : *) *логическое*].

[13] Заметим, что в спецификации параметра «родительский_тип» знак «*» обозначает тот тип, который будет подставляться в качестве аргумента при конкретизации типа «отрезок».

Предполагаем, что используемый в приведенной спецификации тип данных «*логическое*» был определен ранее.

Тип отрезка требует, чтобы все его аргументы относились к одному и тому же перечислимому типу (отражается указанием в спецификации типа параметра «*родительский_тип*» функции succ) и сам, в свою очередь, является перечислимым типом. Параметры «*нижняя_грань*» и «*верхняя_грань*» указывают соответственно нижнюю и верхнюю границы отрезка. Функции into и out осуществляют преобразование значения родительского типа в значение типа отрезка и наоборот и в языке программирования изображаются своими аргументами. Операции first, last, pred и succ вырабатывают соответственно первое, последнее, предыдущее и последующее значения типа, при этом значение атрибута first соответствует аргументу «*нижняя_грань*», а last — аргументу «верхняя_грань». Операция succ входит дважды. Первое вхождение соответствует родительскому типу, второе — определяемому типу. Они должны соответствовать друг другу. Однако второе вхождение учитывает ограничения множества значений, накладываемое типом данных *отрезок*. То же самое можно сказать и про операцию «=». Смысл остальных операций очевиден.

Как и тип перечисления, в языках программирования тип отрезок всегда представляется только своим аргументом. Когда родительский тип распознается однозначно по изображениям границ отрезка, он также не указывается явным образом. Поэтому вместо канонического

тип *от1до10=отрезок (целый, 1, 10),*

например, на Паскале пишут **тип** *от*1*до*10=1 .. 10.

Массив. Тип массива определяет множество значений, каждое из которых является совокупностью однотипных компонентов, представляющих функцию из некоторой области определения, задаваемой «типом индекса», в область значений, задаваемую «типом компонентов». Спецификация родового типа массива содержит обе эти области в качестве параметров.

```
тип массив=модуль (тип_индекса : тип < first, last : *;
                                    succ : функция(значение : * ) *>,14
    тип_компонентов : тип < «=« : функция (значение : * ) * >) 15
  [create_array : функция (n : последовательность( тип_компонентов)) *;
    lwb : функция ( m : * ) тип_индекса;
    upb : функция ( m : * ) тип_индекса;
    component : функция (m : *, i : тип_индекса) тип_компонентов;
    «=», «≠» : функция (m1, m2 : * ) логическое].
```

Предполагается, что типы данных «*последовательность*» и «*логическое*» предварительно получили свое описание.

В качестве типа индекса допускается обычно любой перечислимый тип ограниченной мощности (достаточной для размещения массива в памяти компьютера), что отражается в спецификации типа индекса требованием наличия

[14] См. сноску в этом параграфе.
[15] См. там же.

операций first, last и succ. В качестве области значений допускается любой тип, обладающий операцией «=».

Операция create_array создает значение массива на основе значения аргументной последовательности. Операции lwb (lower bound) и upb (upper bound) вырабатывают соответственно значения нижней и верхней границ массива. Операция component вырабатывает значение компонента массива, соответствующего заданному значению индекса. Переменная типа массива представляет собой совокупность переменных типа компонентов. Операции «=» и «≠» позволяют сравнивать между собой два однотипных массива.

В языках программирования операция component изображается обычно квадратными скобками вокруг индекса. Таким образом, если μ и *i* — переменные типа массива и типа индекса соответственно, то операция μ[i] вырабатывает переменную-компонент, соответствующую индексу *i*; если же μ — константа типа массива, то эта же операция вырабатывает значение компонента, соответствующего индексу *i*.

Операция конкретизации типа массива типом индекса T_i и типом компонентов T_k вместо канонического **массив** (T_i, T_k) языках программирования приобретает вид **array** [T_i] **of** T_k.

Функции. Введенных ранее средств достаточно, чтобы дать программисту возможность определять в программах собственные типы функций и конструировать соответствующие им конкретные функции. Эти средства воплощаются в родовом типе функции со следующей спецификацией:

тип функция=модуль (параметры_функции : блок_спецификаций, тип_результата : тип) [create_func : функция (тело_функции : терм тип_результата) *;
appl : функция (функция : * , параметры_функции) тип_результата].

Предполагаем, что типы данных «блок_спецификаций» и «терм» определены предварительно. Заметим, что при конкретизации типа функции тип результата может задаваться любым типовым выражением, т. е. не только идентификатором типа, но и, например, функцией, вырабатывающей результат типа **тип**.

Операция create_func на основе терма, вырабатывающего значение типа результата (тела функции), создает конкретную функцию данного типа. Она всегда префиксируется изображением типа функции и составляет с ним единое целое, в языках программирования она называется «описанием функции». Например, требуется образовать функцию *f* типа

функция *(a : целое, b : вещественное) вещественное*

с телом «a/b». В канонической форме запись этой операции приобретает вид:

*f=**функция** (a : целое, b : вещественное) вещественное) create_func(a/b);*

в языке программирования она приобретает форму описания функции

*f=**функция** (a : целое, b : вещественное) вещественное) **начало** a/b **конец.***

Таким образом, первая часть описания функции, а именно

функция *(a : целое, b : вещественное) вещественное,*

которая в языках программирования обычно называется заголовком, или спецификацией, функции, на самом деле есть описание конкретного типа функции, а вторая часть — описание операции создания функции. Стандартными действиями функции, созданной этой операцией, являются сохранение существующего контекста (запоминание в стеке содержимого некоторых регистров, глобальных ячеек и т. п.) перед началом исполнения собственно тела функции, установка нового контекста, восстановление предыдущего контекста после исполнения тела функции и возврат управления в точку вызова функции.

Операция appl означает применение (аппликацию) функции к своим аргументам с выработкой значения типа результата. В языках программирования она изображается обычно своими аргументами. Операция применения рассмотренной выше функции *f* к своим аргументам из канонической формы appl (*f*, 2, 3.5) в языке программирования запишется как *f* (2, 3.5).

Подобным же образом происходит аппликация и функций-модулей к своим аргументам для конкретизации родовых типов. Стандартными действиями этой операции являются помещение аргументов функции в определенные регистры и/или ячейки стека и передача управления на соответствующую функцию.

Контрольные вопросы

1. Поясните особенности спецификации типов данных «идентификатор», «переменная», «отрезок», «массив», «функция».

Задачи

В приведенных задачах нужно построить спецификации соответствующих типов данных.

1. Спецификация натурального типа данных

Построить спецификацию натурального типа данных. Натуральный тип определяет множество натуральных чисел. Основные операции — это создание «пустого» значения типа и конструктор, преобразующий строку цифр в значение натурального типа.

2. Спецификация целого типа данных

Построить спецификацию целого типа данных. Предусмотреть среди прочих операции: создания «пустого» значения типа, преобразования строки в целое, образования диапазона целых чисел.

3. Спецификация вещественного типа данных

Построить спецификацию вещественного типа данных. Предусмотреть среди прочих операции: создания «пустого» значения типа, конструктора значения типа, преобразования целого в вещественное, образования диапазона вещественных чисел.

4. Спецификация символьного типа данных

Построить спецификацию символьного типа данных. Символьный тип данных определяет множество значений, соответствующих множеству знаков,

имеющихся на устройствах ввода-вывода. В отличие от литер, значения символьного типа данных служат не для построения компонентов программ, а являются объектами обработки. Предусмотреть операцию конструирования символа, для которой функция реализации обычно не является фиктивной: она может производить такую работу, как удаление из исходного кода литеры бита нечетности или перекодировки литеры. Также предусмотреть операции: выработки «пустого» значения и диапазона.

5. Спецификация строки символов

Построить спецификацию строки символов.

6. Спецификация спецификации имени

Построить тип данных «спецификация имени». Данный тип определяет множество спецификаций имен, каждая из которых является парой <идентификатор, тип>. Предусмотреть операции: создания спецификации имени, выделения имени, выделения типа.

7. Спецификация константы

Построить спецификацию константы. Тип констант определяет множество констант, каждая из которых является тройкой <идентификатор, тип, значение>. В языках программирования константа задается часто в виде: const a=5. Предусмотреть операции: создания константы, поиска описания константы, получения идентификатора, типа, значения из описания константы. В качестве усложнения задачи можно рассмотреть отдельно спецификации: типовой константы, константы-переменной, константы-функции и константы-процедуры.

8. Спецификация типа данных «последовательность»

Построить спецификацию типа данных «последовательность». Последовательность является наиболее распространенным типом данных языков программирования, но редко оформляемым формально.

9. Спецификация типа данных «блок спецификаций»

Построить спецификацию типа данных «блок спецификаций». Тип определяет множество значений, каждое из которых образуется последовательностью спецификаций имен.

10. Спецификация типа данных «запись» («структура»)

Построить спецификацию типа «запись».

11. Спецификация типа данных «оператор»

Построить спецификацию типа «оператор». Тип «оператор» определяет множество значений, являющихся операционными единицами языка програм-

мирования в том смысле, что каждое значение может быть проинтерпретировано вычислителем как указание исполнить то или иное действие. Обычно идентификатор этого типа скрыт от пользователя, поэтому нет возможности описывать в программах переменные этого типа.

12. Спецификация типа данных «программа»

Построить спецификацию типа «программа». В отношении этого типа данных справедлив комментарий к задаче 11.

ЛАНЬ®

ЛАНЬ®

5. ПРОГРАММИРОВАНИЕ В ОПЕРАЦИОННОЙ СИСТЕМЕ WINDOWS

Никогда не вини себя в том, в чем можно обвинить систему.

С сайта:
http://www.aforizmov.net/tema/tags/sistema/page/2/

Сегодня уже трудно вспомнить то время, когда не было Windows. Как бы программисты ни относились к Windows, нужно признать, что это настоящее, а может быть и будущее компьютерной индустрии.

Стиль программирования Windows-приложений принципиально отличается от того, который сложился в операционных системах раннего поколения. В операционной системе MS DOS (Microsoft Disk Operating System) программа монопольно владеет всеми ресурсами системы и является инициатором взаимодействия с операционной системой. Совсем иначе дело обстоит в операционной системе Windows, которая строилась как многозадачная, и именно операционная система является инициатором обращения к программе. Все ресурсы Windows являются разделяемыми, и программа, называемая приложением, не может владеть ими монопольно. В связи с такой идеологией построения операционной системы приложение должно ждать посылки сообщения операционной системы и лишь после его получения выполнить определенное действие, затем вновь перейти в режим ожидания очередного сообщения.

Windows генерирует множество различных сообщений, которые направляются приложению, например, о щелчке кнопки мыши или о нажатии клавиши на клавиатуре. Если приложение не обрабатывает какие-то сообщения, реакция на них осуществляется операционной системой стандартным способом, так что задачей программиста является обработка лишь тех сообщений, которые необходимы приложению.

Интерфейс прикладного программирования (Application Programming Interface — API) Windows представляет собой набор функций, при помощи которых любое приложение может взаимодействовать с операционной системой и, через операционную систему, с другими приложениями. Масштаб современного API впечатляет, он содержит несколько тысяч функций, сотни сообщений, макросов, предопределенных констант.

В системе программирования RAD Studio используется очень мощная и сложная библиотека визуальных компонентов VCL (Visual Components Library), которая помимо непосредственных абстракций содержит также средства организации взаимодействия с операционной системой. В этой библиотеке находятся компоненты для визуального отображения информации, работы с базами данных, с системными объектами, компоненты для работы с Internet-протоколами, классы для написания своих COM-объектов и многое другое. Модули библиотеки подключаются к компиляции по мере необходимости.

Создавая приложение, программист постоянно решает вопросы взаимодействия программы с операционной системой. Используя систему программи-

рования, программист, как правило, избавлен от явного программирования многих действий, так как эти действия инкапсулированы в VCL. Однако для понимания поведения и квалифицированного использования компонентов необходимы знания принципов и идей, заложенных в Windows. Кроме того, без знания Windows многие возможности операционной системы могут остаться неиспользованными. Поэтому предлагаемая глава представляет собой краткое введение в операционную систему Windows.

Выбор обсуждаемых тем обусловлен, не в последнюю очередь, попыткой ответа на вопросы: «Почему операционная система является операционной системой «Windows» («окна»)?» и «Как из этих «окон» извлечь инструментальные средства разработки программ?»

5.1. Основные концепции Windows

В операционной системе Windows интерфейс пользователя представляет собой целостную систему различных элементов. Элементами служат окна, органы управления, меню, диалоговые панели и другие объекты Windows. Каждый элемент обычно задают множеством параметров состояния, входных и выходных сообщений. Для элементов одного класса описывают единый метод изменения параметров состояния и обработки входных и выходных сообщений. Основной задачей проектирования интерфейса пользователя является разработка интегрированной системы управления множеством состояний программного продукта.

Любую современную программу или программную технологию можно представить как совокупность программных «слоев». Каждый из этих слоев производит свою собственную работу, которая заключается в повышении уровня абстракции производимых операций. Так, самый низший слой (слои) вводит понятия, которые позволяют абстрагироваться от используемого оборудования; следующий слой (слои) позволяет программисту абстрагироваться от сложной последовательности вызовов функций, вводя такое понятие, как *протокол* и т. д. Практически в любом современном программном продукте можно обнаружить и выделить около десятка последовательных слоев абстракции.

Абстракция от оборудования и низкоуровневых протоколов вводится в ядра операционных систем в виде библиотек API (Application Programming Interface). Однако современные тенденции приводят к необходимости абстрагирования и от самих операционных систем, что позволяет переносить программы с одной операционной системы на другую путем простой перекомпиляции (транслируемые программы, в основном, вообще не требуют никаких действий по переносу).

Абстракцию, которая доступна программисту в виде библиотек API, можно назвать *базовой*. Это самый низкий уровень абстракции, который доступен для прикладного программирования. На уровне ядра системы доступны и более низкие уровни абстракции, однако для их использования необходимо разрабатывать специализированные программы (драйвера, модули). Базовый уровень абстракции (API) предоставляет максимально широкие возможности

для прикладного программирования и является наиболее гибким. Однако программирование с использованием API является гораздо более трудоемким и приводит к значительным объемам исходного кода программы, чем программирование с использованием дополнительных библиотек.

Дополнительные библиотеки поставляются со многими средствами разработки с целью уменьшения трудоемкости и сроков разработки программ, что в итоге приводит к повышению их конкурентоспособности. Но применение дополнительных библиотек абстракций приводит к резкому увеличению размеров откомпилированных программ из-за того, что в программу включается код используемых библиотек, к тому же это включение зачастую бывает неэффективным — в программу включаются неиспользуемые участки кода. Кроме того, чем больше уровень абстракции библиотеки, тем сложнее ее код и тем больше трудностей возникает при решении сложных задач. Приходится учитывать множество взаимосвязей и взаимных влияний отдельных элементов и процессов библиотеки друг на друга. Кроме того, структура и функциональность любой библиотеки обычно рассчитывается на удовлетворение всех потенциально возникающих задач, что приводит к ее громоздкости и неэффективности.

Среди преимуществ операционной системы Windows обычно выделяют следующие особенности применяемых инструментов:

— графический интерфейс пользователя;
— многозадачность;
— управление памятью;
— независимость от аппаратных средств.

5.1.1. Графический интерфейс пользователя

Graphical User Interface[16] (GUI) дает возможность пользователям работать с приложениями максимально удобным способом. Каждое приложение представлено на экране дисплея своим окном, которое выглядит как прямоугольная рабочая область с набором стандартных элементов управления. Окно идентифицируется своим заголовком, имеет кнопки минимизации и максимизации размеров, а также кнопку завершения приложения. Под заголовком часто находится строка меню для выбора различных команд или режимов работы. Всплывающие окна диалога, вызываемые командами меню, также содержат привычные для пользователя элементы управления, такие как текстовые поля ввода информации, открывающиеся списки и кнопки различных типов. Большинство программ для Windows поддерживают работу и с клавиатурой, и с мышью.

Стандартизация графического интерфейса имеет очень большое значение для пользователя, потому что одинаковый интерфейс экономит его время и упрощает изучение новых приложений. С точки зрения программиста, стандартный вид интерфейса обеспечивается использованием подпрограмм, встроенных непосредственно в Windows, что также приводит к существенной экономии времени при написании новых программ.

[16] Графический интерфейс пользователя.

5.1.2. Многозадачность

Многозадачные операционные системы позволяют пользователю одновременно работать с несколькими приложениями или несколькими копиями одного приложения. Например, пользователь может открыть систему программирования и набирать в текстовом редакторе код своей программы, одновременно слушая приятную музыку. В это же время программа **explorer.exe** может заниматься поиском необходимого файла по всему дисковому пространству компьютера.

Многозадачность осуществляется в Windows при помощи процессов и потоков. Любое приложение Windows после запуска реализуется как *процесс* (process). Грубо говоря, процесс можно представить как совокупность программного кода и выделенных для его исполнения системных ресурсов. При инициализации процесса система всегда создает первичный (основной) *поток* (thread), который исполняет код программы, манипулируя данными в адресном пространстве процесса.

Из основного потока при необходимости могут быть запущены один или несколько вторичных потоков, которые выполняются одновременно с основным потоком. На самом деле истинный параллелизм возможен только при исполнении программы на многопроцессорной компьютерной системе, когда есть возможность распределить потоки между разными процессорами. В случаях однопроцессорного компьютера или при нехватке процессоров для всех потоков операционная система выделяет по очереди некоторый квант времени каждому потоку.

Но каковы соотношения между потоками и окнами? Дело в том, что окно всегда принадлежит некоторому потоку. Поток может быть владельцем одного или нескольких окон, а может быть и вовсе безоконным. Например, если вторичный поток создан для приема данных из COM-порта, то он вполне может обойтись без своего окна.

Наконец, сами окна, принадлежащие потоку, находятся в некоторых иерархических взаимоотношениях. Одно окно является *окном верхнего уровня* (top-level window)[17], другие окна называются *дочерними* (child windows). Дочерние окна подчиняются своим родительским окнам (parent windows). Рисунок 5.1 иллюстрирует эти взаимоотношения.

Если на экране находится несколько окон, то в каждый момент времени только одно из них может быть *активным*. Активное окно отличается более ярким фоном на строке заголовка, в то время как пассивные окна имеют более тусклый фон заголовка. Говорят, что активное окно имеет *фокус ввода*. Это означает, что любая информация, вводимая пользователем при помощи клавиатуры или мыши, будет перенаправляться операционной системой именно данному окну.

[17] Окно верхнего уровня не имеет родительского окна.

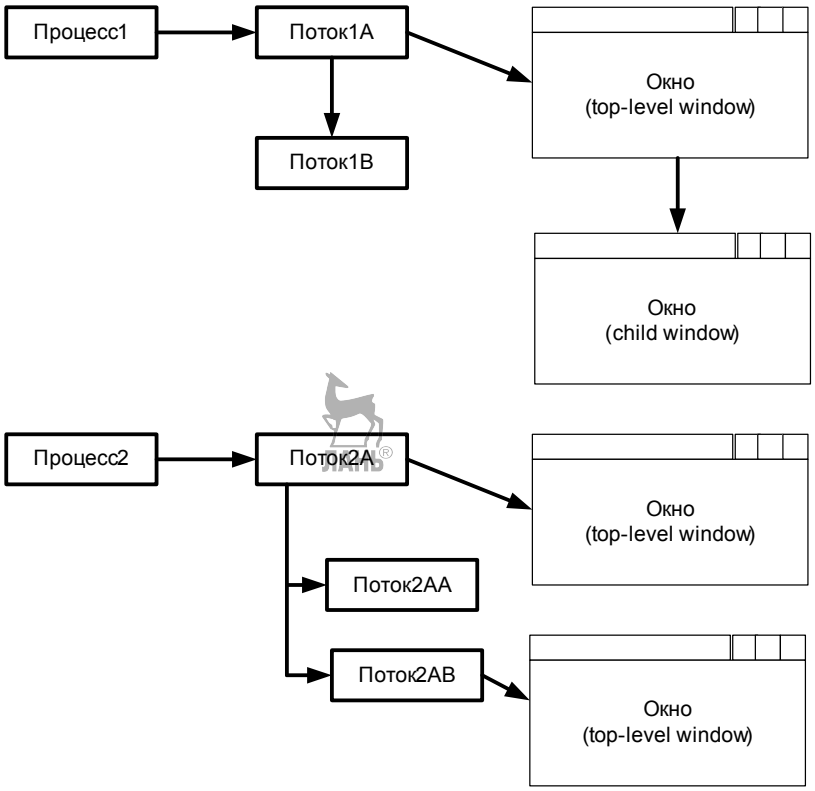


Рис. 5.1
Соотношение между процессами, потоками, окнами

5.1.3. Управление памятью

Память — это один из важнейших разделяемых ресурсов в операционной системе. Если одновременно запущены несколько приложений, то они должны разделять память, не выходя за пределы выделенного адресного пространства. Так как одни программы запускаются, а другие завершаются, то память фрагментируется. Система должна уметь объединять свободное пространство памяти, перемещая блоки кода и данных.

Система Windows обеспечивает достаточно большую гибкость в управлении памятью. Если объем памяти меньше объема исполняемого файла, то система может загружать исполняемый файл по частям, удаляя из памяти отработавшие фрагменты. Если пользователь запустил несколько копий, которые также называют отдельными экземплярами приложения, то система размещает в памяти только одну копию исполняемого кода, которая используется этими экземплярами совместно. Программы, запущенные в Windows, могут использовать также функции из других файлов, которые называют *библиотеками динамической компоновки* — DLL (dynamic link libraries). Система Windows поддерживает механизм связи программ во время их работы с функциями из DLL. Даже сама операционная система Windows, по существу, является набором динамически подключаемых библиотек.

Механизмы управления памятью непрерывно совершенствуются по мере развития Windows.

5.1.4. Независимость от аппаратных средств

Еще одним преимуществом Windows является независимость от используемой платформы. У программ, написанных для Windows, нет прямого доступа к аппаратной части таких устройств отображения информации, как, например, экран и принтер. Вместо этого они вызывают функции графической подсистемы, называемой *графическим интерфейсом устройства* (Graphic Device Interface, GDI).

Функции GDI реализуют основные графические команды при помощи обращения к программным драйверам соотвествующих аппаратных устройств. Одна и та же команда (например, LineTo — нарисовать линию) может иметь различную реализацию в разных драйверах. Эта реализация скрыта от программиста, использующего Windows API, что упрощает разработку приложений.

Таким образом, приложения, написанные с использованием Windows API, будут работать с любым типом дисплея и любым типом принтера, для которых имеется в наличии драйвер Windows. То же самое относится и к устройствам ввода данных — клавиатуре, манипулятору «мышь» и др. Такая независимость Windows от аппаратных средств достигается благодаря указанию требований, которым должна удовлетворять аппаратура, в совокупности с SDK (Software Development Kit — набор разработки программ) и/или DDK (Driver Development Kit — набор разработки драйверов устройств). Разработчики нового оборудования поставляют его вместе с программными драйверами, которые обязаны удовлетворять этим требованиям.

5.1.5. Вызов функций

Windows API поддерживает вызовы свыше двух тысяч функций, которые можно использовать в приложениях. Все основные функции Windows объявлены в заголовочных файлах. Главным заголовочным файлом является **windows.h**. В этом файле содержится множество ссылок на другие заголовочные файлы.

Вызовы функций Windows API в программе осуществляются аналогично вызовам библиотечных функций систем программирования. Для языков программирования C и C++ основное различие заключается в том, что компоновщик связывает код библиотечных функций с кодом программы на этапе компоновки (статическое связывание), в то время как для функций Windows это связывание откладывается и осуществляется только на этапе выполнения программы (динамическое связывание). Библиотеки динамической компоновки (DLL) содержатся в файлах с расширением **.dll**. Большая часть этих библиотек расположена в подкаталоге SYSTEM каталога установки Windows.

Операционная система позволяет компилятору использовать библиотеки импорта, поставляемые в составе используемой среды программирования. Библиотеки импорта содержат имена всех функций из динамически подключаемых библиотек, а также ссылки на них. Используя эту информацию, компоновщик

размещает в исполняемом файле таблицу, по которой в процессе загрузки программы настраиваются адреса вызываемых функций Windows API.

5.1.6. Разделение аппаратных ресурсов

Одним из наиболее существенных свойств Windows является то, что под его управлением можно одновременно запускать на выполнение несколько программ. Различные приложения делят между собой эти ограниченные ресурсы, распределение которых осуществляет операционная оболочка Windows.

Программируя для Windows, нельзя забывать о том, что все аппаратные ресурсы являются разделяемыми, и для их правильного использования следует придерживаться определенных правил.

Так, если программа для MS-DOS имеет доступ ко всей свободной оперативной памяти и может обращаться к этой памяти так, как захочется программисту, то в Windows, где одновременно может выполняться несколько программ, память является разделяемым ресурсом.

Другим примером разделяемого ресурса является дисплей. Программируя в MS-DOS, можно в любой момент переключить режим видеоадаптера, изменить цвет или даже напрямую обратиться к контроллеру и что-то записать в видеопамять. Такая техника полностью исключается при программировании в Windows. Дисплей — разделяемый системный ресурс, и обращаться к нему можно только так, как позволяет Windows.

Одной из основных целей создателей Windows являлось обеспечение визуального доступа к большинству приложений одновременно. Для этого каждому приложению выделяется часть экрана для взаимодействия с пользователем. В Windows каждое приложение в любой момент времени имеет доступ к определенной части экрана. Таким образом, экран в Windows — разделяемый ресурс, средством разделения которого является окно приложения.

Итак, два важных вывода.

Во-первых, загруженная на выполнение программа не забирает на себя полностью все ресурсы — эти ресурсы выделяются приложению при помощи специальных механизмов.

Во-вторых, в каждый момент времени программа не знает, какая именно область экрана принадлежит окну приложения, поскольку пользователь в любой момент может переместить окно и изменить его размеры (для определения координат окна нужно обратиться к специальной функции Windows).

5.1.7. Объектно-ориентированное программирование

Хотя формально операционная система Windows не является объектно-ориентированной системой, тем не менее в ней реализована именно объектно-ориентированная идеология. Это наиболее очевидно для базового объекта, с которым имеют дело и операционная система, и программист, и пользователь приложения, т. е. для окна.

Как уже говорилось ранее, окно выглядит как прямоугольная область на экране. Окно получает информацию от клавиатуры или мыши пользователя и выводит графическую информацию на экран.

Пользователь рассматривает окна в качестве объектов и непосредственно взаимодействует с ними, перемещая их по экрану, выбирая команды меню, нажимая различные кнопки и передвигая бегунок на полосах прокрутки.

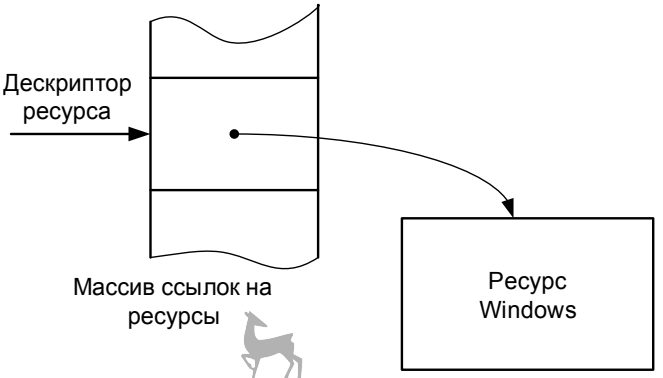


Рис. 5.2
Понятие дескриптора ресурса

Программист тоже рассматривает окна в качестве объектов, которые получают от пользователя информацию в виде оконных сообщений. Кроме этого, окно обменивается сообщениями с другими окнами. Понимание этих сообщений — ключ к осмысленному программированию в среде Windows.

Можно привести и другие примеры объектов в Windows, к которым относятся, например, многочисленные графические объекты, используемые для рисования, такие как перья, кисти, шрифты, палитры и многие другие объекты.

Независимо от своего типа, любой объект (ресурс) в Windows идентифицируется своим *дескриптором*, или *описателем*. Оба эти названия являются переводом английского термина handle. Дескриптор — это своего рода ссылка на объект. Все взаимоотношения программного кода с объектом осуществляются только через его дескриптор. Система Windows тщательно скрывает свои внутренние секреты и не допускает прямого доступа к внутренним структурам объекта. Упрощенно понятие дескриптора ресурса иллюстрируется на рисунке 5.2: дескриптор ресурса представляет собой индекс массива ссылок на ресурсы.

5.1.8. Типы данных

Программиста, пишущего на C/C++, первое знакомство с программой для Windows поражает обилием типов данных. На самом деле все они определены посредством директив #define или #typedef в заголовочных файлах Windows. В таблице 5.1 приведены некоторые наиболее часто встречающиеся типы данных Windows.

Таблица 5.1

Некоторые часто используемые типы данных Windows

Тип	Пояснение
BOOL	Булевский тип
BYTE	Байт (8-битное целое без знака)

Продолжение табл. 5.1

Тип	Пояснение
DWORD	32-битное целое без знака
HANDLE	Дескриптор объекта
HBITMAP	Дескриптор растрового изображения
HBRUSH	Дескриптор кисти
HCURSOR	Дескриптор курсора
HDC	Дескриптор контекста устройства
HFONT	Дескриптор шрифта
HICON	Дескриптор иконки
HINSTANCE	Дескриптор экземпляра приложения
HMENU	Дескриптор меню
HPEN	Дескриптор пера
HWND	Дескриптор окна
INT	32-битное целое со знаком
LONG	32-битное целое со знаком
LPARAM	Тип, используемый для описания IParam, четвертого параметра оконной процедуры
LPCSTR	Указатель на константную С-строку[18]
LPCTSTR	LPCWSTR, если определен макрос UNICODE, и LPCSTR в противном случае
LPCWSTR	Указатель на константную Unicode-строку[19]
LPSTR	Указатель на С-строку
LPTSTR	LPWSTR, если определен макрос UNICODE, и LPSTR в противном случае
LPWSTR	Указатель на Unicode-строку
LRESULT	Значение типа LONG, возвращаемое оконной процедурой
NULL	((void*)0)
TCHAR	Wchar_t(Unicode-символ), если определен макрос UNICODE, и char в противном случае
UINT	32-битное целое без знака
WPARAM	Тип, используемый для описания wParam, третьего параметра оконной процедуры

ЛАНЬ®

Контрольные вопросы

1. Опишите значение стандартизации графического интерфейса для программирования и для использования программ графического интерфейса пользователя.
2. Что понимается под многозадачностью операционной системы?
3. Чем отличаются понятия «процесс» и «поток»?
4. Что считается окном верхнего уровня?
5. Может ли процесс обходиться без окна?
6. Может ли поток обходиться без окна?
7. Как в Windows используется оперативная память?

[18] С-строка обязана завершиться нулевым байтом. В каждом байте содержится один символ в соответствии с кодировкой ANSI. Далее, если это не оговаривается особо, под строками будут подразумеваться именно С-строки.

[19] Unicode-строка также завершается нулевым байтом, но для хранения каждого символа используется два байта. Чтобы использовать в проекте кодировку UNICODE, необходимо предусмотреть директиву #define UNICODE.

8. За счет чего обеспечивается в Windows независимость от аппаратных средств?

9. Объясните понятия «динамическое связывание», «статическое связывание».

10. Является ли экран разделяемым ресурсом?

11. Что является дескриптором устройства?

5.2. Архитектура, управляемая событиями

В основу Windows положен принцип *событийного управления.* Это значит, что и сама система, и приложения после запуска ожидают действий пользователя и реагируют на них заранее заданным образом. Любое действие пользователя (нажатие клавиши на клавиатуре, щелчок кнопкой мыши, перемещение мыши) называется *событием.*

Для того чтобы уметь хорошо проектировать приложения для операционной системы Windows, весьма полезно рассмотреть анатомию простого Windows-приложения.

Windows-приложения содержат два элемента: функцию WinMain и оконную процедуру.

Функция WinMain является обязательным компонентом любого приложения. С нее начинается выполнение программы, и ею же обычно заканчивается. Эта функция состоит из трех частей, в которых выполняется:

— начальная инициализация программы (подготовка данных класса окна и его регистрация);

— создание главного окна приложения;

— запуск цикла обработки сообщений, извлекаемых из очереди сообщений приложения.

В основе взаимодействия программы с внешним миром и с операционной системой лежит концепция *сообщений.*

С точки зрения приложения, сообщение является уведомлением о том, что произошло некоторое событие, которое может требовать, а может и не требовать выполнения определенных действий. Это событие может быть следствием действий пользователя, например, перемещения курсора или щелчка кнопкой мыши, изменения размеров окна или выбора пункта меню. Кроме того, событие может генерироваться приложением, а также операционной системой.

Сообщение — это структура данных, содержащая следующие элементы:

— дескриптор окна, которому адресовано сообщение;

— код (номер) сообщения;

— дополнительную информацию, зависящую от кода сообщения (параметры сообщения).

Не следует забывать, что Windows — многозадачная операционная среда. Сообщения от внешних источников, например от клавиатуры, адресуются в каждый момент времени только одному из работающих приложений, а именно — приложению, имеющему активное окно. Но каким же образом Windows играет роль диспетчера сообщений? Для этого с момента старта операционная система

создает в памяти глобальный объект, называемый *системной очередью сообщений*. Все сообщения, генерируемые как аппаратурой, так и приложениями, помещаются в эту очередь. Windows периодически опрашивает эту очередь и, если она не пуста, посылает очередное сообщение нужному адресату, определяемому при помощи дескриптора окна.

Сообщения, получаемые приложением, могут поступать асинхронно из разных источников. Например, приложение может работать с системным таймером, посылающим ему сообщения с заданным интервалом, и одновременно оно должно быть готово в любой момент времени получить любое сообщение от операционной системы. Чтобы не допустить потери сообщений, Windows одновременно с запуском приложения создает глобальный объект, называемый *очередью сообщений приложения*. Время жизни этого объекта совпадает со временем жизни приложения.

Таким образом, путь следования сообщений, вызванных аппаратными событиями, можно представить так:

аппаратное событие → системная очередь сообщений → очередь сообщений приложения → приложение.

5.2.1. Оконные процедуры и функции обратного вызова

Для понимания механизма обработки приложением поступающих к нему сообщений сначала нужно выяснить, чем является «оконная процедура».

Программисты на языке C хорошо знакомы с понятием вызова операционной системы для выполнения каких-то действий. Например, функция **fopen** используется для открытия файла. Библиотечные функции, поставляемые компилятором, содержат код, который фактически вызывает для открытия файла операционную систему. Здесь все просто. Но операционная система Windows ведет себя иначе.

Хотя в Windows имеется очень много доступных программисту функций, Windows также и сама посылает вызовы программе. Такие вызовы называются сообщениями и регистрируются оконными процедурами. Свое название оконная процедура получила в связи с тем, что любое окно Windows имеет программную поддержку в виде отдельной процедуры. Оконная процедура связана с классом окна, который программа регистрирует с помощью вызова функции **RegisterClass**. Таким образом, операционная система имеет сведения о каждом окне и сообщает оконной процедуре, какие изменения должны произойти в управляемом этой оконной процедурой окне. Другими словами, окно, создаваемое на основе оконного класса, использует оконную процедуру для обработки всех посылаемых окну сообщений; Windows посылает сообщения окну, вызывая оконную процедуру.

Рассмотрим ситуации, когда Windows обращается к оконной процедуре приложения:

— первый раз при создании окна;

— при изменении размеров окна, при его перемещении, при его сворачивании;

— при выборе пункта меню;

— при манипуляции с полосами прокрутки или с мышью;
— при необходимости перерисовать рабочую область окна;
— при удалении окна и т. п.

Все эти вызовы имеют форму сообщений. В большинстве Windows-программ основная часть программы направлена на обработку этих сообщений. Свыше 200 различных сообщений, которые Windows может отправить оконной процедуре, идентифицируются именами, которые начинаются с символов **WM_** и определяются в заголовочных файлах Windows.

Фактически идея функции, находящейся в программе, но которая вызывается не из самой программы, не является абсолютно новой в традиционном программировании. Функция **signal** в языке С может перехватить нажатие комбинации клавиш <Ctrl+Break>. Операционная система регистрирует нажатие клавиш <Ctrl+Break> и сообщает прикладной программе о произошедшем событии. В свою очередь, прикладная программа должна иметь средства ответной реакции на произошедшее событие. В Windows эта идея расширена и пронизывает всю систему. Любое событие, относящееся к окну, передается оконной процедуре в виде сообщения**.** Таким способом реализуется идея, получившая название «событийное программирование».

Оконная процедура играет роль диспетчера сообщений. Сами приложения никогда не получают к ней прямого доступа. Для этого им сначала нужно сделать запрос к Windows на выполнение соответствующего действия. Поэтому все оконные процедуры должны быть функциями обратного вызова (callback). Подобная функция регистрируется в операционной системе и вызывается всякий раз, когда пользователь или программа выполняют некоторое действие над окном.

Когда приложение обрабатывает сообщения, оно обычно делает это посредством вызова функции **DispatchMessage** для каждого принятого сообщения — это происходит в цикле обработки сообщений. В свою очередь, функция **DispatchMessage** вызывает соответствующую оконную процедуру, проверяя, какому классу окна предназначено сообщение. Именно эта оконная процедура (функция окна) действительно обрабатывает сообщение, переданное окну.

Основное поведение окна определяется классом окон. Класс окна (не путать с понятием «класс» языка программирования) хранит информацию о начальном внешнем виде окна, пиктограмме по умолчанию, курсоре и ресурсе меню, связанном с окном, и, что, возможно, более важно — об адресе функции, называемой оконной процедурой (оконной функцией).

Оконная функция анализирует это сообщение и его параметры, а затем, при необходимости, выполняет некоторые действия.

Можно считать, что единственная для каждого окна функция окна реализует все методы окна как объекта. В языке С++, напротив, каждый метод объекта реализуется отдельной функцией. Для реализации методов оконная функция анализирует сообщение и однозначно определяет нужный метод.

Сформулируем теперь сказанное кратко в виде определений.

Оконная процедура — это функция, непосредственно связанная с конкретным окном и предназначенная для обработки сообщений, адресованных «оконному классу», в котором содержится ссылка на данную процедуру.

Рассмотрим и другое определение оконной процедуры, которое отражает происходящие при обработке событий процессы.

Оконная процедура — это функция, имеющая возможность получать и вызывать функции обратного вызова.

Функция обратного вызова (callback, *англ*. call — вызов, *англ*. back — обратный) — это функция, которая является параметром другой функции. Вызов передаваемой функции получил название обратного вызова. Обратный вызов позволяет в некоторой функции исполнять код, который задается в параметрах этой функции. Код функции обратного вызова определяется обычно в других контекстах программного кода и может быть недоступным для прямого вызова из этой функции.

Поясним на примере ипользование функции обратного вызова в традиционном программировании. Пусть имеется массив, который мы хотим сортировать разными способами упорядочения. Решение этой задачи приведено в программном коде:

```
int mas[10]={1,-3,5,-83,7,32,45,67,-4, 0};
typedef int ( * ptrFunction)(int, int);
ptrFunction SortCondition;
int nonDecreasing(int a, int b)      //  non-decreasing order — неубывание
{
  return (a <= b);
}
int nonIncreasing(int a, int b)      //   nonincreasing — невозрастание
{
  return (a >= b);
}
void SortingPermutations(int m[], int len, ptrFunction SortOrder)
{
        bool boolArePermutations;
        do
          {
                boolArePermutations=False;
                for (int i=0; i < len – 1; i++)
                {
                      if (!SortOrder (m[i],m[i+1]))
                      {
                        boolArePermutations=True;
                        int r=m[i];
                        m[i]=m[i+1];
                        m[i+1]=r;
                      }
                }
          }
        while (boolArePermutations);
}

void MainFunction()
```

```
{
        SortCondition=&nonDecreasing;
        SortingPermutations(mas, 10, SortCondition);
                        …
        SortCondition=&nonIncreasing;
        SortingPermutations(mas, 10, SortCondition);
}
```

В первой строке приведенного примера находится описание подлежащего сортировке массива **mas**. В следующих двух строках идет описание типа данных **ptrFunction**, который является ссылкой на функцию с сигнатурой (int, int, int), и переменной **SortCondition** соответствующего типа. Далее идет описание двух функций: **nonDecreasing** (сравнение двух элементов по неубыванию) и **nonIncreasing** (сравнение двух элементов по невозрастанию). Функция **SortingPermutations** выполняет сортировку элементов массива по методу перестановок и получает в качестве третьего параметра **SortOrder** — ссылку на функцию обратного вызова. Первые два параметра определяют, соответственно, получаемый для сортировки массив и его размер. Обратный вызов осуществляется в строке **if (!SortOrder (m[i],m[i+1]))**. Внутри функции **MainFunction** приведены варианты использования функций обратного вызова.

Замечание. В литературе нет единства в понимании термина «функция обратного вызова». В одном случае функцией обратного вызова называют функцию, в параметрах которой имеется функция, которая вызывается внутри первой функции. В другом случае под функцией обратного вызова понимают вторую — вызываемую функцию. Не претендуя на абсолютную истину и для определенности в настоящей работе под функцией обратного вызова, будем понимать функцию, которая вызывается внутри основной функции.

Для лучшего понимания причин использования обратного вызова рассмотрим задачи выполнения следующих операций над массивом чисел: вывести на экран все числа, возвести все числа в квадрат, обнулить массив, увеличить все числа на два и т. п. В основе алгоритма этих четырех операций лежит цикл обхода всех элементов массива с некоторым действием в теле цикла, применяемый к каждому элементу. Это несложный код, и в принципе можно написать его 4 раза. Но давайте рассмотрим более сложный случай, когда список у нас хранится не в памяти, а на диске, и со списком могут работать несколько процессов одновременно, и необходимо решать проблемы синхронизации доступа к элементам (несколько процессов могут выполнять разные задачи — удаления некоторых элементов из списка, добавления новых, изменение существующих элементов в списке). В этом случае задача обхода всех элементов списка будет довольно сложным кодом, который не хотелось бы копировать несколько раз. Правильнее создать функцию общего назначения для обхода элементов списка и дать возможность программистам абстрагироваться от того, как именно устроен алгоритм обхода и записывать лишь функцию обратного вызова для обработки отдельного элемента списка.

Функция обратного вызова является альтернативой полиморфизму функций, а именно, позволяет создавать функции более общего назначения вместо того, чтобы создавать серию функций, одинаковых по структуре, но отличающихся лишь в отдельных местах исполняемыми подзадачами. Функции, принимающие в качестве аргументов другие функции или возвращающие функции в качестве результата, называют функциями высшего порядка. Техника обратного вызова играет важную роль для достижения повторного использования кода.

Компилятор узнает функцию с возможностью обратного вызова по спецификатору CALLBACK. Оконная процедура обычно имеет заголовок со стандартным синтаксисом:

LRESULT CALLBACK Имя_функции (HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam).

В этом определении **LRESULT —** тип возвращаемого значения (см. табл. 5.1); **hWnd** — дескриптор окна, которому адресовано сообщение; **uMsg** — код сообщения; **wParam** и **lParam** — параметры сообщения. Имя функции может быть произвольным, но для главного окна приложения обычно используется имя WndProc. В теле функции после объявления необходимых локальных переменных обычно размещается оператор **switch**, внутри которого происходит распознавание и обработка нужных сообщений.

Каждому коду сообщения в Windows сопоставлен уникальный символический идентификатор. Все системные идентификаторы определены при помощи директивы **#define** в заголовочном файле **winuser.h**. Это облегчает чтение и понимание программ.

Чаще всего приложение обрабатывает *оконные сообщения* (window messages), начинающиеся с префикса **WM_**, например, **WM_PAINT**, **WM_SIZE**, **WM_MOVE** и многие другие. Другие типы сообщений могут поступать от элементов управления, например, сообщения с префиксом **BM_** поступают от кнопок, сообщения с префиксом **EM** — от текстовых полей, сообщения с префиксом **LB** — от списков.

При необходимости разработчик может определить в приложении и потом использовать собственные коды сообщений. Обмен такими сообщениями возможен, конечно, только между окнами данного приложения. Другие приложения не смогут обрабатывать пользовательские сообщения.

Теперь о новом термине «оконный класс». На самом деле все окна создаются на базе того или иного оконного класса. Оконный класс играет роль *типа* для данного окна.

ЛАНЬ®

5.2.2. Оконный класс

Оконный класс (window class), или класс окна, — это структура, определяющая основные характеристики окна. К ним относятся стиль окна и связанные с окном ресурсы, такие как пиктограмма, курсор, меню и кисть для закрашивания фона. Кроме того, одно из полей структуры содержит адрес оконной процедуры, предназначенной для обработки сообщений, получаемых любым окном данного класса.

Ссылка на оконный класс передается функции **CreateWindow**, вызываемой для создания окна.

Использование класса окна позволяет создавать множество окон на основе одного и того же оконного класса и, следовательно, использовать одну и ту же оконную процедуру. Например, все кнопки в программах Windows созданы на основе оконного класса **BUTTON**. Оконная процедура этого класса, расположенная в димически подключаемой библиотеке, управляет обработкой сообщений для всех кнопок всех окон. Аналогичные системные классы имеются и для других *элементов управления*, таких как, например, списки и поля редактирования. В совокупности эти классы называются *стандартными оконными классами.*

Windows содержит *стандартный оконный класс* также и для диалоговых окон, играющих важную роль в графическом интерфейсе пользователя.

Для главного окна приложения обычно создается собственный класс окна, учитывающий индивидуальные требования к программе.

Замечание. Не следует путать понятие оконного класса Windows с понятием класса C++ или, в частности, с понятием класса в библиотеках VCL или MFS. Дело в том, что исторически концепция оконных классов опередила по времени использование в Windows объектно-ориентированных языков.

5.2.3. Сообщения

Существует много разновидностей сообщений, представляющих события на разных уровнях. Каждому простому событию, каждому простому действию сопоставляется сообщение, посылаемое окну для последующей обработки.

Приходящее приложению Windows-сообщение состоит из нескольких частей, для его представления используется структура **MSG**. Приведем описание структуры, применяющейся для представления сообщения, т. е. описывающей тип сообщения:

```
typedef struct tagMSG { HWND hwnd;      // окно, которому посылается это сообщение
                        UINT message;   // значение самого сообщения
                       WPARAM wParam; // для передачи дополнительной информации
                       LPARAM lParam; // для передачи дополнительной информации
                        DWORD time;         // время, когда произошло событие
                        POINT pt;         // точка, где произошло событие (для мыши)
                        } MSG;
```

Элемент структуры **hwnd** однозначно идентифицирует окно, которому посылается это сообщение. Каждое окно в Windows имеет такой идентификатор.

Элемент **message** идентифицирует само сообщение. Этот элемент может иметь сотни различных значений, означающих одно из многих сотен различных сообщений, которые могут получать приложения Windows. Для идентификаторов сообщений обычно используются символьные представления (например, **WM_PAINT**, **WM_TIMER**), а не числовые значения. Эти символьные значения определены в стандартных файлах заголовков Windows (приложению необходимо включать в свой исходный текст только файл **windows.h** — он содержит директивы #include для остальных файлов).

Сообщения можно разделить на несколько групп в зависимости от их функций. Самой насыщенной группой сообщений является группа сообщений управления окнами. Символьные идентификаторы для этих сообщений начинаются с **WM_**. Эта группа настолько велика, что ее уместно еще раз разбить на категории. Эти категории включают:

- сообщения DDE (dynamic data exchange);
- сообщения буфера обмена (clipboard);
- сообщения мыши;
- сообщения клавиатуры;
- сообщения неклиентской (non-client) области окна;
- сообщения MDI (multiple-document interface);
- и многие другие типы.

Перечисленные категории несколько неточны, не всегда строго определены; они просто служат для удобства программистов, чтобы можно было представить картину множества событий управления окнами. Множество сообщений **WM_** также не фиксировано, оно растет по мере добавления новых возможностей операционной системы.

Другие группы сообщений связаны с определенными типами окон. Существуют сообщения, определенные для полей ввода, кнопок, списков, комбинированных списков, полос прокрутки, элементов просмотра списка деревьев и т. д. Эти сообщения за редким исключением обычно обрабатываются оконной процедурой окна элемента управления и редко представляют интерес для программистов приложений.

Существует два способа введения программистом собственных сообщений.

Первый способ пригоден только для случая, если сообщение пересылается между окнами (частями) одного и того же приложения. Для этого в программе необходимо определить символическое имя нового сообщения при помощи директивы #define, например:

#define СимволическоеИмяОдногоСообщения (WM_USER+1),
#defineСимволическоеИмяДругогоСообщения (WM_USER+2).

Вторым способом определения собственного сообщения является использование функции RegisterWindowMessage, которая возвращает уникальный идентификатор для сообщения. Использование собственных типов сообщений, полученных таким способом, позволяет частям приложения связываться между собой; разные приложения также могут обмениваться информацией таким способом. Однако для взаимодействия приложений доступны более мощные механизмы, например, отображаемые в память файлы.

Итак, передача окну сообщений означает вызов операционной системой Windows оконной процедуры. Но в программах для Windows имеется и цикл обработки сообщений, который берет сообщения из очереди сообщений, вызывая функцию **GetMessage**, и отправляет их оконной процедуре, вызывая функцию **DispatchMessage**.

Одни и те же сообщения могут быть как синхронными (queued), так и асинхронными (nonqueued). Синхронными сообщениями называются сообще-

ния, которые Windows помещает в очередь сообщений программы и которые извлекаются и диспетчеризируются в цикле обработки сообщений. Асинхронные сообщения передаются непосредственно окну, когда Windows вызывает оконную процедуру. Говорят, что синхронные сообщения помещаются (отправляются) в очередь сообщений (post), а асинхронные посылаются прямо в оконную процедуру (send).

Структура программ для Windows очень проста, поскольку у них имеется только одно центральное место обработки сообщений. В результате оконная процедура получает все предназначенные для окна сообщения, как синхронные, так и асинхронные.

Синхронными становятся сообщения, в основном, тогда, когда они являются результатом пользовательского ввода путем нажатия клавиш (например, **WM_KEYDOWN** или **WM_KEYUP**), это символы, введенные с клавиатуры (**WM_CHAR**), результат движения мыши (**WM_MOUSEMOVE**) и щелчков кнопки мыши (**WM_LBUTTONDOWN**). Кроме этого, синхронные сообщения включают в себя сообщения таймера (**WM_TIMER**), сообщение о выходе из программы (**WM_QUIT**), а также все сообщения, которые помещаются в очередь сообщений посредством вызова приложением функции **PostMessage**.

Сообщения становятся асинхронными во всех остальных случаях. Часто асинхронные сообщения являются результатом вызова определенных функций Windows или непосредственным результатом вызова функции **SendMessage**.

Очевидно, что процесс этот сложен, но большая часть сложностей ложится на Windows, а не на пользовательские программы.

С позиции оконной процедуры совершенно не важно, для обработки какого сообщения и кем она вызвана. Оконная процедура может что-то сделать с этими сообщениями, а может и проигнорировать их. По этой причине оконную процедуру назвали конечным пунктом обработки (ultimate hook). Сообщения извещают оконную процедуру почти обо всем, что влияет на окно.

Например, когда **WinMain** вызывает функцию **CreateWindow**, Windows отправляет оконной процедуре асинхронное сообщение **WM_CREATE —** окно создано. Когда WinMain вызывает **UpdateWindow,** Windows отправляет оконной процедуре асинхронное сообщение **WM_PAINT —** изображение окна устарело, окно необходимо перерисовать.

Сообщения не похожи на аппаратные прерывания. Во время обработки в оконной процедуре одного сообщения программа не может быть прервана другим сообщением. Только в этом случае, если оконная процедура вызвала функцию, которая сама стала источником нового сообщения, то оконная процедура начнет обрабатывать это новое сообщение еще до того, как функция вернет управление программе.

Но как приложение узнает, что к нему пришло сообщение? Об этом рассказывается в следующем параграфе.

5.2.4. Цикл обработки сообщений

Непременным компонентом всех Windows-приложений является *цикл обработки сообщений*. У приложения всегда есть главная функция **WinMain**.

Обычно она содержит вызовы функций для инициализации и создания окон, после чего следует цикл обработки сообщений и необходимый код для закрытия приложения.

Что происходит в цикле обработки сообщений? Как известно, все сообщения, адресованные приложению, Windows записывает в очередь сообщений приложения. Извлечение сообщения из этой очереди осуществляет функция **GetMessage**.

Если очередное сообщение имеет код **WM_QUIT**, то происходит выход из цикла обработки сообщений, в результате чего приложение завершает свою работу.

Если очередное сообщение не является сообщением **WM_QUIT**, то оно передается функции **DispatchMessage**, которая возвращает сообщение обратно Windows. Далее Windows отправляет сообщение для его обработки соответствующей оконной процедуре, иными словами, Windows вызывает оконную процедуру. После возврата из оконной процедуры Windows передает управление оператору, который расположен после **DispatchMessage**, и работа цикла продолжается.

Цикл обработки сообщений и оконная процедура работают не параллельно. Когда оконная процедура обрабатывает сообщение, то это результат вызова функции **DispatchMessage** в главной функции **WinMain**. Функция **DispatchMessage** не завершается до тех пор, пока оконная процедура не обработала сообщение.

Часто возникает необходимость того, чтобы оконная функция сохраняла информацию, полученную в сообщении, и использовала ее при обработке другого сообщения. Тогда эту информацию следует описывать в оконной процедуре в виде статических переменных, либо хранить в глобальных переменных.

Контрольные вопросы

1. Какое назначение функции WinMain?
2. Какое назначение оконной процедуры?
3. Как Windows работает с системной очередью сообщений?
4. Поясните понятие функции обратного вызова.
5. Как моделируется структура «оконный класс»?
6. Какую информацию содержит структура сообщения?
7. Опишите назначение цикла обработки сообщений.

5.3. Примеры программ

5.3.1. Простейшая программа для Windows

Для создания программы необходимо выполнить следующие действия:

1. Запустить **CodeGear RAD Studio for Microsoft** Windows.
2. В меню **File** → **New** выбрать **VCL Form Application-C++Builder**, будет создан новый проект.
3. В меню **Project** выбрать пункт **Remove from Project** (Удалить из проекта).
4. Удалить **Unit1.cpp** из проекта.
5. В окне **File** выбрать **Project1.cpp**.

6. Появившийся в окне редактора текст исправить следующим образом:

```
#include <windows.h>
#include <vcl.h>
int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int CmdShow)
    {
        MessageBox(NULL,
                   “Для продолжения нажмите на кнопку”,
                   “Выполняется простейшая программа для Windows”,
                    MB_OK);
        return 0;
    }.
```

7. Запустить программу.

На экране появится окно, представленное на рисунке 5.3. Возможные действия с этой программой просты и обсуждаться не будут, а содержание самой программы стоит обсудить.

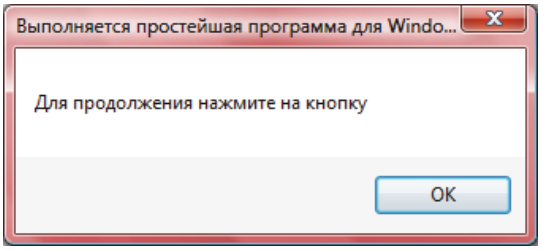


Рис. 5.3
Окно простейшей программы для Windows

В первых двух строках программы находятся инструкции

```
#include <windows.h>
#include <vcl.h>.
```

Эти директивы подключают к программе главный заголовочный файл Windows-приложений и заголовочный файл визуальных компонентов VCL системы программирования.

Далее следует заголовок функции **WinMain**

```
int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int CmdShow).
```

С вызова этой функции всегда начинается выполнение программы. Функция возвращает значение целого типа, а получает значения следующих параметров:

— **HINSTANCE hInstance** — дескриптор приложения, Windows автоматически присваивает ему значение при запуске;

— **HINSTANCE hPrevInstance** — дескриптор предыдущего экземпляра приложения, если одновременно выполняется несколько экземпляров приложения; этот параметр остался для совместимости с ранними версиями Windows и поэтому всегда принимает нулевое значение;

— **LPSTR lpCmdLine** — указатель на строку, в которую копируются параметры приложения, если оно запускается в режиме командной строки; для Windows такой способ запуска программы, скорее, является экзотикой, так как запуск программы щелчком мыши не предполагает передачи каких-либо параметров;

— **int CmdShow** — целое значение, которое может быть передано функции **ShowWindow**; этот параметр в нашем примере не используется и будет рассмотрен позже.

Спецификатор **WINAPI** определяет *соглашения о вызове*, т. е. принятый в Windows порядок передачи параметров в функцию. Для нашей дальнейшей работы этой информации достаточно, поэтому нужно запомнить, что этот спецификатор должен присутствовать всегда.

Разобравшись с заголовком функции, перейдем к ее телу. Здесь имеется вызов функции **MessageBox**, которая, собственно, и выполняет всю содержательную работу приложения.

Возникают вопросы. Где цикл обработки сообщений? Где текст оконной процедуры? Ответ на поставленные вопросы заключается в том, что написание такой лаконичной программы стало возможным только благодаря использованию функции **MessageBox**. Если поближе познакомиться с указанной функцией, то можно обнаружить, что функция **MessageBox** создает, отображает и обслуживает *окно сообщений*. Окно сообщений — это диалоговое окно, содержащее указанное программистом текстовое сообщение и одну или несколько кнопок. Заголовок функции имеет вид:

int MessageBox (HWND hWnd, LPCTSTR lpText,
LPCTSTR lpCaption, UINT uType).

Параметры функции интерпретируются следующим образом:

— **hWnd** — дескриптор родительского окна, он принимает значение **NULL**, если родительского окна нет;

— **lpText —** указатель на строку, содержащую текст сообщения;

— **lpCaption** — указатель на строку, содержащую текст заголовка диалогового окна;

— **uType** — параметр задает комбинацию флагов, задающих количество и типы кнопок в диалоговом окне, а также наличие заданной пиктограммы.

Сейчас для нас важно лишь то, что эта функция создает диалоговое окно, относящееся к стандартному оконному классу диалоговых окон. А каждый стандартный оконный класс диалоговых окон содержит свою оконную процедуру, спрятанную в недрах Windows. Кроме того, выполняя функцию MessageBox, система создает невидимый для программиста цикл обработки сообщений, который обслуживает созданное окно диалога.

Вот почему в тексте нашей программы нет ни оконной процедуры, ни цикла обработки сообщений, и она претендует на титул самой короткой программы для Windows.

5.3.2. Полноценная программа для Windows

Полноценная программа для Windows должна содержать как минимум две функции:

— **WinMain** — главную функцию, в которой создается основное окно программы и запускается цикл обработки сообщений;

— **WndProc** — оконную процедуру, обеспечивающую обработку сообщений для основного окна программы.

Основное назначение функции **WinMain** заключается в следующем.

Шаг 1. Подготовить и зарегистрировать класс окна с требуемыми характеристиками.

Шаг 2. Создать экземпляр окна зарегистрированного класса.

Шаг 3. Пока не произошло завершающее программу событие выполнять:

— извлечь очередное сообщение из очереди сообщений;

— передать через Windows очередное сообщение оконной функции.

Шаг 4. Завершить программу.

Основное назначение оконной процедуры **WndProc** заключается в обработке очередного сообщения.

Для подготовки к написанию программы необходимо выполнить действия, приведенные в начале первого примера настоящего параграфа, и ввести следующий текст.

```
#include <vcl.h>
#include <tchar.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
// Главная функция — точка входа в программу----------------------------------
int WINAPI _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nCmdShow)
{
     HWND hMainWnd;
     wchar_t * sClassName=L»MyClass»;
              MSG msg;
// Структура класса окна
     WNDCLASSEX wc;
     wc.cbSize      =sizeof(wc);
     wc.style       =CS_HREDRAW | CS_VREDRAW;
     wc.lpfnWndProc  =WndProc;
     wc.cbClsExtra   =0;
     wc.cbWndExtra   =0;
     wc.hInstance    =hInstance;
     wc.hIcon        =LoadIcon(NULL, IDI_APPLICATION);
     wc.hCursor      =LoadCursor(NULL, IDC_ARROW);
     wc.hbrBackground=(HBRUSH) GetStockObject(WHITE_BRUSH);
     wc.lpszMenuName =NULL;
```

```
    wc.lpszClassName=sClassName;
    wc.hIconSm     =LoadIcon (NULL, IDI_APPLICATION);
// Регистрируем класс окна
    if (!RegisterClassEx(&wc))
     {
      MessageBox(NULL, L«Класс окна не зарегистрирован», L«Ошибка», MB_OK);
      return 0;
     };
// Создаем основное окно приложения
     hMainWnd=CreateWindow(
                          sClassName,
                          L«Выполняется полноценное приложение Windows»,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          0,
                          CW_USEDEFAULT,
                          0,
                          (HWND)NULL,
                          (HMENU)NULL,
                          (HINSTANCE)hInstance,
                          NULL
                           );
    if (!hMainWnd)
      {
       MessageBox(NULL, L«Главное окно не создано», L«Ошибка», MB_OK);
       return 0;
      };
// Показываем созданное окно
    ShowWindow (hMainWnd, nCmdShow);
// Выполняем цикл обработки сообщений до закрытия приложения
    while (GetMessage (&msg, NULL, 0, 0))
        {
          TranslateMessage (&msg);
          DispatchMessage (&msg);
        };
    return msg.wParam;
}

// Оконная процедура ----------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg,
                                    WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    RECT rect;
    switch (uMsg)
      {
       case WM_PAINT:
```

```
            hDC=BeginPaint (hWnd, &ps);
            GetClientRect (hWnd,&rect);
            DrawText (hDC, L«Сообщения обрабатываются!», -1, &rect,
                                    DT_SINGLELINE | DT_CENTER | DT_VCENTER);
            EndPaint (hWnd, &ps);
            break;
        case WM_CLOSE:
            DestroyWindow (hWnd);
            break;
        case WM_DESTROY:
            PostQuitMessage (0);
            break;
        default:  return DefWindowProc(hWnd, uMsg, wParam, lParam);
        };
    return 0;
}
```

После компиляции и запуска программы на экране появится окно, показанное на рисунке 5.4.

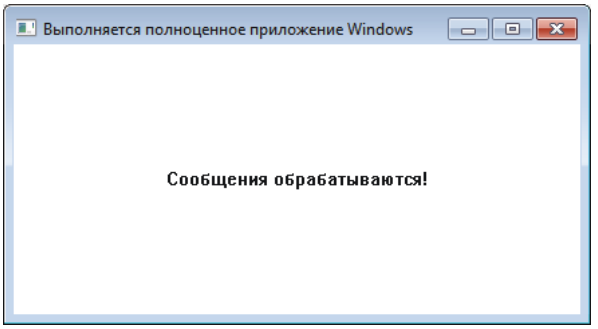


Рис. 5.4
Полноценная программа для Windows

Обсудим теперь подробно содержание программы.

В программе имеется две функции: **_tWinMain** и **WndProc**. Функция **_tWinMain** — это точка входа в программу. Функция **WndProc** — это оконная процедура для главного окна программы. Оконная процедура инкапсулирует код, отвечающий за ввод информации, обычно осуществляемый при помощи клавиатуры или мыши, а также — за вывод информации на экран.

В программе отсутствуют инструкции для непосредственного вызова **WndProc**, так как оконная процедура вызывается только из Windows. Однако в функции **_tWinMain** имеется ссылка на **WndProc** (адрес оконной процедуры присваивается полю **wc.lpfnWndProc**), поэтому прототип функции **WndProc** объявлен в самом начале программного файла, еще до описания функции **_tWinMain**.

Обратим внимание на идентификаторы, написанные полностью прописными буквами. Эти идентификаторы задаются в заголовочных файлах Windows. Некоторые из них содержат двух- или трехбуквенный префикс, завершаемый

символом подчеркивания, например, **CS_HREDRAW**, **IDI_APPLICATION**, **IDC_ARROW**, **WS_OVERLAPPEDWINDOW**, **CW_USEDEFAULT**. Это просто числовые константы. Некоторые из этих категорий приведены в таблице 5.2.

Таблица 5.2

Префиксы для числовых констант

Префикс	Категория
CS_	Опция стиля класса
CW_	Опция создания окна
DT_	Опция рисования окна
IDC_	Идентификатор предопределенного курсора
IDI_	Идентификатор предопределенной иконки
WM_	Сообщение окна
WS_	Стиль окна

Также следует обратить внимание на типы данных, специфичные для системы Windows. Некоторые из них были приведены в таблице 5.1.

Регистрация класса окна. Сразу после входа в функцию **WinMain** создается и регистрируется класс главного окна приложения. Для этого необходимо заполнить структуру типа **WNDCLASSEX**, а затем передать адрес этой структуры в виде аргумента функции **RegisterClassEx**.

Структура **WNDCLASSEX** имеет двенадцать полей:

```
typedef struct tagWNDCLASSEX
{
  UINT cbSize;                  // размер данной структуры в байтах
  UINT style;                   // стиль класса окна
  WNDPROC lpfnWndProc; // указатель на функцию окна (оконную процедуру)
  int cbClsExtra;              // число дополнительных байтов, которые должны быть
 // распределены в конце структуры класса
  int cbWndExtra;              // число дополнительных байтов, которые должны быть
 // распределены вслед за экземпляром окна
  HINSTANCE hInstance; // дескриптор экземпляра приложения, в котором нахо-
дится
 // оконная процедура для этого класса
  HICON hIcon;                  // дескриптор пиктограммы
  HCURSOR hCursor;          // дескриптор курсора
  HBRUSH hbrBackground;  // дексриптор кисти, используемой для закраски фона
окна
  LPCTSTR lpszMenuName; // указатель на строку, содержащую имя меню, приме-
няемого
 // по умолчанию для этого класса
  LPCTSTR lpszClassName;     // указатель на строку, содержащую имя класса окна
  HICON hIconSm;                // дескриптор малой пиктограммы
} WNDCLASSEX;
```

Рассмотрим назначение полей структуры **WNDCLASSEX**.

Значение первого поля, **cbSize**, должно быть равно длине структуры.

Второе поле, **style**, может содержать в своем значении один или несколько стилей, наиболее употребительные стили перечислены в таблице 5.3.

Таблица 5.3

Стили класса окна

Стиль	Описание
CS_GLOBALCLASS	Создать класс, доступный всем приложениям. Обычно этот стиль применяется для создания определяемых пользователем элементов управления в DLL
CS_HREDRAW	Перерисовывать все окно, если изменен размер по горизонтали
CS_NOCLOSE	Запретить команду Close в системном меню
CS_OWNDC	Выделить уникальный контекст устройства для каждого окна, созданного при помощи этого класса
CS_VREDRAW	Перерисовывать все окно, если изменен размер по вертикали

В заголовочных файлах Windows идентификаторы, начинающиеся с префикса **CS_**, задаются в виде 32-разрядной константы, в которой только один разряд установлен в единичное состояние. Например, константа **CS_VREDRAW** задана как 0×0001, а **CS_HREDRAW** — как 0×0002. Подобные константы могут объединяться с помощью операции ИЛИ языка C++.

В рассматриваемой программе используется комбинация стилей **CS_VREDRAW** | **CS_HREDRAW**. Это означает, что все окна этого класса должны целиком перерисовываться при изменении как горизонтального, так и вертикального размеров окна. Если попробовать изменить размеры окна «Выполняется полноценное приложение Windows», то можно увидеть, что строка текста «Сообщения обрабатываются!» переместится в новый центр окна. Механизм уведомления оконной процедуры об изменении размеров окна будет рассмотрен позже.

Третье поле, **lpfnWndProc**, содержит адрес оконной процедуры (в нашей программе это **WndProc**).

Четвертое, **cbClsExtra**, и пятое, **cbWndExtra**, поля используются крайне редко, а назначение шестого поля, **hInstance**, очевидно, поэтому их обсуждать не будем.

Седьмое поле, **hIcon**, содержит дескриптор пиктограммы, которая предназначена для этого класса окна. Значение **hIcon** обычно получают вызовом функции **LoadIcon**, которая имеет следующий прототип:

HICON LoadIcon (HINSTANCE hInstance, LPCTSTR lpIconName).

Эта функция загружает ресурс пиктограммы, заданной параметром **lpIconName**, из экземпляра приложения, указанного параметром **hInstance**. Функцию можно использовать также для загрузки одной из системных пиктограмм, если передать первому аргументу значение **NULL**. В этом случае второй аргумент должен содержать константу, идентификатор которой начинается с префикса **IDI_** («идентификатор значка» — ID for icon). Возможные значения второго аргумента для системных пиктограмм приведены в таблице 5.4.

Таблица 5.4

Идентификаторы системных пиктограмм

Значение	Описание
IDI_APPLICATION	Пиктограмма приложения по умолчанию
IDI_ASTERISK	То же, что и **IDI_INFORMATION**

Продолжение табл. 5.4

Значение	Описание
IDI_ERROR	Пиктограмма в виде белого креста на фоне красного круга. Используется в серьезных предупреждающих сообщениях
IDI_EXCLAMATION	Тог же, что и **IDI_WARNING**
IDI_HAND	Тог же, что и **IDI_ERROR**
IDI_INFORMATION	Пиктограмма «i», которая используется в информационных сообщениях
IDI_QUESTION	Пиктограмма «?»
IDI_WARNING	Пиктограмма «!», которая используется в предупреждающих сообщениях
IDI_WINLOGO	Логотип Windows

Восьмое поле, **hCursor**, содержит дескриптор курсора мыши, используемого приложением в клиентской области окна. Значение **hCursor** обычно получают с помощью вызова функции **LoadCursor**, имеющей следующий прототип:

HCURSOR LoadCursor (HINSTANCE hInstance, LPCTSTR lpCursorName).

Эта функция загружает ресурс курсора, заданный вторым параметром (**lpCursorName**), из экземпляра приложения, заданного первым параметром (**hInstance**).

Функцию можно также использовать для загрузки одного из системных курсоров, если передать первому аргументу значение **NULL**. Значение второго аргумента при этом выбирается из таблицы 5.5.

Таблица 5.5

Идентификаторы курсора

Значение	Описание
IDC_APPSTARTING	Стандартная стрелка и малые песочные часы
IDC_ARROW	Стандартная стрелка
IDC_CROSS	Перекрестье
IDC_HELP	Стрелка и вопросительный знак
IDC_IBEAM	Текстовый двутавр
IDC_NO	Перечеркнутый кружок
IDC_SIZEALL	Четырехконечная стрелка
IDC_SIZENESW	Двухконечная стрелка, указывающая на северо-восток и юго-запад
IDC_SIZENS	Двухконечная стрелка, указывающая на север и юг
IDC_SIZENWSE	Двухконечная стрелка, указывающая на северо-запад и юго-восток
IDC_SIZEWE	Двухконечная стрелка, указывающая на запад и восток
IDC_UPARROW	Вертикальная стрелка
IDC_WAIT	Песочные часы

Девятое поле, **hbrBackground**, содержит дескриптор кисти, используемой приложением для закраски фона и клиентской области окна. *Кисть* (brush) — это графический объект, который представляет собой шаблон пикселов различных цветов, используемый для закрашивания области. В Windows имеется несколько стандартных кистей. Вызов функции **GetStockObject** с аргументом **WHITE_BRUSH** возвращает дескриптор белой кисти. Возвращаемое значение имеет тип **HGDIOBJ**, поэтому необходимо преобразование его к типу **HBRUSH**.

Десятое поле, **lpszMenuName**, указывает на строку, содержащую имя меню, применяемого по умолчанию для данного класса. Рассматриваемая программа не имеет меню, поэтому поле установлено в **NULL**.

Одиннадцатое поле, **lpszClassName**, указывет на строку, содержащую имя класса. Это имя должно использоваться в параметре **lpClassName** функции **CreateWindow**.

Двенадцатое поле, **hIconSm**, содержит дескриптор малой пиктограммы, которая предназначена для использования в строке заголовка окон, созданных при помощи этого класса. Размеры пиктограммы должны быть 16×16 пикселов. Если поле равно **NULL**, то система ищет ресурс пиктограммы, указанный полем **hIcon**, чтобы сформировать малую пиктограмму из него.

Итак, с заполнением структуры класса окна мы разобрались. Осталось зарегистрировать подготовленный класс, вызвав функцию **RegisterClassEx**, передав ей параметром структуру **WNDCLASSEX**[20].

Создание окна. Если регистрация класса окна прошла успешно, то следующий этап — создание окна. Для этого вызывается функция **CreateWindow**, прототип которой приведен ниже:

```
HWND CreateWindow (
LPCTSTR lpClassName,      // имя зарегистрированного класса
LPCTSTR lpWindowName,  // заголовок окна
DWORD dwStyle,              //стиль окна
int x,                            //горизонтальная позиция окна
int y,                            //вертикальная позиция окна
int nWidth,                    //ширина окна
int Height,                    //высота окна
HWND hWndParent,         //дескриптор родительского окна
HMENU hMenu,              //дескриптор меню или идентификатор элемента управления
HINSTANCE hInstance,      //дескриптор экземпляра приложения
LPVOID lParam     //указатель на данные, передаваемые в сообщении WM_CREATE
                  ).
```

Рассматривая назначение параметров, не будем забывать, что функция **CreateWindow** позволяет создавать не только основное окно приложения, но и любые другие окна, включая окна стантартных классов, с помощью которых реализуются элементы управления Windows.

Первый параметр, **lpClassName**, — это указатель на строку, содержащую допустимое имя класса. Таким именем может быть либо имя класса, зарегистрированное ранее при помощи функции **RegisterClassEx** или **RegisterClass**, либо имя одного из стандартных классов, перечисленных в таблице 5.6.

Второй параметр, **lpWindowName**, — это указатель на строку, содержащую заголовок окна. Место отображения заголовка зависит от вида окна. Например, для главного окна приложения оно выводится как заголовок окна, а для окна стандартного класса **BUTTON** размещается по центру кнопки.

[20] Это расширенная версия (**Ex** — от extended — расширенный) функции **RegisterClass** из предыдущих версий Windows.

Таблица 5.6

Стандартные классы окон

Класс	Описание	Префикс в обозначении стилей
BUTTON	Прямоугольное окно кнопки, группы, флажка, переключателя или пиктограммы	**BS_**
COMBOBOX	Элемент управления, объединяющий элементы **LISTBOX** и **EDIT**. В поле редактирования отображается выбранная строка из **LISTBOX**	**CBS_**
EDIT	Прямоугольное окно (окно редактирования), предназначенное для ввода текста с клавиатуры	**ES_**
LISTBOX	Прямоугольное окно со списком строк, из которого пользователь может выбрать любую строку	**LBS_**
MDICLIENT	Клиентское окно многодокументного интерфейса. Это окно получает сообщения, которые управляют дочерними окнами в MDI-приложении	
RICHEDIT	Элемент управления Rich Edit версии 1.0. В дополнение к возможностям элемента **EDIT** позволяет редактировать текст с разными шрифтами и стилями	**ES_**
RICHEDITCLASS	Элемент управления Rich Edit версий 2.0, 3.0. Усовершенствованная версия элемента **RICHEDIT** с дополнительными возможностями	**ES_**
SCROLLBAR	Элемент управления линейкой прокрутки	**SBS_**
STATIC	Элемент управления статическим текстом. Применяется для размещения в окне текста или рамок	**SS_**

Третий параметр, **dwStyle**, позволяет указывать стиль окна, состоящий из значений, указанных в таблице 5.7.

Таблица 5.7

Стили окна

Обозначение стиля	Описание
WS_BORDER	Создать окно с рамкой в виде тонкой линии
WS_CAPTION	Создать окно, которое имеет область заголовка (включает стиль **WS_BORDER**)
WS_CHILD	Создать дочернее окно. Окно этого стиля не может иметь полосу меню и не может иметь стиль **WS_POPUP**
WS_CLIPCHILDREN	Исключить перерисовку дочерних окон, принадлежащих данному родительскому окну
WS_CLIPSIBLINGS	Исключить перерисовку соседних дочерних окон при перерисовке данного дочернего окна
WS_DLGFRAME	Создать окно, имеющее рамку со стилем, типичным для диалоговых окон. Окно этого стиля не может иметь область заголовка
WS_GROUP	Считать данное окно первым в группе элементов управления (обычно группируются переключатели)
WS_HSCROLL	Создать окно с горизонтальной линейкой прокрутки
WS_MAXIMIZE	Создать окно, которое первоначально является развернутым
WS_MAXIMIZEBOX	Создать окно с кнопкой развертывания
WS_MINIMIZE	Создать окно, которое первоначально является свернутым

Продолжение табл. 5.7

Обозначение стиля	Описание
WS_MINIMIZEBOX	Создать окно с кнопкой свертывания
WS_OVERLAPPED	Создать перекрывающееся окно. Окно этого стиля имеет область заголовка и рамку
WS_OVERLAPPEDWINDOW	Сочетание стилей **WS_OVERLAPPED**, **WS_CAPTION**, **WS_SYSMENU**, **WS_THICKFRAME**, **WS_MINIMIZEBOX** и **WS_MAXIMIZEBOX**
WS_POPUP	Создать всплывающее окно. Этот стиль не может использоваться со стилем **WS_CHILD**
WS_POPUPWINDOW	Сочетание стилей **WS_BORDER**, **WS_POPUP** и **WS_SYSMENU**. Чтобы сделать системное окно видимым, необходимо добавить стиль **WS_CAPTION**
WS_SYSMENU	Создать окно с системным меню в области заголовка
WS_TABSTOP	Стиль указывает, что на окне может остановиться фокус ввода, перемещаемый при помощи клавиши TAB
WS_THICKFRAME	Создать окно с рамкой, которая позволяет изменять его размеры
WS_VISIBLE	Создать окно, которое сразу же является видимым
WS_VSCROLL	Создать окно с вертикальной линейкой прокрутки

В рассматриваемой программе используется стандартный стиль для главного окна приложения **WS_OVERLAPPEDWINDOW**. Это обычное перекрывающееся окно с заголовком, системным меню, расположенным в левой части строки заголовка, кнопками для сворачивания, разворачивания и закрытия окна справа на строке заголовка и «толстой» рамкой, позволяющей изменять размеры окна.

Четвертый параметр, ***x***, задает горизонтальную позицию левого верхнего угла окна. Для главного окна позиции ***x*** и ***y*** определяются в *экранных координатах*, а для дочерних окон и элементов управления координаты отсчитываются относительно левого верхнего угла родительского окна.

Если позиция ***x*** не важна, то можно установить значение **CW_USEDEFAULT**. В этом случае Windows использует значения ***x*** и ***y*** по умолчанию. Это означает, что система располагает следующие друг за другом перекрывающиеся окна, равномерно отступая по горизонтали и вертикали от левого верхнего угла экрана.

Пятый параметр, ***y***, задает вертикальную позицию левого верхнего угла экрана. Если параметр ***x*** имеет значение **CW_USEDEFAULT**, то значение параметра **y** игнорируется.

Шестой параметр, **nWidth**, задает ширину окна в пикселах. Если ему присвоено значение **CW_USEDEFAULT**, то система будет использовать значения **nWidth** и **nHight** по умолчанию.

Седьмой параметр, **nHight**, задает высоту окна в пикселах. Если параметр **nWidth** установлен в **CW_USEDEFAULT**, то значение **nHight** игнорируется.

Восьмой параметр, **nWndParent**, в рассматриваемой программе имеет значение **NULL**, поскольку у главного окна программы отсутствует родительское окно. Заметим, что, если между двумя окнами существует связь типа *родительское — дочернее*, дочернее окно всегда появляется только на поверхности родительского.

Девятый параметр, **hMenu**, содержит дескриптор меню окна или идентификатор элемента управления. Интерпретация значения параметра зависит от вида окна.

Если приложение использует меню, определенное в классе окна (поле **lpszMenuName** в структуре **WNDCLASSEX**), то параметру **hMenu** необходимо присвоить значение **NULL**.

Если создаваемое окно относится к элементу управления, то параметру **hMenu** передается целочисленное значение, которое далее используется как идентификатор созданного элемента. Этот идентификатор, например, будет содержаться в сообщениях **WM_COMMAND**, поступающих от элемента управления.

Десятому параметру, **hInstance**, должен быть присвоен дескриптор экземпляра приложения, переданный программе в качестве аргумента функции **_tWinMain**.

Последний параметр, **lParam**, в рассматриваемом примере тоже имеет значение **NULL**. При необходимости он может быть использован в качестве указателя на дополнительные данные, передаваемые окну в момент его создания с помощью сообщения **WM_CREATE**.

Итак, функция **CreateWindow** вызвана. Отработав, она возвращает дескриптор созданного окна. Если по какой-то причине создать окно не удалось, то функция возвращает значение **NULL**. Поэтому рекомендуется проверять возвращаемое функцией значение и в случае неудачи выдавать соответствующее сообщение, используя уже знакомую функцию **MessageBox**.

Использование функции CreateWindowEx. Для создания окна вместо функции **CreateWindow** может быть вызвана функция **CreateWindowEx**, прототип которой приведен ниже:

```
HWND CreateWindowEx (
DWORD dwExStyle,
LPCTSTR lpClassName,
LPCTSTR lpWindowName,
DWORD dwStyle,
int x,
int y,
int nWidth,
int nHeight,
HWND hWndParent,
HMENU hMenu,
HINSTANCE hInstance,
LPVOID lParam
  ).
```

Все параметры этой функции, кроме первого, имеют тот же смысл, что и у функции **CreateWindow**. Первый параметр, **dwExStyle**, задает расширенный стиль окна, применяемый совместно со стилем, определенным в параметре **dwStyle**. Например, в качестве расширенного стиля можно задать один или несколько флагов, приведенных в таблице 5.8.

Таблица 5.8

Расширенные стили окна

Стиль	Описание
WS_EX_ACCEPTFILES	Создать окно, которое принимает перетаскиваемые файлы
WS_EX_CLIENTEDGE	Рамка окна имеет утопленный край
WS_EX_CONTROLPARENT	Разрешить пользователю перемещаться по дочерним окнам с помощью клавиши Tab
WS_EX_MDICHILD	Создать дочернее окно многодокументного интерфейса
WS_EX_STATICEDGE	Создать окно с трехмерной рамкой. Этот стиль предназначен для элементов, которые не принимают ввод от пользователя
WS_EX_TOOLWINDOW	Создать окно с инструментами, предназначенное для реализации плавающих панелей инструментов
WS_EX_TRANSPARENT	Создать прозрачное окно. Любые окна того же уровня, накрываемые этим окном, получат сообщение **WM_PARWNT** в первую очередь, тем самым создавая эффект прозрачности
WS_EX_WINDOWEDGE	Создать окно, имеющее рамку с активированным краем

Функция **CreateWindowEx** так же, как и функция **CreateWindow**, может быть применена для создания любых окон, в том числе и окон элементов управления Windows.

Отображение окна на экране. Для отображения на экране созданного окна вызывается функция **ShowWindow**, имеющая следующий прототип:

BOOL ShowWindow (HWND hWnd, int nCmdShow).

Первым параметром функции является дескриптор окна, а второй параметр определяет, в каком виде будет показано окно.

При начальном отображении главного окна рекомендуется присваивать второму параметру то значение, которое передается приложению через параметр **nCmdShow** функции **WinMain**.

При последующих отображениях можно использовать любое из значений, приведенных в таблице 5.9.

Таблица 5.9

Часто используемые значения параметра nCmdShow функции ShowWindow

Значение	Описание
SW_HIDE	Скрыть окно
SW_MAXIMIZE	Развернуть окно
SW_MINIMIZE	Свернуть окно
SW_SHOW	Активизировать окно и показать в его текущих размерах и позиции
SW_SHOWMINNOACTIVE	Окно не выводится, а на панели задач появляются его имя и пиктограмма

Обработка сообщений. После вывода на экран программа должна подготовить себя для получения информации от пользователя. Эта информация обычно вводится с помощью клавиатуры или мыши. Windows поддерживает *очередь сообщений* (message queue) для каждой программы, работающей в системе. Любое действие пользователя система Windows преобразует в сообщение, которое помещается в указанную очередь.

Программа извлекает сообщения из очереди, выполняя блок команд, известный как *цикл обработки сообщений* (message loop):

```
while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
return msg.wParam;
```

В этом фрагменте кода переменная **msg** — это структура типа **MSG**, которая определена в заголовочных файлах Windows следующим образом:

```
typedef struct tagMSG
{
      HWND hwnd;              // дескриптор окна, которому адресовано сообщение
      UINT message;           // номер (идентификатор) сообщения
      WPARAM wParam;          // параметр сообщения wParam
      LPARAM lParam; // параметр сообщения lParam
      DWORD time;             // время отправки сообщения
      POINT pt;          // позиция курсора (в экранных координатах) в момент отправки
                                 //          сообщения
} MSG;
```

Интерпретация параметров **wParam** и **lParam** зависит от номера сообщения. Тип данных **POINT** используется для представления точки парой координат:

```
typedef struct tagPoint
   {
      LONG x;
      LONG y;
   } POINT;
```

Извлечение очередного сообщения из *очереди сообщений* осуществляется с помощью функции **GetMessage**, имеющей прототип:

BOOL GetMessage (LPMSG lpMsg, HWND hWnd, UINT wMsgFilterMin, UINT wMsgFilterMax),

где

— параметр **lpMsg** задает адрес структуры типа **MSG**, в которую помещается выбранное сообщение;

— параметр **hWnd** содержит дескриптор окна, принимающего сообщение. Обычно значение этого параметра равно **NULL**, что позволяет выбрать сообщения для любого окна приложения;

— параметр **wMsgFilterMin** указывает минимальный номер принимаемого сообщения. Обычно он имеет нулевое значение;

— параметр **wMsgFilterMax** задает максимальный номер принимаемого сообщения и тоже обычно имеет нулевое значение. Если оба последних параметра равны нулю, то функция выбирает из очереди любое очередное сообщение.

Функция **GetMessage** возвращает значение **TRUE** при извлечении любого сообщения, кроме одного — **WM_QUIT**. Получив сообщение **WM_QUIT**, функция возвращает значение **FALSE**. В результате этого происходит немедленный выход из цикла, и приложение завершает работу, возвращая операционной системе код возврата **msg.wParam**.

Но какова дальнейшая судьба принятого сообщения?

В теле цикла обработки сообщений можно увидеть вызов двух функций: **TranslateMessage** и **DispatchMessage**.

Строго говоря, вызов **TranslateMessage** нужен только в тех приложениях, которые должны обрабатывать ввод данных с клавиатуры. Дело в том, что для обеспечения независимости от аппаратных платформ и различных национальных раскладок клавиатуры в Windows реализована двухуровневая схема обработки сообщений от символьных клавиш. Сначала система генерирует сообщения о так называемых *виртуальных клавишаях*, например: сообщение **WM_KEYDOWN** — когда клавиша нажимается, и сообщение **WM_KEYUP** — когда клавиша отпускается. В сообщении **WM_KEYDOWN** содержится также идентифицирующая нажатую клавишу информация, которая зависит от аппаратной платформы.

Функция **TranslateMessage** преобразует пару сообщений **WM_KEYDOWN** и **WM_KEYUP** в сообщение **WM_CHAR**, которое содержит код символа (**wParam**) в вид значения типа **TCHAR**[21]. Сообщение **WM_CHAR** помещается в очередь, а на следующей итерации цикла функция **GetMessage** извлекает его для последующей обработки.

Наконец, функция **DispatchMessage** передает структуру **msg** обратно в Windows. Очередной ход за Windows. Windows отправляет сообщение для его обработки соответствующей оконной процедуре, вызывая ее как *функцию обратного вызова*.

Следует обратить внимание на то, что сообщение передается оконной процедуре именно того окна, которому оно было адресовано. Это может быть или **WndProc** (если сообщение адресовано главному окну), или оконная процедура некоторого диалогового окна, или оконная процедура одного из стандартных элементов управления, спрятанная в недрах Windows (т. е. в DLL).

После возврата из оконной процедуры Windows передает управление оператору, указанному за вызовом **DispatchMessage**, т. е на начало цикла **while**, и работа цикла продолжается.

Как уже говорилось ранее, цикл обработки сообщений функционирует до тех пор, пока из очереди не поступит сообщение **WM_QUIT**. Появление этого сообщения вызывает немедленный выход из цикла и прекращение работы программы.

Оконная процедура. Реальная работа приложения осуществляется в *оконной процедуре* (window procedure). Оконная процедура определяет то, что выводится в клиентскую область окна, и то, как окну реагировать на пользовательский ввод.

[21] Тип **TCHAR** интерпретируется как **char**, если макрос **UNICODE** не определен. В ином случае он интерпретируется как **wchar_t** (Unicode — символ).

Заголовок оконной процедуры всегда имеет следующий вид:

LRESULT CALLBACK WndProc (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam).

Четыре параметра оконной процедуры идентичны первым четырем полям структуры **MSG**. Первый параметр функции содержит дескриптор окна, получающего сообщение. Во втором параметре указывается идентификатор сообщения. Для системных сообщений в Windows зарезервированы номера от 0 до 1024. Третий и четвертый параметры содержат дополнительную информацию, которая распознается системой в зависимости от типа полученного сообщения.

Вместо имени **WndProc** можно использовать любое другое имя, но оно должно совпадать со значением поля **lpfnWndProc** структуры **wc**.

В рассматриваемой программе имеется только одно окно класса **MyClass**. Дескриптор окна **hMainWnd** получен вызовом функции **CreateWindow**. Поэтому при вызове оконной процедуры ее параметру **hWnd** всегда передается значение **hMainWnd**. Если же в программе создается несколько окон на основе одного и того же класса окна (и, следовательно, имеющих одну и ту же оконную процедуру), тогда параметру **hWnd** будет передаваться дескриптор конкретного окна, получающего сообщение.

Обычно программисты используют оператор **switch** для определения того, какое сообщение получено и как обрабатывать полученное сообщение. Если сообщение обрабатывается, то оконная процедура обязана вернуть нулевое значение. Все сообщения, не обрабатываемые оконной процедурой, должны передаваться системной функции **DefWindowProc**. В этом случае оконная процедура должна вернуть то значение, которое возвращает **DefWindowProc**.

Таким образом, программист пишет код только для тех сообщений, которые нуждаются в нестандартной обработке. В рассмотренной программе таких сообщений всего три: **WM_PAINT**, **WM_CLOSE**, **WM_DESTROY**.

Обработка сообщения WM_PAINT. Обработка сообщения **WM_PAINT** крайне важна для программирования под управлением Windows. Это сообщение уведомляет программу, что часть или вся клиентская область окна *недействительна* (invalid) и ее необходимо перерисовать. В каких ситуациях клиентская область становится недействительной? Рассмотрим некоторые случаи.

Очевидно, что при создании окна недействительна вся его клиентская область, поскольку в ней еще ничего не нарисовано. Это первый случай.

Если мы меняем размеры окна, клиентская область также объявляется системой недействительной. Это второй случай.

Когда мы минимизируем окно программы, сворачивая его на панель задач, а затем снова разворачиваем окно до начального размера, то Windows объявляет клиентскую область окна недействительной. Это третий случай.

Если мы перемещаем окна так, что они перекрываются, а затем закрытая часть окна вновь открывается, то Windows помечает требующую восстановления клиентскую область как недействительную. Это четвертый случай.

Список примеров недействительной клиентской области можно продолжить, но ограничимся перечисленными случаями.

Во всех перечисленных случаях, кроме первого, операционная система автоматически помечает клиентскую область окна как недействительную, что влечет за собой посылку сообщения **WM_PAINT**. В первом случае, когда окно только что было создано, аналогичный эффект достигается вызовом функции **ShowWindow**, вызываемой в функции **_tWinMain**. В процессе ее выполнения генерируются сообщения **WM_SIZE** и **WM_MOVE**, а обрабатывая **WM_SIZE**, система автоматически генерирует сообщение **WM_PAINT**.

Вернемся к анализу нашей программы. Обработку сообщения **WM_PAINT** рекомендуется всегда начинать с вызова функции **BeginPaint**:

hDC=BeginPaint (hWnd, &pc);

Первый параметр функции содержит дескриптор окна, полученный через аргумент оконной процедуры, а второй — адрес структуры **pc** типа **PAINTSTRUCT**. Поля этой структуры, заполняемые в результате выполнения функции **BeginPaint**, в дальнейшем используются операционной системой. Для нас сейчас более важно возвращаемое функцией значение — это дескриптор так называемого контекста устройства.

Контекст устройства (device context) описывает физическое устройство вывода информации, например дисплей или принтер. Этот важнейший объект графической подсистемы Windows представлен внутренней структурой данных, сохраняющей часто используемые графические атрибуты, такие как цвет фона, *перо*, *кисть*, *шрифт* и им подобные параметры. Эти атрибуты используются при вызове всех рисующих функций, получающих дескриптор **hDC** в качестве параметра.

Прежде чем перейти к следующей строке программного текста, отметим два побочных эффекта выполнения функции **BeginPaint**.

Во-первых, обрабатывая вызов **BeginPaint**, система Windows обновляет фон клиентской области, если обновляемый *регион* помечен для стирания[22]. По умолчанию для этого используется *кисть*, заданная в поле **hbrBackGround** структуры **WNDCLASSEX**.

Во-вторых, вызов функции **BeginPaint** делает всю клиентскую область действительной (valid), т. е. не требующей перерисовки. Это предотвращает повторную генерацию системой сообщения **WM_PAINT** до тех пор, пока вновь не произойдет одно из событий, требующих перерисовки окна.

После вызова **BeginPaint** в рассматриваемой программе следует вызов функции **GetClientRect**, предназначенной для получения размеров клиентской области окна:

GetClientRect (hWnd, &rect);

Результат работы функции помещается в переменную **rect** типа **RECT**. Структура **RECT**, описывающая прямоугольник (rectangle), определена в заголовочных файлах Windows следующим образом:

[22] Информация об этом факте хранится в одном из полей структуры ps.

```
typedef struct tagRECT
{
    LONG left;
    LONG top;
    LONG  right;
    LONG bottom;
} RECT;
```

Поля этой структуры задают координаты левого верхнего угла (**left**, **top**) и правого нижнего угла (**right**, **bottom**) прямоугольника.

После выполнения функции **GetClientRect** поля **left** и **top** всегда получают нулевые значения, а поля **right** и **bottom** содержат ширину и высоту клиентской области в пикселах. Информация о размерах клиентской области, сохраненная в переменной **rect**, используется далее в «рисующей» функции **DrawText**.

В нашей программе все рисование сводится к выводу текста «**Сообщения обрабатываются!**» при помощи функции **DrawText**, имеющей следующий прототип:

```
BOOL DrawText
(
   HDC hdc,              // дескриптор контекста устройства
   LPCTSTR lpString,     // указатель на символьную строку
   int nCount,           // длина текста
   LRECT lpRect, // указатель на ограничивающий прямоугольник
   UINT uFormat // флаги форматирования текста
);
```

Функция выводит текст из строки **lpString** в прямоугольную область, заданную структурой типа **RECT**, используя метод форматирования, заданный параметром **uFormat**. Количество символов в выводимой строке задается параметром **nCount**. Если установить значение **nCount** в **–1**, то система сама определит длину строки **lpString** по завершающему нулевому символу. Но в этом случае программист должен позаботиться о том, чтобы строка действительно завершалась нулевым байтом.

На месте последнего параметра функции задан набор флагов **DT_SINGLELINE | DT_CENTER | DT_VCENTER**, значения которых определяются в заголовочных файлах Windows. Флаги показывают, что текст будет выводиться в одну строку, по центру относительно горизонтали и вертикали внутри прямоугольной области, заданной четвертым параметром. Благодаря этому текст «**Сообщения обрабатываются!**» выводится в центре клиентской области.

Когда клиентская область становится недействительной (например, при изменении размеров окна), **WndProc** получает новое сообщение **WM_PAINT**. Обрабатывая его, программа вновь вызывает функцию **GetClientRect** и поэтому рисует текст опять в центре окна.

После окончания работы с графическими функциями в блоке обработки сообщения **WM_PAINT** необходимо освободить полученный контекст устройства с помощью функции **EndPaint**.

Добавим к сказанному: пара функций **BeginPaint** и **EndPaint** выполняют дополнительную задачу. Первая, **BeginPaint**, блокирует ресурс (в нашем случае окно) от одновременного обращения к нему других функций с целью изменения состояния ресурса. В нашем случае функции **GetClientRect** и **DrawText** будут обрабатывать наше окно монопольно. Вторая, **EndPaint**, освобождает ресурс (окно) для свободного доступа других компонентов приложения и операционной системы.

Следует отметить, что при выводе текста используются текущие установки для шрифта, цвета фона и цвета текста.

Контрольные вопросы

1. Почему приведенную простейшую программу мы назвали простейшей?
2. Чем принципиально отличается полноценная программа от простейшей программы?

5.4. Программирование графики

5.4.1. Основные средства

Графический интерфейс устройства (graphic device interface, GDI) — это составная часть Windows API, обеспечивающая графический вывод на устройства отображения информации, такие как дисплей или принтер. Windows-приложение не имеет непосредственного доступа к аппаратным устройствам. Вместо этого оно обращается к функциям GDI, а GDI транслирует эти обращения к программным драйверам физических устройств, обеспечивая аппаратную независимость приложений (рис. 5.5). Код библиотеки GDI находится в файле gdi32.dll, т. е. библиотека является динамически загружаемой. Драйверы стандартных устройст поставляются как часть Windows, а драйверы специализированных устройств создаются производителями оборудования.

5.4.2. Контекст устройства

Взаимодействие приложения с GDI осуществляется при непременном участии еще одного посредника — так называемого контекста устройства.

Контекст устройства (device context) — это внутренняя структура данных, которая определяет набор графических объектов и ассоциированных с ними атрибутов, а также графических режимов, влияющих на взаимодействие с устройствами.

В следующем списке приведены основные графические объекты:

— *перо* (pen) для рисования линий;

— *кисть* (brush) для заполнения фона или заливки фигур;

— *растровое изображение* (bitmap) для отображения рисунков в указанной области окна;

— *палитра* (palette) для определения набора доступных цветов;

— *шрифт* (font) для вывода текста;

— *регион* (region) для отсечения области вывода.

Если необходимо рисовать на устройстве графического вывода (экране дисплея или принтере), то сначала нужно получить *дескриптор контекста устройства*. Возвращая этот дескриптор после вызова соответствующих функций, Windows тем самым предоставляет разработчику право на использование данного устройства. После этого дескриптор контекста устройства передается как параметр в функции GDI, чтобы идентифицировать устройство, на котором должно выполняться рисование.

Контекст устройства содержит много атрибутов, определяющих поведение функций GDI. Благодаря этому списки параметров функций GDI содержат только самую необходимую информацию, например, начальные координаты или размеры графического объекта. Все остальное система извлекает из контекста устройства.

Когда в приложении предыдущего параграфа вызывалась функция **DrawText**, то в ее параметрах нужно было указать только дескриптор контекста устройства, адрес строки с выводимым текстом, его длину, положение и размеры прямоугольной области для размещения текста, а также способ позиционирования текста внутри этой области. Однако при этом не указывались шрифт, цвет текста, цвет фона, режим смешивания фона, режим рисования и другие атрибуты, потому что все они являются частью контекста устройства и имеют значения по умолчанию. И только если значения по умолчанию не устраивают разработчика, необходимо вызывать функции GDI, изменяющие значения соответствующих атрибутов.

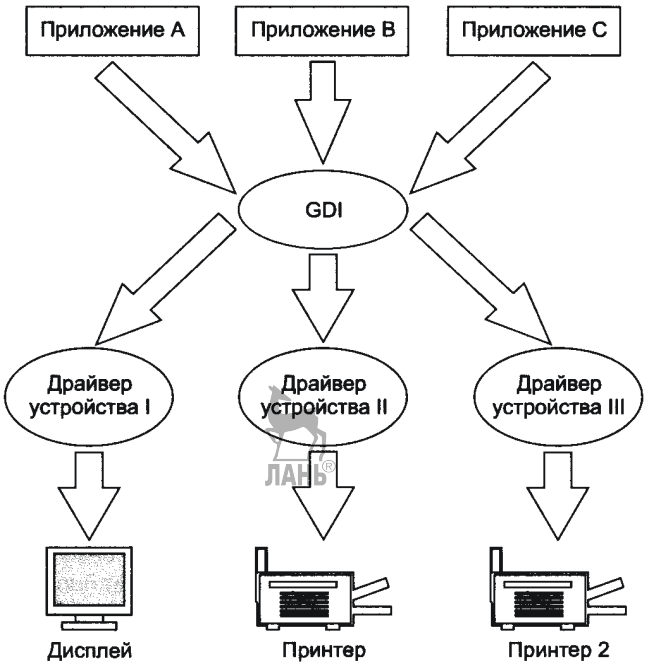


Рис. 5.5
Взаимодействие между приложениями, GDI и драйверами устройств

5.4.3. Типы контекстов устройства

Windows поддерживает следующие типы *контекстов устройства*:

— контекст дисплея;
— контекст принтера;
— контекст в памяти (совместимый контекст);
— метафайловый контекст;
— информационный контекст.

Конечно, чаще всего используется контекст дисплея. Первое знакомство с ним состоялось, когда рассматривалась обработка сообщения **WM_PAINT** в программе предыдущего параграфа. Сообщение **WM_PAINT** уведомляет программу, что часть или вся клиентская область окна *недействительна* и ее следует перерисовать. Понятие недействительного, или обновляемого, региона очень важно для взаимодействия Windows и приложения, поэтому, прежде чем продолжить более подробное рассмотрение контекста дисплея, мы остановимся кратко на следующей теме.

5.4.4. Регионы Windows

Для повышения эффективности работы Windows оперирует с несколькими типами регионов. Идея заключается в том, чтобы рисовать именно в той части окна, которая требует обновления, а не перерисовывать все окно. Такие регионы позволяют отсекать вывод той части графической информации, которая не может быть отображена в данный момент. Вообще полное изучение всей иерархии регионов и их взаимодействия является непростой задачей, требующей пространного изложения. В то же время, приведенного ниже упрощенного описания достаточно для понимания работы большинства функций Windows API.

Обновляемый регион (update region), или как его тоже иногда называют, *недействительный регион* (invalid region), — это часть окна, которая требует обновления после возникновения тех или иных событий.

Видимый регион (visible region) — та часть окна, которую в данный момент видит пользователь. Система изменяет видимый регион окна и в том случае, когда окно изменяет размеры, и в том случае, когда перемещение другого окна либо закрывает часть данного окна, либо открывает закрытую прежде часть.

Регион отсечения (clipping region) ограничивает область, внутри которой система разрешает отображение графической информации. Когда приложение получает контекст устройства при помощи функции **BeginPaint**, система устанавливает регион отсечения путем пересечения видимого региона и обновляемого региона. Приложение может ужесточить регион отсечения и ввести дополнительные ограничения при помощи вызова функций **SetWindowRgn**, **SelectClipPath** или **SelectClipRgn**.

Если при создании окна функцией **CreateWindow** был использован стиль **WS_CLIPCHILDREN** или **WS_CLIPSIBLINGS**, то это вносит дополнительные правила в определение видимого региона, исключая из него любое дочернее или любые «сестринские» окна. Благодаря этому рисование не затрагивает отображаемые области таких окон.

5.4.5. Получение контекста устройства

Контекст устройства создавать не нужно — об этом уже позаботилась операционная система. Нужно лишь получить его дескриптор. Windows предоставляет для этого два метода.

Первый метод получения дескриптора контекста устройства. Этот метод используется при обработке сообщения **WM_PAINT**. Контекст устройства получают вызовом функции **BeginPaint**, имеющей следующий прототип:

```
HDC BeginPaint (
        HWND hWnd,              // дескриптор окна
      LPPAINTSTRUCT lPaint  // указатель на структуру типа PAINTSTRUCT
        );.
```

В случае успешного завершения функция возвращает дескриптор контекста дисплея для клиентской области окна.

Кроме этого функция заполняет поля структуры **PAINTSTRUCT**, имеющей следующее определение:

```
typedef struct tagPAINTSTRUCT
{
   HDC hdc;        // контекст устройства
   BOOL fErase; // признак стирания фона клиентской области
   RECT rcPaint; // границы недействительного прямоугольника
   BOOL fRestore;
   BOOL fIncUpdate;
   BYTE rgbReserved [32];
} PAINTSTRUCT;
```

Последние три поля используются операционной системой.

Поле **rcPaint** типа **RECT** содержит координаты обновляемого прямоугольника в пикселах *относительно* левого верхнего угла клиентской области окна. Эти координаты либо определяются системой, либо задаются при вызове функции **InvalidateRect**. Регион отсечения в этом случае определяется посредством пересечения видимого региона и обновляемого прямоугольника.

Поле **fErase** определяет, будет ли Windows обновлять фон недействительного региона. Чаще всего **fErase** имеет значение **TRUE**, что означает стирание (обновление) фона. Когда обновляемый регион формируется вызовом функции **InvalidateRect** или **InvalidateRgn**, один из параметров этих функций разрешает или подавляет стирание фона.

Если задано стирание фона, то функция **BeginPaint** посылает оконной процедуре сообщение **WM_ERASEBKGND**. Приложение может обработать это сообщение, чтобы отобразить однородный или растровый фон. Однако обычно сообщение обрабатывается по умолчанию функцией **DefWindowProc**, которая обновляет фон с использованием кисти, определенной в поле **hbrBackground** класса окна.

Следует отметить, что функция **BeginPaint** сбрасывает обновляемый регион в **NULL**, или, другими словами, превращает недействительный регион в действительный. Это предотвращает повторную генерацию системой сообщения **WM_PAINT** до тех пор, пока обновляемый регион вновь не изменится.

До своего завершения функция **BeginPaint** посылает оконной процедуре еще одно сообщение — **WM_NCPAINT**. Оно заставляет приложение перерисовать так называемую неклиентскую область, которая представляет собой остальную часть окна — кроме клиентской области. Как правило, это сообщение обрабатывается также функцией **DefWindowProc**.

Типовой процесс обработки сообщения **WM_PAINT** выглядит следующим образом:

```
case WM_PAINT:
hDC=BeginPaint (hWnd, &ps);
     ...                  // использование функций GDI
EndPaint (hWnd, &ps);
return 0;.
```

После завершения операций рисования приложение должно вызвать функцию **EndPaint**, чтобы освободить контекст устройства.

Второй метод получения дескриптора контекста устройства. Иногда рисование должно происходить не в блоке обработки сообщения **WM_PAINT**, а при обработке других сообщений. Или, быть может, дескриптор контекста устройства нужен не для рисующих, а для иных функций (например, в качестве аргумента функции **GetTextMetrics**), вызываемых вне блока обработки сообщения **WM_PAINT**. В таких ситуациях контекст устройства получают либо вызовом функции **GetDC**:

```
hdc=GetDC (hWnd);
```

либо вызовом функции **GetWindowDC**:

```
hdc=GetWindowDC (hWnd);
```

Первая функция возвращает дескриптор контекста дисплея для клиентской области окна **hWnd**, а вторая — для всего окна. Если в качестве аргумента hWnd передать значение **NULL**, то обе функции вернут дескриптор устройства для всего экрана.

По окончании работы с функциями GDI необходимо освободить контекст устройства с помощью функции **ReleaseDC**, например:

```
hDC=GetDC (hWnd);
     ...      // использование функций GDI
ReleaseDC (hWnd, hDC);
```

Заметим, что, в отличие от контекста устройства, полученного при помощи вызова **BeginPaint**, контекст устройства, возвращаемый функцией **GetDC**, работает с регионом отсечения, который равен всей клиентской области. Это значит, что рисование можно производить в любом месте клиентской области, а не только в недействительном прямоугольнике, если он вообще определен. В отличие от функции **BeginPaint**, функция **GetDC** не делает действительными какие-либо недействительные зоны.

Обычно вторым методом получения дескриптора контекста дисплея пользуются программы текстовых редакторов, обрабатывая сообщения от клавиатуры, а также программы рисования, обрабатывая сообщения от мыши. Это

позволяет обновлять клиентскую область непосредственно в ответ на ввод информации с клавиатуры или от мыши и при этом не объявлять часть окна недействительной для генерации сообщения **WM_PAINT**. Тем не менее подобные программы должны уметь обновлять окно в любой момент, когда они получают сообщение **WM_PAINT**.

5.4.6. Использование сообщения WM_PAINT

Обычно приложение рисует что-либо в окне, реагируя на сообщение **WM_PAINT**. Система посылает это сообщение окну во всех случаях, требующих перерисовки клиентской области окна. Например, типичными причинами генерации этого сообщения могут быть следующие события:

— изменились размеры или местоположение окна;

— клиентская область была частично или полностью закрыта другим окном или выпадающим меню, а теперь закрывающий объект исчез;

— приложение вызвало одну из функций работы с полосами прокрутки.

Кроме того, приложение может само инициировать посылку сообщения **WM_PAINT** посредством вызова одной из функций, **InvalidateRect**, **InvalidateRgn** или **UpdateWindow**.

Функция **UpdateWindow** посылает сообщение **WM_PAINT** непосредственно в оконную процедуру, минуя очередь приложения.

Работа с сообщениями **WM_PAINT** требует от разработчика понимания общей парадигмы отображения графики на экране, которая принята в системах семейства Windows. Желательно структурировать программу таким образом, чтобы информация, необходимая для рисования в клиентской области, готовилась там, где это удобно с точки зрения реализуемого алгоритма. Но само рисование должно выполняться только тогда, когда появляется сообщение **WM_PAINT**. Впрочем, иногда решаемая задача диктует и другие подходы к рисованию, например, оно может выполняться при обработке сообщений от мыши или от таймера.

5.4.7. Использование функций InvalidateRect и InvalidateRgn

Если в каком-то месте алгоритма необходимо сгенерировать сообщение **WM_PAINT** для перерисовки некоторой части окна, то приложение может воспользоваться функциями **InvalidateRect** и **InvalidateRgn**.

Функция **InvalidateRect** имеет следующий прототип:

```
BOOL InvalidateRect (HWND hWnd, CONST RECT* lpRect, BOOL bErase);
```

Параметры этой функции перечислены в следующем списке:

— **hWnd** — дескриптор окна, у которого изменился обновляемый регион. Если этот параметр равен **NULL**, то система обновляет и перерисовывает все окна приложения, а также посылает сообщения **WM_ERASEBKGND** и **WM_NCPAINT** оконной процедуре до возврата из функции;

— **lpRect** — указатель на структуру типа **RECT**, содержащую клиентские координаты прямоугольника, который добавляется к обновляемому региону. Если этот параметр имеет значение **NULL**, то к обновляемому региону добавляется вся клиентская область;

— **bErase** — флаг, определяющий, будет ли стираться фон обновляемого региона. Если этот параметр равен **TRUE**, то фон стирается, когда вызывается функция **BeginPaint**. Если указано значение **FALSE**, то фон остается без изменения.

Последний параметр очень важен. Нужно быть очень внимательным, определяя его значение. Ни в коем случае нельзя передавать значение **TRUE**, если можно обойтись значением **FALSE**, так как это может оказаться причиной очень неприятного мерцания фона окна. Если же стирание фона все-таки необходимо, следует постараться минимизировать обновляемый регион. Иными словами, если можно ограничить стираемую область некоторым прямоугольником **rcUpdate**, то в качестве второго параметра следует передавать ссылку на этот прямоугольник, а не значение **NULL**.

Функция **InvalidateRgn** имеет следующий прототип:

BOOL InvalidateRgn (HWND hWnd, HRGN hRgn, BOOL bErase);

В этой функции все параметры, кроме второго, имеют тот же смысл, что и в предыдущей функции. Параметр **hRgn** содержит дескриптор региона, добавляемого к обновляемому региону. Этот дескриптор можно получить при помощи функций типа **CreateRectRgn**, **CreatePolygonRgn** и других функций.

Таким образом, и функция **InvalidateRect**, и функция **InvalidateRgn** изменяют обновляемый регион так, что в любом случае он уже не является пустым. Если же обновляемый регион пуст, то система отправляет окну **hWnd** сообщение **WM_PAINT**.

5.4.8. Системы координат и преобразования

Система координат для окна базируется на координатной системе дисплея. Основной единицей измерения служит пиксел. Точки на экране задаются парой координат (x, y). При этом x-координаты возрастают слева направо, а y-координаты — сверху вниз. Расположение начала координат зависит от режима отображения.

Позиция окна на экране задается в приложениях в так называемых *экранных координатах* (screen coordinates) либо в *клиентских координатах* (client coordinates), для которых началом координат является левый верхний угол экрана. Часто эта позиция описывается структурой типа **RECT**, содержащей экранные координаты левого верхнего угла и правого нижнего угла окна.

Позиция точки в окне задается либо в *оконных координатах* (window coordinates), либо в *клиентских координатах* (client coordinates) (рис. 5.6). Выбор системы координат зависит от используемых функций Windows, поэтому следует обращать внимание на спецификацию системы координат в справочных материалах. Оконные и клиентские координаты гарантируют надлежащее позиционирование точки в окне независимо от положения окна на экране.

Приложения используют координатные системы и геометрические преобразования для масштабирования, вращения, перемещения, сдвига и зеркального отражения графических объектов. Каждая из используемых систем явля-

ется некоторой разновидностью декартовой прямоугольной системы координат. По практическим соображениям в Windows реализована поддержка четырех координатных систем, функционирующих подобно некоторому конвейеру, по которому движется геометрическая модель, описывающая размеры и местонахождение объектов:

— *мировая система координат* (world space) может использоваться как начальная система координат, обеспечивающая любое преобразование;

— *страничная система координат* (page space) используется или как следующая система координат после мировой системы координат, или как стартовая система, которая поддерживает ограниченные преобразования. В этой системе могут устанавливаться режимы отображения;

— *система координат устройства* (device space) используется после страничной системы. В ней осуществляется только перемещение начала координат, чтобы обеспечить надлежащее положение изображения в физической системе координат;

— *физическая система координат* (physical device space) — последняя система в конвейере геометрических преобразований. Она используется драйвером графического устройства.

Физическая система координат представляет собой матрицу пикселов фиксированной ширины и высоты. Начало координат находится в левом верхнем углу. Ось *X* направлена слева направо, а ось *Y* — сверху вниз. Максимальный размер физического устройства $2^{27}\times2^{27}$ пикселов.

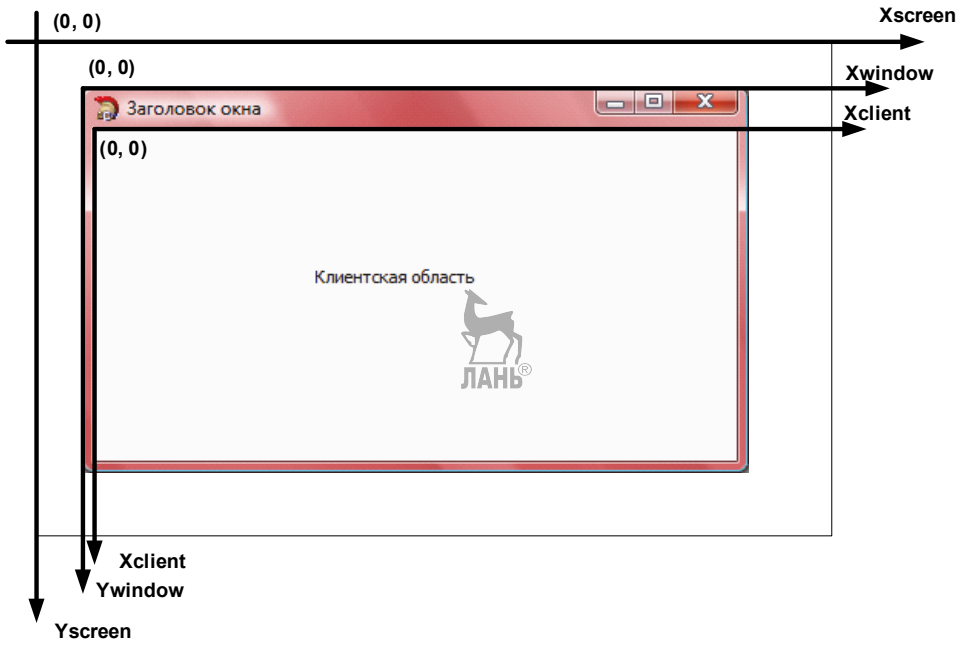


Рис. 5.6
Системы координат Windows

Для дисплея физические координаты называют также *экранными координатами*. Экранные координаты обычно используются в операциях управления окнами. Например, функция **GetWindowRect** возвращает ограничивающий прямоугольник окна в *экранных координатах*. Также *экранные координаты* содержатся в параметрах таких сообщений, как **WM_NCMOUSEMOVE**. Это сообщение генерируется при перемещении курсора мыши в пределах неклиентской части окна.

Следует учитывать, что бывают случаи, когда не все пикселы физической поверхности могут отображаться устройством. Например, для принтеров существуют механические ограничения, не позволяющие печатать у краев страниц. В таких ситуациях приложение должно получить информацию об отображаемой части поверхности при помощи функции **GetDeviceCap**.

Система координат устройства. *Координаты устройства* (device coordinates) используются при работе *контекстами устройств*. В общем случае система координат устройства является подмножеством соответствующей физической системы координат.

Для контекстов устройств, созданных функциями **CreateDC**, **CreateIC** и **CreateCompatibleDC**, система координат устройства идентична физической системе координат.

Для *контекстов устройств*, связанных с конкретными окнами (т. е. возвращаемых функциями **GetDC**, **GetWindowDC** и **BeginPaint**), система координат устройства определяется прямоугольником окна или его клиентской области.

Как и в физической системе координат, левый верхний угол системы координат устройства имеет координаты (0, 0), ось *X* направлена слева направо, а ось *Y* — сверху вниз. Зная дескриптор контекста устройства, можно узнать относительную позицию системы координат устройства в его физических координатах при помощи функции **GetDCOrgEx**.

В интерактивных графических приложениях, использующих сообщения мыши для работы с графическими объектами, возникает необходимость преобразования между физическими экранными координатами и координатами устройств (клиентскими координатами). Windows предлагает соответствующие функции для решения этой задачи:

```
BOOL ClientToScreen(HWND hWnd, LPOINT lpPoint);
BOOL ScreenToClient(HWND hWnd, LPOINT lpPoint);
```

В области управления окнами координаты устройства чаще всего интерпретируются как *клиентские координаты*. Клиентские координаты содержатся также в сообщениях, генерируемых мышью, таких как **WM_MOUSEMOVE** и **WM_MOUSEDOWN**.

Страничная система координат и режимы отображения. Обе рассмотренные системы координат ограничиваются представлением в виде аппаратнозависимого массива пикселов. Размер окна на экране с высоким разрешением отличается от размера окна при низком разрешении, а толщина напечатанной линии из трех пикселов будет зависеть от разрешения принтера. Чтобы графи-

ческое программирование в меньшей степени зависело от устройства, Windows позволяет приложениям создавать собственные логические системы координат, приближенные к их геометрическим моделям. В таких системах координат удобнее работать, и они гораздо меньше зависят от оборудования.

Одной из двух логических систем координат в Windows является *страничная система координат*. Мировые координаты являются второй логической системой координат. Однако под логической системой координат чаще всего подразумевается страничная система координат.

Контрольные вопросы

1. Что считается графическим интерфейсом устройства?
2. Что представляет собой контекст устройства?
3. Что считается дескриптором контекста устройства?
4. Перечислите основные графические объекты.
5. Какие бывают регионы Windows?
6. Какие имеются разновидности работы с контекстом устройства?
7. В каких случаях приложению отправляется сообщение **WM_PAINT**?
8. Какая система координат используется для работы с контекстом устройства?

5.5. Рисование линий, фигур, текста

GDI содержит набор графических объектов, обеспечивающих выполнение графических операций. К таким объектам относятся перья, кисти, растровые изображения, палитры, шрифты. Использование любого графического объекта предполагает выполнение следующей последовательности операций.

1. Создать графический объект.
2. Выбрать созданный объект в контекст устройства.
3. Вызвать графическую функцию, работающую с объектом.
4. Удалить объект из контекста устройства, вернув предшествующий объект.
5. Уничтожить графический объект.

Для создания GDI-объектов предназначены соответствующие функции **Create …**, которые в случае успешного завершения возвращают дескриптор объекта.

Выбор объекта в контекст устройства осуществляется при помощи функции **SelectObject**, которая имеет следующий прототип:

```
HGDIOBJ SelectObject
(
    HDC hdc,                // дескриптор контекста устройства
    HGDIOBJ hgdiobj         // дескриптор GDI-объекта
);
```

В результате ее выполнения новый объект **hgdiobj** заменяет предшествующий объект того же типа в контексте устройства **hdc**.

Функция возвращает дескриптор предшествующего объекта. Для корректной работы приложение должно запомнить этот дескриптор и после окончания рисования с новым объектом (шаг 3) вернуть в контекст устройства предшествующий объект (шаг 4).

Для уничтожения объекта, ставшего ненужным, вызывается функция **DeleteObject**.

Рисование любой линии можно было бы свести к многократному вызову **SetPixel** с соответствующим изменением координат x и y. Однако, во-первых, это не очень удобно для программиста, а во-вторых, подобный способ рисования был бы очень медленным. Значительно более эффективной в графических системах является реализация функций рисования отрезков и других сложных графических операций на уровне драйвера устройства, который содержит код, оптимизированный для этих операций. Поэтому Windows предоставляет соответствующий набор графических функций.

5.5.1. Позиция пера

Любая линия рисуется в Windows с использованием графического объекта, называемого *пером*. Контекст устройства содержит перо по умолчанию — сплошное перо черного цвета толщиной 1 пиксел. Многие графические функции начинают рисование с так называемой текущей позиции пера.

Текущая позиция пера является одним из атрибутов контекста устройства. Она определяется значением типа **POINT**. По умолчанию текущая позиция пера установлена в точку (0, 0). Если нужно изменить текущую позицию, вызывается функция **MoveToEx**:

```
BOOL MoveToEx
(
  HDC hdc,          //  дескриптор контекста устройства
  int X,
  int Y,            // X, Y — точка перехода
  LPPOINT lpPoint       //  предыдущая позиция
);
```

Если предыдущая позиция пера нас не интересует, то нужно присвоить последнему параметру значение **NULL**. В случае успешного завершения функция возвращает ненулевое значение.

В любой момент можно получить значение текущей позиции пера, вызвав функцию

```
GetCurrentPositionEx (hdc, &pt);
```

Результат выполнения функции помещается в переменную **pt** типа **POINT**.

5.5.2. Рисование прямой линии

Для создания прямой линии используется функция **LineTo**:

```
BOOL LineTo
(
```

```
    HDC hdc,      // дескриптор контекста устройства
    int xEnd,     //  x — координата конечной точки
    int yEnd      //  y — координата конечной точки
);
```

Эта функция рисует отрезок, начиная с точки, в которой находится текущая позиция пера, до точки (**xEnd**, **yEnd**), не включая последнюю точку в отрезок. Если функция завершается успешно, то она возвращает ненулевое значение, а текущая позиция пера устанавливается в точку (**xEnd**, **yEnd**).

5.5.3. Рисование ломаной линии

Последовательность связанных отрезков можно нарисовать с помощью функции **Polyline**:

BOOL Polyline (HDC hdc, CONST POINT* lppt, int cPoints);

Второй параметр здесь — это адрес массива точек, а третий — количество точек. Например:

```
POINT pt[5]={{100, 100}, {200, 100}, {200, 200}, {100, 200}, {100, 100}};
Polyline(hDC, pt, 5);
```

Функция **Polyline** не использует текущую позицию пера и не изменяет ее.

Функция **PolylineTo** предназначена для рисования последовательности связанных отрезков:

BOOL PolylineTo (HDC hdc, CONST POINT* lppt, DWORD cPoints);

Она использует текущую позицию пера для начальной точки и после каждого своего выполнения устанавливает текущую позицию в конец нарисованного отрезка. Если мы захотим применить ее в рассмотренном выше примере, то вызов функции **Polyline** надо заменить двумя вызовами:

```
MoveToEx (hDC, pt[0].x, pt[0].y, NULL);
PolylineTo(hDC, pt+1, 4);
```

5.5.4. Рисование дуги

Дуги в Windows рисуются как часть эллипса. Размеры и расположение эллипса определяются ограничивающим прямоугольником. Ограничивающий прямоугольник задается координатами левой верхней и правой нижней вершин. Если обозначить эти координаты как (**xLeft**, **yTop**) и (**xRight**, **yBottom**), тогда центром эллипса будет точка (**x0**, **y0**), где

x0=xLeft+(xRight – xLeft)/2, a y0=yTop+(yBottom – yTop)/2.

Для рисования дуг предназначены функции **Arc**, **ArcTo** и **AngleArc**. Первые две функции имеют одинаковый набор параметров, поэтому рассмотрим прототип функции:

BOOL Arc (HDC hdc, int xLeft, int yTop, int xRight, int yBottom,
int xStart, int yStart, int xEnd, int yEnd);

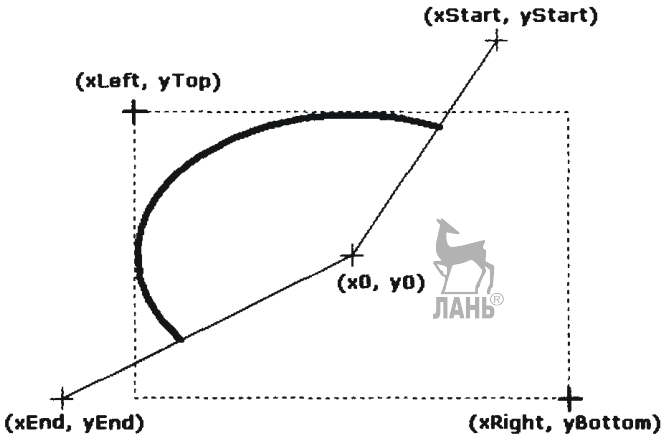


Рис. 5.7
Дуга, нарисованная с помощью функции **Arc** против часовой стрелки

Параметры со второго по пятый задают вершины ограничивающего прямоугольника. Начало и конец дуги определяются начальным и конечным углами, которые задаются косвенно через две дополнительные точки с координатами (**xStart, yStart**) и (**xEnd, yEnd**). Начало дуги — это пересечение эллипса с лучом, который начинается в центре эллипса и проходит через точку (**xStart, yStart**). Конец дуги — это пересечение эллипса с лучом, который начинается в центре эллипса и проходит через точку (**xEnd, yEnd**).

Направление дуги определяется соответствующим атрибутом в контексте устройства, значение которого можно получить вызовом функции **GetArcDirection** или установить вызовом функции **SetArcDirection**. При использовании функции **SetArcDirection** в качестве второго параметра нужно передать одно из значений: **AD_COUNTERCLOCKWISE**, устанавливающее режим рисования против часовой стрелки, либо **AD_CLOCKWISE**, устанавливающее режим рисования по часовой стрелке. По умолчанию в контексте устройства используется значение **AD_COUNTERCLOCKWISE**.

На рисунке 5.7 показана дуга, нарисованная при помощи вызова функции

Arc (hdc, 100, 100, 400, 300, 350, 50, 50, 300);

направление по умолчанию — против часовой стрелки.

Если перед указанным обращением к функции **Arc** вызвать функцию

SetArcDirection (hdc, AD_CLOCKWISE);

то функция **Arc** нарисует дугу, показанную на рисунке 5.8.

Функция **Arc** не использует текущую позицию пера и не обновляет ее.

Функция **ArcTo** отличается от функции **Arc** тем, что она проводит линию из текущей позиции пера в заданную начальную точку дуги, и только после этого рисует дугу. После завершения рисования функция **ArcTo** перемещает текущую позицию пера в конечную точку дуги.

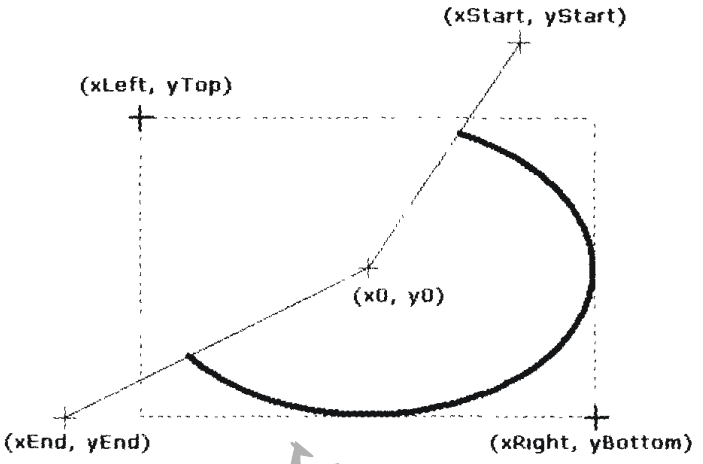


Рис. 5.8
Дуга, нарисованная с помощью функции **Arc** по часовой стрелке

5.5.5. Рисование прямоугольника

Для рисования прямоугольников предусмотрены функции **Rectangle**, **FillRect**, **FrameRect**, **InvertRect** и **DrawFocusRect**. Мы рассмотрим только первую.

Функция **Rectangle** имеет следующий прототип:

BOOL Rectangle (HDC hdc, int xLeft, int yTop, int xRight int yBottom);.

Она рисует прямоугольник, определяемый четырьмя координатами. Параметры **xLeft** и **yTop** задают положение левого верхнего угла, а **xRight** и **yBottom** — положение правого нижнего угла. Стороны прямоугольника всегда параллельны горизонтальной и вертикальной сторонам экрана.

5.5.6. Рисование эллипса

Для рисования эллипсов предназначена функция **Ellipse**, прототип которой приведен ниже:

BOOL Ellipse (HDC hdc, int xLeft, int yTop, int xRight, int yBottom);

Функция рисует эллипс в ограничивающем прямоугольнике, заданном координатами **xLeft, yTop, xRight, yBottom**. Ранее уже рассматривалось рисование дуги эллипса с помощью функции **Arc**.

5.5.7. Рисование многоугольника

Для рисования произвольного многоугольника предназначена функция **Polygon**:

BOOL Polygon (HDC hdc, CONST POINT * lpPoints, int nCount);

Работа этой функции напоминает рисование ломаных линий функцией **Polyline**. Второй параметр функции принимает адрес массива точек, а третий — количество точек. В отличие от **Polyline** функция **Polygon** автоматически замыкает фигуру.

Контрольные вопросы

1. Какую последовательность действий нужно выполнить при рисовании графического объекта?
2. Что называется позицией пера?
3. Какие средства имеются для рисования прямой линии?
4. Как нарисовать ломаную линию?
5. Какова методика рисования дуги?
6. Как рисуется прямоугольник?
7. Какая функция используется для рисования эллипса?
8. Как нарисовать многоугольник?

5.6. Средства ввода

Приложение, работающее под управлением системы Windows, получает данные от пользователя через устройства ввода, главными из которых являются клавиатура и мышь. Когда сообщения от клавиатуры или мыши воспринимаются стандартными элементами управления, такими как кнопки, поля редактирования меню, то обработка этих событий реализована в соответствующих оконных классах. Однако иногда необходимо непосредственно обрабатывать сообщения от клавиатуры или мыши. Например, это нужно в приложениях, воспринимающих пользовательский ввод или в главном окне, или в одном из дочерних окон. В Windows предусмотрен широкий набор функций для управления клавиатурой и мышью.

5.6.1. Клавиатура

Windows обеспечивает поддержку клавиатуры, независимо от аппаратуры и от национального языка, используемого в системе. Независимость от аппаратуры достигается благодаря использованию программного драйвера клавиатуры. Независимость от национального языка реализуется выбором нужной кодовой страницы.

На рисунке 5.9 приведена схема обработки сообщения от клавиатуры, используемая в операционной системе.

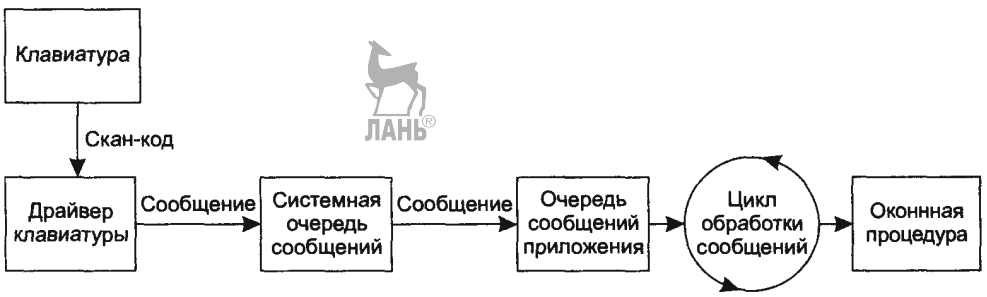


Рис. 5.9
Модель клавиатурного ввода

Каждой клавише на клавиатуре соответствует два уникальных целых числа, называемых скан-кодом нажатия и скан-кодом отпускания клавиши. *Скан-код* (scan code) является аппаратно-зависимым, т. е. на клавиатурах разных производителей одна и та же клавиша может иметь разные скан-коды. Когда пользователь нажимает или отпускает клавишу, драйвер клавиатуры получает соответствующие скан-коды. Полученный скан-код преобразуется драйвером в *виртуальный код клавиши* (virtual key code). Этот код является уже аппаратно-независимым, поэтому в системе Windows он однозначно идентифицирует нажатую или отпущенную клавишу. Завершающим действием драйвера является создание сообщения, содержащего скан-код, виртуальный код клавиши и другую информацию о нажатой или отпущенной клавише. Это сообщение помещается в системную очередь сообщений.

Windows извлекает сообщение из системной очереди сообщений и передает его в очередь сообщений того приложения, которое имеет *активное окно*. Далее сообщение извлекается в *цикле обработки сообщений*, поступая в конечном счете на обработку в соответствующую *оконную процедуру*.

5.6.2. Фокус ввода

Все приложения, работающие одновременно под управлением Windows, должны иметь возможность получать данные от клавиатуры. Если какое-то приложение имеет более одного окна, то клавиатура должна разделяться между окнами в рамках одного приложения. Это разделение Windows реализует, используя понятие фокуса ввода. *Фокус ввода* (keyboard focus) — это временное свойство окна, которое означает, что все сообщения от клавиатуры направляются именно этому окну. В каждый момент времени из всех окон, имеющихся на экране, только одно окно может иметь фокус ввода.

Понятие фокуса ввода тесно связано с понятием активного окна. *Активное окно* — это окно верхнего уровня, с которым в данный момент работает пользователь. Система выделяет текст заголовка активного окна, поэтому отыскать его на экране довольно просто. Если активное окно минимизировано, то Windows выделяет текст заголовка на панели задач.

Фокусом ввода всегда владеет либо активное окно, либо одно из его дочерних окон. Часто дочерними окнами являются элементы управления — кнопки, переключатели, флажки, текстовые поля и списки, которые обычно размещаются в окне диалога. Элементы управления по-разному показывают, что они находятся в фокусе. Так, вокруг текста кнопки выводится точечная линия, а текстовое поле показывает, что оно получило фокус ввода, при помощи мигающего курсора.

Если активное окно минимизировано, то окна с фокусом ввода нет. Windows продолжает посылать программе сообщения клавиатуры, но эти сообщения выглядят иначе, чем сообщения, направляемые активным неминимизированным окнам.

Обрабатывая сообщения **WM_SETFOCUS** и **WM_KILLFOCUS**, оконная процедура может определить текущий статус связанного с ней окна. Первое сообщение показывает, что окно получило фокус ввода, второе — что окно потеряло фокус ввода.

Для работы с фокусом ввода предусмотрены следующие функции:

— функция **HWND SetFocus *(*HWND hWnd*)*** устанавливает фокус ввода на окно **hWnd**;

— функция **HWND GetFocus()** возвращает дескриптор окна, имеющего фокус ввода в текущий момент.

5.6.3. Клавиши и символы

Клавиатура всегда генерирует некоторые числовые коды, но ее можно рассматривать либо как совокупность отдельных физических клавиш, либо как средство генерации кодов символов.

ЛАНЬ®

В первом случае любой генерируемый код должен идентифицировать клавишу и показывать, нажата она или отпущена. Во втором варианте код, вырабатываемый при нажатии клавиши, идентифицирует уникальный символ из набора символов. По умолчанию используется набор символов ANSI.

У многих клавиш современного компьютера нет кодов символов. Ни функциональные клавиши, ни клавиши управления курсором их не генерируют. Поэтому в программах, использующих ввод с клавиатуры нетривиальным способом, обычно приходится иметь дело с клавиатурой и как с совокупностью клавиш, и как с генератором символов одновременно.

Клавиши можно разделить на следующие четыре группы:

— клавиши-переключатели — **Caps Lock**, **Num Lock**, **Scroll Lock** и в некоторых случаях клавиша **Insert**. При нажатии такой клавиши включается ее состояние, а при повторном нажатии — выключается. Часто на клавиатуре имеются световые индикаторы состояния клавиш-переключателей;

— клавиши управления регистром — **Shift**, **Ctrl** и **Alt**. В нажатом состоянии такая клавиша меняет интерпретацию других клавиш. Иногда такие клавиши называют клавишами-модификаторами;

— клавиши, не генерирующие символов, функциональные клавиши, клавиши управления курсором, **Pause**, **Delete**.

— символьные клавиши — буквы, цифры и другие символы, пробел, **Tab**, **Backspace**, **Esc** и **Enter**. Клавиши **Tab**, **Backspace**, **Esc** и **Enter** также можно рассматривать как клавиши без символов.

Сообщения клавиатуры, которые приложение получает от Windows, можно разделить на «аппаратные» и «символьные». Это разделение соответствует указанной выше двойственной интерпретации клавиатуры.

5.6.4. Аппаратные сообщения

Аппаратные сообщения от клавиатуры приведены в таблице 5.10.

Обычно различают системные и несистемные сообщения. Для системных сообщений используется префикс SYS в составе идентификатора кода сообщения.

Таблица 5.10

Аппаратные сообщения от клавиатуры

Категория	Клавиша нажата	Клавиша отпущена
Несистемные сообщения	**WM_KEYDOWN**	**WM_KEYUP**
Системные сообщения	**WM_SYSKEYDOWN**	**WM_SYSKEYUP**

Системные сообщения **WM_SYSKEYDOWN** и **WM_SYSKEYUP**, вырабатываемые обычно при нажатии клавиш в сочетании с клавишей **Alt**, более важны для Windows, чем для приложения. Эти сообщения вызывают команды программного или системного меню. Они также используются для вызова системных функций, таких как смена активного окна (**Alt**+**Tab**) или как быстрые клавиши системного меню (**Alt** в сочетании с функциональной клавишей). Поскольку система Windows самостоятельно отрабатывает всю логику **Alt**-клавиш, то программистам фактически не нужно обрабатывать эти сообщения. Оконная процедура приложения, в конце концов, получит другие сообщения, являющиеся следствием этих аппаратных сообщений клавиатуры (например, выбор команды меню).

Обычно сообщения о нажатии и отпускании клавиш появляются парами. Однако если пользователь оставит клавишу нажатой так долго, что включится автоповтор, то Windows пошлет оконной процедуре серию сообщений **WM_KEYDOWN** и одно сообщение **WM_KEYUP**, когда, в конце концов, клавиша будет отпущена.

Для всех аппаратных сообщений клавиатуры 32-разрядная переменная **lParam**, передаваемая в оконную процедуру, содержит 6 полей (табл. 5.11).

Хотя некоторая информация в **lParam** при обработке аппаратных сообщений может оказаться полезной, гораздо более важен параметр **wParam**, содержащий виртуальный код клавиши.

Таблица 5.11

Интерпретация полей параметра lParam

Разряды	Назначение	Описание
15 … 0	Счетчик повторений	Число повторений данного сообщения. Равно либо единице, либо числу нажатий в режиме автоповтора, когда пользователь удерживает клавишу нажатой. Для сообщения **WM_KEYUP** всегда равен единице
23 … 16	Скан-код	Восьмибитный скан-код OEM[23]
24	Флаг расширенной клавиатуры	Устанавливается в единицу для дополнительных клавиш расширенной клавиатуры IBM (функциональные клавиши в верхней части клавиатуры, клавиши **Alt** и **Ctrl** на правой стороне клавиатуры и некоторые другие клавиши)
29	Код контекста	Устанавливается в единицу, если нажата клавиша **Alt** (для сообщений **WM_KEYDOWN** и **WM_KEYUP** всегда равен нулю)
30	Флаг предыдущего состояния клавиши	Единица, если клавиша была нажата, и нуль, если клавиша была отпущена
31	Флаг нового состояния клавиши	Нуль, если клавиша нажата, и единица, если клавиша отпущена

Виртуальный код клавиши (virtual key code) идентифицирует нажатую или отпущенную клавишу и является аппаратно-независимым. Идентификаторы виртуальных клавиш определены в файле **winuser.h**. В таблице 5.12 приведены наиболее часто используемые коды.

[23] Original Equipment Manufacturer — производитель оригинального оборудования.

Таблица 5.12

Виртуальные коды клавиш

Десятичное значение	Шестнадцатеричное значение	Идентификатор	Клавиатура IBM
3	03	**VK_CANCEL**	**Ctrl+Break**
8	08	**VK_BACK**	**Backspace**
9	09	**VK_TAB**	**Tab**
13	0D	**VK_RETURN**	**Enter**
16	10	**VK_SHIFT**	**Shift**
17	11	**VK_CONTROL**	**Ctrl**
18	12	**VK_MENU**	**Alt**
19	13	**VK_PAUSE**	**Pause**
20	14	**VK_CAPITAL**	**Caps Lock**
27	1B	**VK_ESCAPE**	**Esc**
32	20	**VK_SPACE**	Пробел
33	21	**VK_PRIOR**	**Page Up**
34	22	**VK_NEXT**	**Page Down**
35	23	**VK_END**	**End**
36	24	**VK_HOME**	**Home**
37	25	**VK_LEFT**	Стредка влево
38	26	**VK_UP**	Стрелка вверх
39	27	**VK_RIGHT**	Стрелка вправо
40	28	**VK_DOWN**	Стрелка вниз
44	2C	**VK_SNAPSHOT**	**Print Screen**
45	2D	**VK_INSERT**	**Insert**
46	2E	**VK_DELETE**	**Delete**
48 … 57	30 … 39		**0 … 9** (на основной клавиатуре)
65 … 90	41 … 5A		**A … Z**
112 … 123	70 … 7B	**VK_F1 … VK_F12**	**F1 … F12**
144	90	**VK_NUMLOCK**	**Num Lock**
145	91	**VK_SCROLL**	**Scroll Lock**

Обычно оконная процедура обрабатывает только небольшую часть из всех аппаратных сообщений клавиатуры, игнорируя остальные. Например, она может обрабатывать только те сообщения **WM_KEYDOWN**, которые содержат виртуальные коды для клавиш управления курсором, клавиши **Shift** и функциональных клавиш. В то же время аппаратные сообщения для символьных клавиш удобнее обрабатывать, используя символьное сообщение **WM_CHAR**.

5.6.5. Символьные сообщения

Когда нажимается символьная клавиша, генерируемый код зависит от многих факторов. На его формирование влияет положение клавиш-переключателей и клавиш управления регистром, а также текущая кодовая страница, установленная в системе. Все эти факторы учитывает функция **TranslateMessage**, вызываемая в цикле обработки сообщений, который находится в теле функции **WinMain** и имеет вид:

```
while (GetMessage (&msg, NULL, 0, 0))
{
```

```
    TranslateMessage (&msg);
    DispatchMessage(&msg);
}
```

Напомним, что функция **GetMessage** извлекает очередное сообщение из очереди и помещает его в структурную переменную **msg**, а функция **DispatchMessage** вызывает соответствующую оконную процедуру.

Между этими двумя функциями располагается вызов функции **TranslateMessage**, которая преобразует аппаратные сообщения клавиатуры в символьные сообщения. Если аппаратным сообщением является **WM_KEYDOWN**, а нажатая клавиша в сочетании с положением клавиш-модификаторов соответствует некоторому символу, то **TranslateMessage** помещает символьное сообщение в очередь сообщений.

В результате системной обработки сообщения **WM_KEYDOWN** может быть инициировано символьное сообщение **WM_CHAR** или **WM_DEADCHAR**.

Рассмотрим кратко назначение сообщения **WM_DEADCHAR**. На некоторых неамериканских клавиатурах имеются так называемые *немые клавиши* (dead keys), предназначенные для ввода диакритических знаков, таких как, например, «умляут». Диакритические знаки вводятся в паре с обычными символами, в результате чего формируется составной символ. Например, для ввода на немецкой клавиатуре символа «**Ü**» (**U** — умляут) пользователь сначала нажимает немую клавишу «умляут», а затем — клавишу **U**. В нашей программе не нужно обрабатывать сообщение **WM_DEADCHAR**, поскольку Windows берет это на себя, вырабатывая в результате сообщение **WM_CHAR**, которое содержит ANSI-код буквы с диакритическим знаком.

Таким образом, обработка символьных сообщений обычно сводится к обработке сообщения **WM_CHAR**. Параметр **lParam** сообщения **WM_CHAR**, передаваемый в оконную процедуру, имеет то же значение, что и параметр **lParam** породившего его аппаратного сообщения. Параметр **wParam** содержит код символа ANSI.

Символьные сообщения передаются в оконную процедуру в промежутке между аппаратными сообщениями клавиатуры. Например, если переключатель **Caps Lock** выключен, а пользователь нажимает и отпускает клавишу **A**, то оконная процедура получит следующие три сообщения:

```
WM_KEYDOWN,// Виртуальная клавиша A
WM_CHAR,              // ANSI-код символа «a»
WM_KEYUP,     // Виртуальная клавиша A.
```

Если пользователь, удерживая клавишу **Shift**, нажимает клавишу *A*, отпускает клавишу *A* и затем отпускает клавишу **Shift**, то оконная процедура получит пять сообщений:

```
WM_KEYDOWN,// Виртуальная клавиша VK_SHIFT
WM_KEYDOWN,// Виртуальная клавиша A
WM_CHAR,              // ANSI-код символа «a»
WM_KEYUP,     // Виртуальная клавиша A
WM_KEYUP,     // Виртуальная клавиша VK_SHIFT.
```

5.6.6. Мышь

Мышь — это устройство ввода информации с одной или более кнопками. Windows поддерживает однокнопочную, двухкнопочную или трехкнопочную мышь, а также мышь с колесиком. Наличие мыши можно определить при помощи функции **GetSystemMetrics**:

```
fMouse=GetSystemMetrics (SM_MOUSEPRESENT);
```

Если мышь установлена, то значение **fMouse** будет равно **TRUE**. Количество кнопок у мыши определяется следующим вызовом:

```
cButtons=GetSystemMetrics (SM_CMOUSEBUTTONS);
```

Если мышь не установлена, то функция вернет нулевое значение.

Для двухкнопочной мыши сложилась некоторая традиция интерпретации ее кнопок. Левая кнопка считается главной и выполняет самые распространенные действия, к которым относятся выбор предметов, перетаскивание значков и инициирование действий. Правая кнопка чаще всего используется для вызова контекстного меню. Оно появляется там, где находится курсор мыши, и набор его функций соответствует возможностям, доступным в этой области.

Трехкнопочная мышь не отличалась особой популярностью у пользователей Windows, пока Microsoft не выпустила модель IntelliMouse. Хотя эту мышь трудно считать интеллектуальной в прямом смысле этого слова, ее новый конструктивный элемент в виде небольшого колесика, расположенного между левой и правой кнопками, позволяет ввести в обращение множество дополнительных функций. При нажатии на колесико оно функционирует как средняя кнопка. Но кроме этого колесико можно прокручивать, и программы, поддерживающие эту функцию, будут реагировать прокруткой или масштабированием документа. Это намного удобнее, чем работать мышью на полосе прокрутки, так как колесико выполняет свои функции даже тогда, когда курсор мыши находится за пределами окна с документом.

Наличие мыши с колесиком можно определить при помощи вызова все той же функции **GetSystemMetrics**:

```
fMouseWheel=GetSystemMetrics (SM_MOUSEWHEELPRESENT);
```

Функция возвращает значение **TRUE**, если у мыши есть колесо.

Когда пользователь перемещает мышь, Windows передвигает по экрану небольшую растровую картинку, которая выступает в качестве *курсора мыши* (mouse cursor).

У курсора мыши есть *горячая точка* (hot spot) — пиксел, соответствующий положению курсора на экране. Когда говорят о позиции курсора мыши, то имеют в виду именно позицию горячей точки. Например, горячая точка стандартного курсора в виде стрелки (**IDC_ARROW**) находится на кончике стрелки. Другой курсор, в виде перекрестья (**IDC_CROSS**), имеет горячую точку в центре крестообразного шаблона.

Следует помнить, что курсор, используемый по умолчанию, для конкретного окна задается при определении структуры класса окна, например, следующим оператором:

wc.hCursor=LoadCursor (NULL, IDC_ARROW);

Действия пользователя с мышью принято описывать следующими терминами:

— *щелчок* — нажатие и отпускание кнопки мыши;

— *двойной щелчок* — быстрое двойное нажатие и отпускание кнопки мыши;

— *перетаскивание* — перемещение мыши при нажатой кнопке.

Кнопки трехкнопочной мыши называются *левой*, *средней* и *правой* кнопками. В связанных с мышью идентификаторах, которые определены в заголовочных файлах Windows, используются краткие обозначения, соответственно, **LBUTTON**, **MBUTTON** и **RBUTTON**. На двухкнопочной мыши имеются только левая и правая кнопки. Единственная кнопка однокнопочной мыши является левой.

5.6.7. Сообщения мыши

Изучая ввод с клавиатуры, мы видели, что Windows посылает сообщения клавиатуры только тому окну, которое имеет фокус ввода. С мышью Windows ведет себя несколько иначе. Оконная процедура получает сообщения мыши и когда мышь проходит через окно, и при щелчке внутри окна, даже если окно не активно или не имеет фокуса ввода.

Если мышь перемещается по клиентской области окна, то оконная процедура получает сообщение **WM_MOUSEMOVE**. Если кнопка мыши *нажимается* или *отпускается* внутри клиентской области, то оконная процедура получает следующие сообщения:

Кнопка	Нажатие	Отпускание	Нажатие (второй щелчок)
Левая	**WM_LBUTTONDOWN**	**WM_LBUTTONUP**	**WM_LBUTTONDBLCLC**
Средняя	**WM_MBUTTONDOWN**	**WM_MBUTTONUP**	**WM_MBUTTONDBLCLC**
Правая	**WM_RBUTTONDOWN**	**WM_RBUTTONUP**	**WM_RBUTTONDBLCLC**

Для всех этих сообщений значение параметра **lParam** содержит положение мыши. При этом в младшем слове находится значение координаты ***x***, а в старшем слове — значение координаты **y**. Отсчет координат ведется от левого угла клиентской области окна. Эти значения можно извлечь из **lParam** при помощи макросов **LOWORD** и **HIWORD**.

Значение параметра **wParam** показывает состояние кнопок мыши и клавиш **Shift** и **Ctrl**. Можно проверить параметр **wParam** при помощи соответствующих битовых масок:

```
MK_LBUTTON,   // Левая клавиша нажата
MK_MBUTTON,  // Средняя клавиша нажата
MK_RBUTTON,   // Правая клавиша нажата
MK_SHIFT,              // Клавиша Shift нажата
MK_CONTROL.  // Клавиша Ctrl нажата
```

Если пользователь щелкнет левой кнопкой мыши в клиентской области неактивного окна, то Windows сделает это окно активным, а затем передаст оконной процедуре сообщение **WM_LBUTTONDOWN**.

5.6.8. Обработка двойного щелчка

Два последовательных щелчка воспринимаются системой как двойной щелчок, если они происходят в течение достаточно короткого промежутка времени. Пользователь может самостоятельно задать этот интервал при помощи системного приложения **Мышь** (**Mouse**), которое можно найти в группе **Панель управления**.

Следует особо подчеркнуть, что окно будет получать сообщения о двойном щелчке (**DBLCLK**) только в том случае, если стиль соответствующего класса окна содержит флаг **CS_DBLCLKS**. Поэтому перед регистрацией окна класса нужно присвоить полю **style** структуры **WNDCLASS** значение, включающее этот флаг. Впрочем, если используется объект класса **KWnd**, то следует модифицировать стиль класса окна вызовом следующих инструкций:

```
style=GetClassLong (hWnd, GCL_STYLE);
SetClassLong (hWnd, GCL_STYLE, style | CS_DBLCLKS);
```

Если класс окна определен без флага **CS_DBLCLKS** и пользователь делает двойной щелчок левой кнопкой мыши, то оконная процедура последовательно получает сообщения **WM_LBUTTONDOWN**, **WM_LBUTTONUP**, **WM_LBUTTONDOWN** и **WM_LBUTTONUP**.

Если класс окна определен с флагом **CS_DBLCLKS**, то после двойного щелчка оконная процедура получит сообщения **WM_LBUTTONDOWN**, **WM_LBUTTONUP**, **WM_LBUTTONDBLCLK** и **WM_LBUTTONUP**. Легко заметить, что в этом случае в последовательности сообщений второе сообщение **WM_LBUTTONDOWN** заменяется сообщением **WM_LBUTTONDBLCLK**.

5.6.9. Обработка сообщений от колеса мыши

При нажатии на колесо мыши Windows генерирует такие же сообщения, какие вырабатываются при нажатии средней кнопки трехкнопочной мыши. Прокрутка же колесика вызывает сообщение **WM_MOUSEWHEEL**.

Если мы хотим обрабатывать это сообщение в коде программы, то в начале программного файла необходимо добавить следующую директиву:

```
#include <zmouse.h>
```

В заголовочном файле **zmouse.h** определены все необходимые константы.

Младшая часть параметра **wParam** сообщения **WM_MOUSEWHEEL** интерпретируется точно так же, как для других сообщений мыши, т. е. показывает состояние кнопок мыши и клавиш **Shift** и **Ctrl**.

Старшая часть параметра **wParam** содержит значение, отображающее дистанцию, пройденную колесом. Оно рассчитывается как количество шагов колеса при прокрутке, умноженное на коэффициент **WHEEL_DELTA**. В заголовочном файле **zmouse.h** этот коэффициент определен со значением 120.

Обработка сообщения **WM_MOUSEWHEEL** довольно проста, и ее пример можно найти в следующем параграфе.

5.6.10. Пример программы «Просмотрщик текста»

Рассмотрим приложение, предназначенное для чтения, вывода на экран и просмотра текстов. Назовем его «TextViewer». Приложение должно обрабаты-

вать сообщения мыши, а также обеспечивать для пользователя использование клавиатурного интерфейса. Второе требование означает, что все операции, выполняемые мышью, должны быть продублированы на клавиатуре.

Далее следует исходный текст приложения TextViewer. Приложение размещено в трех модулях: главном модуле **uTextViewer.cbproj**; модуле **uTVWindow.h(.cpp)** регистрации оконного класса, создания окна и показа окна на экране; модуле **uDocument.h(.cpp)** обработки текстового файла и вывода документа в клиентскую область окна.

```
//---------------------------------------------------------------------
// модуль uTextViewer.cbproj
//---------------------------------------------------------------------
#include <windows.h>
#include <zmouse.h>      // для обработки колесика мыши
#include «uTVWindow.h»
#include «uTVDocument.h»
#include <vcl.h>
#define FILE_NAME «C:\\Барков\\УчебникООПС++\\ProgramsAPI\\TextVi\\FILE.txt»
LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);
TTVDocument doc;
//---------------------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                        LPSTR lpCmdLine, int nCmdShow)
{
MSG msg;
if (!doc.Open(FILE_NAME)) return 0;
// если файл с текстом успешно открыт, создаем окно приложения
TTVWindow(«Просмотрщик текстовых файлов», hInstance, nCmdShow,
MainWndProc,
                        NULL, CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                          CS_HREDRAW | CS_VREDRAW,
                WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL);
// цикл обработки сообщений
while (GetMessage(&msg, NULL, 0, 0))
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
return msg.wParam;
}
//---------------------------------------------------------------------
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT uMsg,
                                          WPARAM wParam, LPARAM lParam)
{
  HDC hDC;
  PAINTSTRUCT PaintStruct;
  TEXTMETRIC TextMetric;         // Сведения о тексте
  int intClientHeight=0, intClientwidth=0;
```

```
  static int xInc, yInc;
  short status;

  switch (uMsg)
  {
    case WM_CREATE:                 //  Обработка создания окна
        hDC=GetDC(hWnd);          //  Чтение контекста окна
        GetTextMetrics(hDC, &TextMetric);    //  Чтение информации о тексте
        doc.Initialize(&TextMetric);        //  Инициализация текста
        ReleaseDC(hWnd, hDC);       //  Освобождение контекста окна
        break;
    case WM_SIZE:                   //  Обработка изменения размера окна
        hDC=GetDC(hWnd);          //  Чтение контекста окна
        intClientHeight=LOWORD(lParam);   //   Ширина клиентской области окна
        intClientwidth=HIWORD(lParam);   //  Высота клиентской области окна
        if (intClientHeight > 0) doc.ScrollSettings(hWnd, intClientHeight, intClientwidth);
        ReleaseDC(hWnd, hDC);       //  Освобождение контекста окна
        break;
     case WM_VSCROLL:               //  Обработка вертикального смещения текста
        switch (LOWORD(wParam))
        {
          case SB_LINEUP: yInc=-1; break;                  //  Стрелка вверх
          case SB_LINEDOWN: yInc=1; break;                  //  Стрелка вниз
          case SB_PAGEUP: yInc=-(int)doc.VScroll.nPage; break;      //  Page Up
          case SB_PAGEDOWN: yInc=(int)doc.VScroll.nPage; break;     //  Page Down
                                                                    // Вертикаль-
ный скроллинг
          case SB_THUMBTRACK: yInc=HIWORD(wParam) – doc.VScroll.nPos; break;
          case SB_TOP: yInc=-(int)doc.VScroll.nPage; break;        // Ctrl+Home
          case SB_BOTTOM: yInc=doc.VScroll.nMax-doc.VScroll.nPos; break; // Ctrl+End
          default: yInc=0;
        }
        doc.UpdateVscroll(hWnd, yInc);      // Изменить положение текста в окне
        break;
     case WM_HSCROLL:               // Обработка горизонтального смещения теста
        switch (LOWORD(wParam))
        {
          case SB_LINELEFT: xInc=-1; break;                  // Стрелка влево
          case SB_LINERIGHT: xInc=1; break;                   // Стрелка вправо
                                                              // Горизонталь-
ный скроллинг
          case SB_THUMBTRACK: xInc=HIWORD(wParam) – doc.HScroll.nPos; break;
          case SB_TOP: xInc=-doc.HScroll.nPos; break;                // Home
          case  SB_BOTTOM: xInc=doc.HScroll.nMax – doc.HScroll.nPos; break;   // End
          default: xInc=0;
        }
        doc.UpdateHscroll(hWnd, xInc);          // Изменить положение текста в окне
        break;
```

```
case WM_MOUSEWHEEL:                    // Колесико мыши
        if (LOWORD(wParam) & MK_SHIFT)
   {
     xInc=-3 * (short)HIWORD(wParam)/WHEEL_DELTA;
     doc.UpdateVscroll (hWnd, yInc);
         }
   else
   {
     yInc=-3 * (short)HIWORD(wParam)/WHEEL_DELTA;
     doc.UpdateVscroll(hWnd,yInc);
    }
   break;
case WM_KEYDOWN:                       // Нажатие клавиши
   switch (wParam)
   {                                                    // Стрелка вверх
     case VK_UP:    SendMessage (hWnd, WM_VSCROLL, SB_LINEUP, 0);
                    break;
                                                        // Стрелка вниз
     case VK_DOWN:  SendMessage (hWnd, WM_VSCROLL, SB_LINEDOWN, 0);
                    break;
                                                        // Стрелка влево
     case VK_LEFT:  SendMessage (hWnd, WM_HSCROLL, SB_LINELEFT, 0);
                    break;
                                                        // Стрелка вправо
     case VK_RIGHT: SendMessage (hWnd, WM_HSCROLL, SB_LINERIGHT, 0);
                    break;
                                                        //  Page Up
     case VK_PRIOR: SendMessage (hWnd, WM_VSCROLL, SB_PAGEUP, 0);
                    break;
                                                        //  Page Down
     case VK_NEXT:  SendMessage (hWnd, WM_VSCROLL, SB_PAGEDOWN, 0);
                    break;
     case VK_HOME:                      //  Home
                 status=GetKeyState(VK_CONTROL);
                 if (0x80 & status)
                     SendMessage (hWnd, WM_VSCROLL, SB_TOP, 0);
                 else SendMessage (hWnd, WM_HSCROLL, SB_TOP, 0);
                 break;
       case VK_END:                                     // End
                 status=GetKeyState(VK_CONTROL);
                 if (0x80 & status)
                     SendMessage (hWnd, WM_VSCROLL, SB_BOTTOM, 0);
                 else SendMessage (hWnd, WM_HSCROLL, SB_BOTTOM, 0);
                 break;
   }
   break;
case WM_PAINT:                         // Перерисовка формы
        hDC=BeginPaint(hWnd, &PaintStruct);
```

```
                doc.PutText(hWnd, hDC);
                EndPaint(hWnd, &PaintStruct);
                break;
    case WM_DESTROY: PostQuitMessage(0);    // Ликвидация окна
                break;
    default: return (DefWindowProc(hWnd, uMsg, wParam, lParam));
  };
  return 0;
}.
```

Есть два пути решения поставленной задачи. Обрабатывая сообщение **WM_KEYDOWN**, можно просто продублировать код, отвечающий за обработку сообщений **WM_VSCROLL** и **WM_HSCROLL**, но это некрасиво и непрактично с точки зрения дальнейшего сопровождения программы. Мы выберем другой путь: при появлении аппаратного сообщения клавиатуры, которое дублирует сообщения мыши, будем вызывать код, который создается для обработки сообщений с префиксом **SB**. Фактически это означает, что оконная процедура должна послать сама себе сообщение **SB**.

В Windows имеется несколько функций для отправки сообщений. В данной ситуации лучше использовать функцию **SendMessage**, имеющую следующий прототип:

LRESULT SendMessage (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);

Параметры функции имеют тот же смысл, что и параметры, передаваемые в оконную процедуру. Когда вызывается функция **SendMessage**, Windows вызывает оконную процедуру с дескриптором окна **hWnd**, передавая ей эти четыре параметра. После того как оконная процедура закончит обработку сообщения, Windows передает управление следующему за вызовом **SendMessage** оператору.

```
//---------------------------------------------------------------------
// модуль uTVWindow.h
//---------------------------------------------------------------------
#ifndef uTVWindowH
#define uTVWindowH
#include <windows.h>
//---------------------------------------------------------------------
class TTVWindow
{
  protected:
 HWND hWnd;
 WNDCLASSEX WndClass;
  public:
 TTVWindow(LPTSTR windowName, HINSTANCE hInst, int cmdShow,
           LRESULT (WINAPI* pWndProc) (HWND,UINT,WPARAM,LPARAM),
            LPTSTR menuName=NULL,
              int x=CW_USEDEFAULT, int y=0,
               int width=CW_USEDEFAULT, int height=0,
                UINT classStyle=CS_HREDRAW | CS_VREDRAW,
```

```
                    DWORD winStyle=WS_OVERLAPPEDWINDOW,
                      HWND hParent=NULL);
  HWND GetWnd() {return hWnd;}
};

//---------------------------------------------------------------------
// модуль uTVWindow.cpp
//---------------------------------------------------------------------
#include «uTVWindow.h»
//---------------------------------------------------------------------
TTVWindow::TTVWindow(LPTSTR windowName, HINSTANCE hInst, int cmdShow,
        LRESULT (WINAPI* pWndProc) (HWND,UINT,WPARAM,LPARAM),
        LPTSTR menuName, int x, int y, int width, int height,
        UINT classStyle, DWORD windowStyle, HWND hParent)
{
char szClassName[]=«WindowClass»;
WndClass.cbSize    =sizeof(WndClass);
WndClass.style    =classStyle;
WndClass.lpfnWndProc=pWndProc;
WndClass.cbClsExtra=0;
WndClass.cbWndExtra=0;
WndClass.hInstance=hInst;
WndClass.hIcon=LoadIcon (NULL, IDI_APPLICATION);
WndClass.hCursor=LoadCursor (NULL, IDC_ARROW);
WndClass.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH);
WndClass.lpszMenuName=menuName;
WndClass.lpszClassName=szClassName;
WndClass.hIconSm=LoadIcon (NULL, IDI_APPLICATION);
// Регистрируем класс окна
if (!RegisterClassEx(&WndClass))
        {
          char msg[100]=«Невозможно зарегистрировать окно: «;
          strcat(msg, szClassName);
          MessageBox(NULL, msg, «Ошибка», MB_OK);
          return;
        }
// Создаем окно
hWnd=CreateWindow (szClassName, windowName, windowStyle,
                                    x, y, width, height, hParent,
                                          (HMENU)NULL, hInst, NULL);
if (!hWnd)
  {
          char text[100]=«Невозможно создать окно: «;
          strcat (text, windowName);
          MessageBox (NULL, text, «Ошибка», MB_OK);
          return;
  }
// Показываем окно
```

```
ShowWindow (hWnd, cmdShow);
}.
```

Обратим внимание на некоторые моменты предложенного решения:

— заголовок конструктора класса **TTVWindow** содержит 12 параметров. Первые четыре параметра являются обязательными, остальные имеют значения по умолчанию;

— четвертый параметр является указателем на функцию, принимающую четыре аргумента и возвращающую значение типа **LRESULT**. Здесь обязателен спецификатор **WINAPI** или эквивалентный ему спецификатор **__stdcall**;

— код, отвечающий за подготовку и создание окна, размещен в теле конструктора класса **TTVWindow**.

```
//---------------------------------------------------------------------------
// модуль uTVDocument.h
//---------------------------------------------------------------------------
#ifndef uTVDocumentH
#define uTVDocumentH
#include <windows.h>
#include <vector>
#include <string>
using namespace std;
class TTVDocument
{
  private:
 vector<string> lines; // вектор для хранения строк документа
  public:
 BOOL Open (const char* file);          // открытие файла с документом
 void Initialize (LPTEXTMETRIC tm);
 void ScrollSettings (HWND hwnd, int width, int height);
 void UpdateHscroll (HWND hwnd,int xInc);
 void UpdateVscroll (HWND hwnd, int yInc);
 void PutText (HWND hwnd, HDC hdc);
 int intCharWidth;       // Средняя ширина символа
 int intLineStep;        // высота (шаг) строки
 int intLineMaxLen;   // максимальная длина строки
 SCROLLINFO VScroll;   // вертикальный скроллинг
 int intVScrollRange;    //  диапазон вертикальной полосы прокрутки
 SCROLLINFO HScroll;   //  горизонтальный скроллинг
 int intHScrollRange;    //  диапазон горизонтальной полосы прокрутки
};
//---------------------------------------------------------------------------
#endif
//---------------------------------------------------------------------------
// модуль uTVDocument.cpp
//---------------------------------------------------------------------------
//#pragma hdrstop
#include <Windows.h>
#include <fstream>
```

```
#include «uTVDocument.h»
//---------------------------------------------------------------------
BOOL TTVDocument::Open(const char* file)
{
ifstream finp(file);
char buf[200];
if (!finp.good())
  {
     MessageBox(NULL, «Не найден входной файл», «Ошибка», MB_OK);
     return FALSE;
  }
// Прочитать файл, сохранив информацию в векторе строк lines
  while (!finp.eof())
          {
             finp.getline(buf, 199);
             buf[199]=0;
             lines.push_back(string(buf));
          }
// Вычислить максимальную длину строки
  intLineMaxLen=0;
  for (int i=0; i < lines.size(); i++)
  {
     int lineLen=lines[i].size();
// Корректировка длины строки, если строка содержит символы табуляции
     int iTabPos=0;
     while (1)
         {
           iTabPos=lines[i]. find('\t', iTabPos);
           if (iTabPos != -1)
           {
              lineLen +=8;
              iTabPos++;
           }
           else break;
         }
     if (lineLen > intLineMaxLen) intLineMaxLen=lineLen;
  }
  return TRUE;
}
//---------------------------------------------------------------------
void TTVDocument::Initialize(LPTEXTMETRIC tm)
{
   intCharWidth=tm–>tmAveCharWidth;
   intLineStep=tm–>tmHeight+tm–>tmExternalLeading;
   VScroll.nMin=VScroll.nPos=0;
   HScroll.nMin=HScroll.nPos=0;
}
//---------------------------------------------------------------------
```

```
void TTVDocument::ScrollSettings (HWND hwnd, int width, int height)
{
// Вертикальный скроллинг
  VScroll.cbSize=sizeof(VScroll);
  VScroll.fMask=SIF_RANGE | SIF_PAGE | SIF_POS;
  VScroll.nPage=height/intLineStep – 1;          // количество строк
  VScroll.nMax=lines.size() – 1;
  if (VScroll.nPage > VScroll.nMax) VScroll.nPos=VScroll.nMin;
  intVScrollRange=VScroll.nMax – VScroll.nMin+1;
  SetScrollInfo (hwnd, SB_VERT, &VScroll, TRUE);
// Горизонтальный скроллинг
  HScroll.cbSize=sizeof(SCROLLINFO);
  HScroll.fMask=SIF_RANGE | SIF_PAGE | SIF_POS;
  HScroll.nPage=width/intCharWidth – 2;
  HScroll.nMax=intLineMaxLen;
  if (HScroll.nPage > HScroll.nMax) HScroll.nPos=HScroll.nMin;
  intHScrollRange=HScroll.nMax – HScroll.nMin+1;
  SetScrollInfo (hwnd, SB_HORZ, &HScroll, TRUE);
}
//---------------------------------------------------------------------
void TTVDocument::UpdateVscroll(HWND hwnd, int yInc)
{
// Ограничение на положительное приращение
   yInc=min (yInc, intVScrollRange – (int)VScroll.nPage – VScroll.nPos);
// Ограничение на отрицательное приращение
   yInc=max(yInc, VScroll.nMin – VScroll.nPos);
   if (yInc)
  {
     ScrollWindow (hwnd, 0, -intLineStep * yInc, NULL, NULL);
     VScroll.nPos += yInc;
     SetScrollInfo (hwnd, SB_VERT, &VScroll, TRUE);
     InvalidateRect (hwnd, NULL, TRUE);
     UpdateWindow (hwnd);
  }
}
//---------------------------------------------------------------------
void TTVDocument::UpdateHscroll(HWND hwnd, int xInc)
{
// Ограничение на положительное приращение
    xInc=min (xInc, intHScrollRange – (int)HScroll.nPage – HScroll.nPos);
// Ограничение на отрицательное приращение
    xInc=max(xInc, HScroll.nMin – HScroll.nPos);
    if (xInc)
    {
      ScrollWindow (hwnd, 0, -intCharWidth * xInc, NULL, NULL);
      HScroll.nPos += xInc;
      SetScrollInfo (hwnd, SB_HORZ, &HScroll, TRUE);
      InvalidateRect (hwnd, NULL, TRUE);
```

```
      UpdateWindow (hwnd);
  }
}
//---------------------------------------------------------------------------
void TTVDocument::PutText(HWND hwnd, HDC hdc)
{
   RECT rect;
   GetClientRect(hwnd, &rect);
   rect.left += intCharWidth;
   rect.right -= intCharWidth;
   HRGN hRgn=CreateRectRgnIndirect (&rect);
   SelectClipRgn (hdc, hRgn);
   int x=intCharWidth * (HScroll.nMin – HScroll.nPos+1);
   int y=intLineStep;
   int amountLines=lines.size();
   int iBeg=VScroll.nPos;
   int iEnd=(VScroll.nPos+VScroll.nPage < amountLines)? VScroll.nPos+VScroll.nPage :

amountLines;
   for (int i=iBeg; i < iEnd; ++i)
  {
     int iTabPos=lines[i].find('\t');
     if (-1 == iTabPos) TextOut (hdc, x, y, lines[i].c_str(), lines[i].size());
     else TabbedTextOut (hdc, x, y, lines[i].c_str(), lines[i].size(), 0, 0, x);
     y += intLineStep;
  }
  SelectClipRgn (hdc, NULL);
}
//---------------------------------------------------------------------------
#pragma package(smart_init).
```

Контрольные вопросы

1. Для лучшего понимания рекомендуется перенести программу на компьютер и проверить режимы работы.

Задачи

I. Задания по тексту главы

1. Введите и отладьте программу из п. 5.3.1 «Простейшая программа для Windows».
2. Введите и отладьте программу из п. 5.3.2 «Полноценная программа для Windows».
3. Разработайте программу, выводящую в окно стандартные графические фигуры из п. 5.5.
4. Введите и отладьте программу из п. 5.6.10 «Просмотрщик текста». Проследите в шаговом режиме отладчика последовательность обработки клавиш клавиатуры и мыши.

5. Расширьте возможности программы «Просмотрщик текста» возможностями выделения фрагмента текста.

В каждой следующей задаче требуется реализовать в том или ином виде определение нового класса. В каждом параграфе приводятся конкретные требования к программированию классов параграфа. Для демонстрации работы с объектами нового типа во всех заданиях нужно написать главную функцию и оконную процедуру.

II. Структура с двумя полями

В задачах этого параграфа используется структура с двумя полями: first и second. Требуется реализовать тип данных с помощью такой структуры. Во всех заданиях обязательно должны присутствовать:

— метод инициализации Init; метод должен контролировать значения аргументов на корректность;

— метод Read для ввода значений аргументов с клавиатуры или с помощью полей класса TEdit экранной формы;

— метод Display для вывода значений полей структуры на экран.

Список заданий:

II.1. Поле first — дробное число; поле second — целое число, показатель степени. Реализовать метод Power() возведения числа first в степень second. Метод должен правильно работать при любых допустимых значениях first и second.

II.2. Поле first — дробное число; поле second — дробное число, показатель степени. Реализовать метод Power() возведения числа first в степень second. Метод должен правильно работать при любых допустимых значениях first и second.

II.3. Поле first — целое положительное число, числитель; поле second — целое положительное число, знаменатель. Реализовать метод IntegerPart() — выделение целой части дроби first/second. Необходимо проверять неравенство знаменателя нулю.

II.4. Поле first — целое положительное число, номинал купюры; номинал может принимать значения 1, 2, 5, 10, 50, 100, 1000, 5000. Поле second — целое положительное число, количество купюр данного достоинства. Реализовать метод Summa() — вычисления денежной суммы.

II.5. Поле first — целое положительное число, калорийность 100 г продукта. Поле second — дробное положительное число, масса продукта в килограммах. Реализовать метод Power() — вычисления общей калорийности продукта.

II.6. Поле first — дробное положительное число, цена товара; поле second — целое положительное число, количество единиц товара. Реализовать метод Cost() — вычисление стоимости товара.

II.7. Поле first — дробное число, левая граница диапазона; поле second — дробное число, правая граница диапазона. Реализовать метод RangeCheck() — проверку заданного числа на принадлежность диапазону.

II.8. Поле first — целое число, левая граница диапазона, включается в диапазон; поле second — целое число, правая граница диапазона, не включается в диапазон. Пара чисел представляет полуоткрытый интервал [first, second]. Реализовать метод RangeCheck() — проверку заданного числа на принадлежность диапазону.

II.9. Поле first — целое положительное число, часы; поле second — целое положительное число, минуты. Реализовать метод Minutes() — приведение времени в минуты.

II.10. Линейное уравнение y=Ax+B. Поле first — дробное число, коэффициент A; поле second — дробное число, коэффициент B. Реализовать метод Function() — вычисление для заданного x значения функции y.

II.11. Линейное уравнение y=Ax+B. Поле first — дробное число, коэффициент A; поле second — дробное число, коэффициент B. Реализовать метод Root() — вычисление корня линейного уравнения. Необходимо проверять неравенство коэффициента B нулю.

II.12. Поле first — дробное число, координата x точки на плоскости; поле second — дробное число, координата y точки на плоскости. Реализовать метод Distance() — расстояние точки от начала координат.

II.13. Поле first — дробное положительное число, катет a прямоугольного треугольника; поле second — дробное положительное число, катет b прямоугольного треугольника. Реализовать метод Hipotenuse() — вычисление гипотенузы.

II.14. Поле first — дробное положительное число, оклад; поле second — целое положительное число, количество отработанных дней в месяце. Реализовать метод Summa() — вычисление начисленной суммы за заданное количество дней для заданного месяца:

оклад/дни_месяца × отработанные дни.

II.15. Поле first — целое положительное число, продолжительность телефонного разговора в минутах; поле second — дробное положительное число, стоимость одной мины в рублях. Реализовать метод Cost() — вычисление общей стоимости разговора.

II.16. Поле first — целое число, целая часть числа; поле second — дробное положительное число, дробная часть числа. Реализовать метод Multiply() — умножение на произвольное дробное число типа double. Метод должен правильно работать при любых допустимых значениях first и second.

II.17. Поле first — целое положительное число, координата курсора по горизонтали; поле second — целое положительное число, координата курсора по вертикали. Реализовать метод ChangeX() — изменение горизонтальной координаты курсора. Реализовать метод ChangeY() — изменение вертикальной координаты курсора. Методы должны проверять выход за границу экрана (или формы).

II.18. Поле first — целое число, целая часть числа; поле second — целое положительное число, дробная часть числа. Реализовать метод Multiply() — умножение на произвольное целое число типа int. Метод должен правильно работать при любых допустимых значениях first и second.

II.19. Число сочетаний по k объектов из n объектов (k < n) вычисляется по формуле

$$C(n,k)=\frac{n!}{(n-k)!\cdot k!}.$$

II.20. Поле first — целое положительное число, k; поле second — целое положительное число, n. Реализовать метод CombinationX() — вычисление

C(n, k). Элемент a_j геометрической прогресии вычисляется по формуле $a_j=a_0 r^j$, j=0, 1, 2, ... Поле first — дробное число, первый элемент прогрессии a_0; поле second — постоянное отношение r. Реализовать метод ElementJ() — вычисление заданного элемента прогрессии.

III. Структуры и классы

Во всех заданиях, помимо указанных в задании операций, обязательно должны быть реализованы следующие методы:

— метод инициализации Init;

— метод Read для ввода значений аргументов с клавиатуры или с помощью полей класса TEdit экранной формы;

— метод Display для вывода значений полей структуры на экран;

— преобразование в строку ToString.

Все задания должны быть реализованы тремя способами:

— тип данных представляется структурой с необходимыми полями, а операции реализуются как внешние функции, которые получают объекты данного типа в качестве аргументов;

— тип данных представляется как класс с закрытыми полями, где операции реализуются как методы класса;

— инкапсулировать поля класса в независимой структуре и в ней реализовать методы Init(), Read(), Display(), ToString(); в основном классе должно быть одно поле данных, представленное объектом-структурой.

Список заданий:

III.1. Комплексное число представляется парой целых чисел (a, b), где a — действительная часть, b — мнимая часть. Реализовать класс TComplex для работы с комплексными числами. Обязательно должны присутствовать операции:

a. сложения add,
$$(a, b)+(c, d)=(a+c, b+d);$$
b. вычитания sub,
$$(a, b) – (c, d)=(a – c, b – d);$$
c. умножения mul,
$$(a, b)\times(c, d)=(ac – bd, ad+bc);$$
d. деления div,
$$(a, b)/(c, d)=(ac+bd, bc – ad)/(c2+d2);$$
e. сравнения equ,
$$(a, b)=(c, d), \text{ если } (a=c) \text{ и } (b=d);$$
f. сопряженное число,
$$conj(a, b)=(a, –b).$$

III.2. Создать класс TVvector3D, задаваемый тройкой координат. Обязательно должны быть реализованы: сложение и вычитание векторов, скалярное произведение векторов, умножение на скаляр, сравнение векторов, вычисление длины вектора, сравнение длины векторов.

III.3. Создать класс TModelWindow для работы с моделями экранных окон. В качестве полей задаются: заголовок окна, координаты левого верхнего

угла, размер по горизонтали, размер по вертикали, цвет окна, состояние «видимое/невидимое», состояние «с рамкой/без рамки». Координаты и размеры указываются в целых числах (пикселах). Реализовать операции: передвижение окна по горизонтали, по вертикали; изменение высоты и/или ширины окна; изменение цвета; изменение состояния; опрос состояния. Операции передвижения и изменения размера должны осуществлять проверку на пересечение границ экрана. Функция вывода на экран должна индуцировать состояние полей объекта.

III.4. Создать класс TMoney для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа unsigned char — для копеек. Дробная часть (копейки) при выводе на экран должна быть отделена от целой части запятой. Реализовать сложение, вычитание, деление сумм, деление суммы на дробное число, умножение на дробное число и операции сравнения.

III.5. Создать класс TTriangle для представления треугольника. Поля данных должны включать углы и стороны. Требуется реализовать операции: получения и изменения полей данных, вычисления площади, вычисления периметра, вычисления высот, а также определения вида треугольника (равносторонний, равнобедренный или прямоугольный).

III.6. Создать класс TAngle для работы с углами на плоскости, задаваемыми величиной в градусах и минутах. Обязательно должны быть реализованы: перевод в радианы, приведение к диапазону 0–360, увеличение и уменьшение угла на заданную величину, получение синуса, сравнение углов.

III.7. Создать класс TPoint для работы с точками на плоскости. Координаты точки — декартовы. Обязательно должны быть реализованы: перемещение точки по оси X, перемещение точки по оси Y, определение расстояния между двумя точками, преобразование в полярные координаты, сравнение на совпадение и несовпадение.

III.8. Рациональная (несократимая) дробь представляется парой целых чисел (a, b), где a — числитель, b — знаменатель. Создать класс TRational для работы с рациональными дробями. Обязательно должны быть реализованы операции:

a. сложения add,
$$(a, b)+(c, d)=(ad+bc, bd);$$
b. вычитания sub,
$$(a, b)-(c, d)=(ad-bc, bd);$$
c. умножения mul,
$$(a, b)\times(c, d)=(ac, bd);$$
d. деления div,
$$(a, b)/(c, d)=(ad, bc);$$
e. сравнения equal, greate, less.

Должна быть реализована приватная функция сокращения дроби reduce, которая вызывается при выполнении арифметических операций.

III.9. Создать класс TDate для работы с датами в формате «год.месяц.день». Дата представляется структурой с тремя полями unsigned int: для года, месяца и дня. Класс должен включать не менее трех функций

инициализации: числами, строкой вида «год.месяц.день» (например, «2016.03.31») и датой. Обязательными операциями являются: вычисление даты через заданное количество дней, вычитание заданного количества дней из даты, определение високосности года, присвоение и получение отдельных частей (год, месяц, день), сравнение дат (равно, до, после), вычисление количества дней между датами.

III.10. Создать класс TTime для работы со временем в формате «час:минута:секунда». Класс должен включать в себя не менее четырех функций инициализации: числами, строкой (например, «21:45:59»), секундами и временем. Обязательными операциями являются: вычисление разницы между двумя моментами времени в секундах, сложение времени и заданного количества секунд, вычитание из времени заданного количества секунд, сравнение моментов времени, перевод в секунды, перевод в минуты (с округлением до целой минуты).

III.11. Реализовать класс TFazzyNumber для работы с нечеткими числами, которые представляются тройками чисел ($x - e_1$, x, $x+e_2$). Для чисел $A=(A - a_l, A, A+a_r)$ и $B=(B - b_l, B, B+b_r)$ арифметические операции выполняются по следующим формулам:

a. сложение
$$A+B=(A+B - al - bl, A+B, A+B+ar+br);$$
b. вычитание
$$A - B=(A - B - al - bl, A - B, A - B+ar+br);$$
c. умножение
$$A\times B=(A\times B - B\times al - A*bl+al\times bl, A\times B, A\times B+B\times ar+A\times br+ar\times br);$$
d. обратное число
$$A=(1/(A+ar), 1/A, 1/(A - br)), A>0;$$
e. деление
$$A/B=((A+a_r)/(B+b_r), A/B, (A+a_r)/(B - b_l)), B>0.$$

III.12. Реализовать класс TAccount, представляющий собой банковский счет. В классе должны быть четыре поля: фамилия владельца, номер счета, процент начисления и сумма в рублях. Открытие нового счета выполняется операцией инициализации. Необходимо выполнять следующие операции: сменить владельца счета, снять некоторую сумму денег со счета, положить деньги на счет, начислить проценты, перевести сумму в доллары, перевести сумму в евро, получить сумму прописью (преобразовать в числительное).

III.13. Номиналы российских рублей могут принимать значения 1, 2, 5, 10, 50, 100, 500, 1000, 5000. Копейки представить как 0.01 (1 копейка), 0.05 (5 копеек), 0.10 (10 копеек), 0.5 (50 копеек). Создать класс TMoney для работы с денежными суммами. Сумма должна быть представлена полями-номиналами, значениями которых должно быть количество купюр данного достоинства. Реализовать сложение сумм, вычитание сумм, деление сумм, деление суммы на дробное число, умножение на дробное число и операции сравнения. Дробная часть (копейки) при выводе на экран должна быть отделена от целой части запятой.

III.14. Реализовать класс банкомат (TCashMashine). В классе должны содержаться поля хранения идентификационного номера банкомата, информации о текущей сумме денег, оставшейся в банкомате, минимальной и максимальной суммах, которые разрешается снять клиенту в один день. Сумма денег представляется полями-номиналами 10–1000, значениями которых должно быть количество купюр данного достоинства. Реализовать метод инициализации банкомата, метод загрузки купюр в банкомат и метод снятия определенной суммы денег. Метод снятия денег должен выполнять проверку на корректность снимаемой суммы: она не должна быть меньше минимального значения и не должна превышать максимальное значение. Метод ToString() должен преобразовывать в строку сумму денег, оставшуюся в банкомате.

III.15. Создать класс TFraction для работы с дробными числами. Число должно быть представлено двумя полями: целая часть — длинное целое со знаком, дробная часть — беззнаковое короткое целое. Реализовать арифметические операции сложения, вычитания, умножения, деления и операции сравнения.

III.16. Создать класс TGoods (товар). В классе должны быть представлены поля: наименование товара, дата оформления, цена товара, количество единиц товара, номер накладной, по которой товар поступил на склад. Реализовать методы изменения цены товара, изменения количества товара (увеличения и уменьшения), вычисления стоимости товара. Метод ToString() должен выдавать в виде строки стоимость товара.

III.17. Создать класс TBitString для работы с 64-битовыми строками. Битовая строка должна быть представлена двумя полями типа unsigned long. Должны быть реализованы все традиционные операции для работы с битами: and, or, xor, not. Реализовать сдвиг влево ShiftLeft(), сдвиг вправо ShiftRight() на заданное количество битов.

III.18. Создать класс TLongLong для работы с числами из 64 бит. Число должно быть представлено двумя полями: long — старшая часть, unsigned long — младшая часть. Должны быть реализованы арифметические операции, присутствующие в C++ (без присваивания), и сравнения.

III.19. Создать класс TSalary (зарплата). В классе должны быть представлены поля: фамилия-имя-отчество, оклад, год поступления на работу, процент надбавки, подоходный налог, количество отработанных дней в месяце, количество рабочих дней в месяце, начисленная и удержанная суммы. Реализовать методы: вычисления начисленной суммы, вычисления удержанной суммы, вычисления суммы, выдаваемой на руки, вычисления стажа. Стаж вычисляется как полное количество лет, прошедших от года поступления на работу, до текущего года. Начисления представляют собой сумму, начисленную за отработанные дни, и надбавки, т. е. доли от первой суммы. Удержания представляют собой отчисления в пенсионный фонд (1% от начисленной суммы) и подоходный налог. Подоходный налог составляет 13% от начисленной суммы без отчислений в пенсионный фонд.

III.20. Реализовать класс TCursor. Полями являются координаты курсора по горизонтали и вертикали целые положительные числа, вид курсора — целое

от 1 до 15, размер курсора по горизонтали и вертикали — целые положительные числа. Реализовать методы изменения координат курсора, изменения вида курсора, изменения размера курсора, метод гашения и восстановления курсора.

IV. Классы и объекты

Во всех задачах требуется реализовать по два-три класса. Один класс является основным, остальные — вспомогательные. Вспомогательные классы должны быть определены как независимые. Объекты вспомогательных классов должны использоваться в качестве полей основного класса.

Список заданий:

IV.1. Реализовать класс TAccount (задание III.12), используя для представления суммы класс TMoney из задания II.4.

IV.2. Реализовать класс TAccount (задание III.12), используя для представления суммы класс TMoney из задания III.13.

IV.3. Реализовать класс TAccount (задание III.12). Добавить поле-дату открытия счета, используя класс TDate из задания III.9. Добавить метод, вычисляющий количество дней, прошедших с начала открытия счета, и добавляющий по 0.01% к проценту начисления за каждый день.

IV.4. Реализовать класс TCalculator с полным набором арифметических операций, используя класс TFraction из задания III.15.

IV.5. Реализовать класс TBankomat (задание III.14), используя для представления суммы класс TMoney из задания III.13.

IV.6. Реализовать класс TFraction (задание III.15), используя для представления целой части класс TLongLong из задания III.18, а для представления дробной части положительное дробное число типа double.

IV.7. Реализовать класс TCalculator с полным набором арифметических операций, на основе класса TFraction из предыдущего задания.

IV.8. Реализовать класс TTriangle (задание III.5), опираясь на класс TAngle из задания III.6 для представления углов.

IV.9. Реализовать класс TGoods (задание III.16), добавив поле-дату поступления товара на склад (использовать класс TDate из задания III.9). Реализовать метод, вычисляющий срок хранения товара.

IV.10. Реализовать класс TGoods (задание III.16), используя для представления цены класс TMoney из задания III.4. Реализовать метод уценки товара, уменьшая цену на 1% за каждый день просрочки срока годности.

IV.11. Реализовать класс TTriangle (задание III.5) с полями-координатами вершин. Для представления координат вершин используйте класс TPoint из задания III.7.

IV.12. Реализовать класс TSalary (задание III.19), используя вместо полягода поле-дату класса TDate из задания III.9. Стаж следует вычислять, используя методы класса TDate.

IV.13. Реализовать класс TSalary (задание III.19), используя для представления полей начислений и удержаний класс TMoney из задания III.4.

IV.14. Реализовать класс TMoney (задание III.4), используя класс TFraction из задания III.15.

IV.15. Реализовать класс TModelWindow (задание III.3), добавив поле для курсора. Используйте для представления поля курсора класс TCursor из задания III.20.

IV.16. Реализовать класс TTelephonePainment, представляющий собой разовый платеж за телефонный разговор. Класс должен включать поля: фамилия плательщика, номер телефона, тариф за минуту разговора, скидка (в процентах), время начала разговора, время окончания разговора, сумма к оплате. Для представления времени используйте класс TTime из задания II.10. Реализовать методы извлечения и изменения полей. Время разговора, подлежащее оплате, вычисляется в минутах; неполная минута считается за полную. Метод toString() должен выдавать сумму в рублях.

IV.17. Реализовать класс TSet (множество) не более чем из 64 элементов целых чисел, используя класс TBitString из задания III.17. Множество должно обеспечивать включение элемента во множество, исключение элемента из множества, объединение, пересечение и разность множеств, вычисление количества элементов в множестве.

IV.18. Реализовать класс TRational (задание II.8), используя для представления числителя и знаменателя класс TLongLong из задания II.18.

IV.19. Реализовать класс TMoney (задание III.4), используя для представления рублей класс TLongLong из задания III.18.

IV.20. Реализовать класс TCursor (задание III.20), используя для представления координат класс TLongLong из задания III.18.

ЛАНЬ®

ЛАНЬ®

ЛАНЬ®

ЧАСТЬ II
РЕАЛИЗАЦИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ ТЕХНОЛОГИИ В СИСТЕМЕ ПРОГРАММИРОВАНИЯ

В первой части книги рассмотрены идеи и принципы объектно-ориентированного программирования, а также приведены сведения, необходимые для полноценного освоения объектно-ориентированного программирования.

Вторая часть посвящена изучению методов реализации идей и принципов объектно-ориентированного программирования в системе программирования. Нашей целью будет изучение устройства объектно-ориентированной программы и процессов ее функционирования. Полученные знания используются для освоения приемов, возможностей и традиций объектно-ориентированного программирования. Приемы программирования поясняются графической иллюстрацией происходящих в программе процессов.

6. УСТРОЙСТВО И ФУНКЦИОНИРОВАНИЕ ПРОГРАММЫ

Тот, кто обещает легкие способы обучения C++*, либо лжет, либо сам не понимает сложности данной задачи.*

В. Штерн

Заканчивается этап проектирования, получено описание программного изделия. Перед программистами стоят задачи изготовления программы. Для этого необходимо выбрать язык и систему программирования. В нашем случае выбор уже сделан — это язык программирования C++Builder и система программирования RAD Studio.

Семантика (смысл) любого языка программирования проявляется в тех действиях, которые изменяют состояние вычислительной системы при интерпретации законченного предложения исходного языка. В структурном программировании семантику, например, оператора цикла или оператора выбора можно было представить в виде блок-схемы, которая в данном случае показывала реализацию оператора на абстрактном, но достаточном для осмысленного программирования уровне. Например, смысл оператора цикла с предусловием вида **while** (<Условие>) <Оператор> можно представить следующим образом.

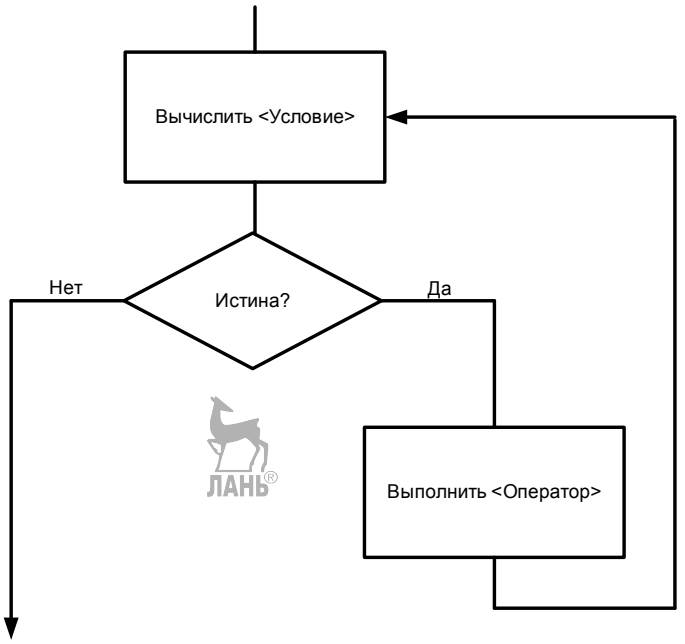


Рис. 6.1
Семантика оператора цикла

Для представления семантики данных (например, массива или записи) можно было изобразить наглядную картинку из прямоугольников с набором текстовых и числовых обозначений. В более сложных случаях можно добавить связи, отображающие, например, структурные отношения.

Обычно язык программирования позволяет реализовать наш замысел несколькими вариантами лингвистических средств. Перед квалифицированным программистом встает задача выбора конкретного лингвистического средства (оператора, описателя данных), наиболее подходящего для реализации замысла. Примером может служить выбор отвечающего конкретной ситуации оператора цикла (**for**, **while** с предусловием или **while** с постусловием). Для этого программист должен иметь возможность оценить семантику лингвистического средства: мысленно заглянуть внутрь исходной программы, промоделировать действия вычислительной системы и, наконец, принять решение о выборе соответствующей ситуации конструкции языка программирования. Таким образом, профессиональное программирование обычно требует знания внутреннего устройства, реализации данных и операций.

Современным системам присущи как функциональная, так и композиционная сложность. Основными способами преодоления сложности являются декомпозиция и моделирование.

В главе 2 рассмотрен унифицированный язык моделирования UML. Напомним, что назначением языка UML является моделирование сложных систем с помощью наглядных средств. Набор из нескольких наглядных моделей создает в сознании специалистов интегральный образ сложной прикладной системы, которую они совместно проектируют. Считается, что язык UML является средством профессионального мышления программиста-системотехника. Вместе с тем, наглядные модели служат эффективным средством документирования автоматизированных систем, а также языком общения между программистами, системными аналитиками и заказчиками автоматизированных систем.

Язык UML создавался для поддержки этапов анализа и проектирования программных изделий. Перед нами сейчас стоит задача изготовления программы, поэтому нам необходимы средства, с помощью которых можно увидеть реализацию принципов объектно-ориентированного программирования в системе программирования и создаваемой программе, тщательно продумать внутреннее устройство программы. Для этой цели в последующих параграфах мы будем интенсивно использовать понятия: «структура класса» и «структура объекта». В качестве базового приема мы опять используем абстрагирование. «Структура класса» и «структура объекта» введены с целью придания наглядности, образности демонстрации наиболее существенных отношений в программе: внутренней структуры классов и объектов, взаимодействия классов, взаимодействия объектов, взаимодействия классов и объектов. Добавим к этому и взаимодействие с операционной системой.

Квалифицированный программист должен хорошо знать свой рабочий инструмент. В любой системе объектно-ориентированного программирования имеются аналогичные «структуре класса» и «структуре объекта» образования, однако их реализация отличается от рассматриваемых в настоящей книге

структур. Введению структуры класса и структуры объекта и методического приема их использования для обучения программированию способствовали следующие соображения.

Во-первых, устройство классов и объектов широко не документируется авторами систем программирования и изменяется от версии к версии. Хотя, если судить по публикациям, любознательные программисты осваивают детали их реализации по исходным текстам модулей системы программирования.

Во-вторых, мы не ставим перед собой задачу рассмотреть детали реализации RAD Studio. Логика структур данных, в которых моделируются основные идеи объектно-ориентированного программирования, логика использования приемов объектно-ориентированного программирования находятся в центре внимания.

ЛАНЬ®

В-третьих, мы будем придерживаться принципа доступности изложения, чему в немалой степени способствует абстракция структур данных, которая позволила отвлечься от деталей их реализации и выдвинуть на передний план принципы обработки данных в объектно-ориентированных системах программирования и проследить в наглядной форме реализацию идей объектно-ориентированного программирования.

Таким образом, мы имеем теперь наглядные изобразительные средства, с помощью которых можем осуществлять декомпозицию и моделирование объектно-ориентированной программы.

Вначале мы рассмотрим логику внутреннего устройства класса и объекта, а затем будем использовать введенные структуры для освоения методики объектно-ориентированного программирования.

6.1. Структура класса и структура объекта

Структура класса содержит всю необходимую информацию о классе и располагается в динамической памяти. Структура класса приведена на рисунке 6.2. В верхней части рисунка находится последовательность полей, которые содержат в основном ссылки на другие структуры. Рассмотрим подробнее содержание указанной последовательности полей и компонентов класса, на которые имеются ссылки.

Поле «Имя класса» содержит ссылку на данное строкового типа, в котором хранится имя класса.

Поле «Размер экземпляра класса» содержит информацию целого типа, используемую при создании объекта для выделения необходимого объема динамической памяти.

Поле «Класс-предок» содержит ссылку на аналогичную структуру предка рассматриваемого класса. С помощью этого поля реализуется отношение наследования.

ЛАНЬ®

Следующие пять ссылок адресуют таблицы данных (полей и свойств) и таблицы методов (статических, виртуальных, методов класса).

В каждой строке таблицы полей содержится имя поля, его тип и ссылка на реализацию поля. В структуре класса реализация полей и свойств отсутствует.

Таблица свойств содержит поля имени и типа, а также ссылки на реализацию полей или методов, используемые при чтении и записи значения свойства.

Таблицы методов содержат имена методов и пустую ссылку на реализацию метода. В структуре класса реализация методов в основном отсутствует, исключение составляют методы класса и конструктор. Конструктор представляет собой специальный метод, используемый для построения экземпляра класса (объекта). Основным признаком присутствия реализации метода в структуре класса является возможность вызова метода непосредственно из структуры класса.

Поле «Деструктор» содержит в структуре класса пустую ссылку на реализацию метода ликвидации объекта. Реализация деструктора появляется в структуре объекта.

Структура класса отражает рассмотренные в главе 4 алгебраические принципы построения типа данных. Напомним основные моменты. Во-первых, сигнатура типа данных включает список основ и список операций; поля и свойства соответствуют нульарным операциям, методы — операциям большей арности. Во-вторых, реализация типа данных в алгебре отделена от спецификации типа данных в теории. В структуре класса представлено в основном описание сигнатуры типа данных. Как уже говорилось, исключением является реализация конструктора и методов класса. Реализация конструктора включена в структуру класса и позволяет построить объект (экземпляр класса). Методы класса предназначены для обработки самой структуры класса. Тем самым реализуется возможность использования понятия «метакласс», введенного в главе 1. Тип данных теперь становится объектом обработки специальными методами класса.

Обратим внимание на унифицированное представление в структуре класса полей, свойств и методов: ссылка на соответствующую таблицу, сама таблица, совокупность ссылок на реализацию и пока отсутствующая реализация. Структура класса позволяет теперь применить операцию присваивания значения реализации как к полям, так и к методам. Понимание принципов построения полей и методов позволяет глубоко освоить основополагающие механизмы объектно-ориентированного программирования: наследование, полиморфизм, инкапсуляция.

Очередное поле задано пустой ссылкой. В процессе создания объекта это поле будет содержать ссылку на метод обработки сообщений.

Последнее поле записи содержит ссылку на структуру, которая названа «Информация о типе времени исполнения» (Run-Type Type Information — RTTI). Информация о типе времени исполнения состоит из данных и процедур, которые создает компилятор для обработки классов и типов данных. Например, сведения о целом типе указывают минимальное и максимальное значения для конкретного целого типа. Эти таблицы доступны компонентам и приложениям во время выполнения. Система программирования использует эти структуры для поддержки интерфейса во время конструирования программы. Например, отслеживается информация о типе времени исполнения для поддержки определения форм и установки свойств форм и их компонент. Инспектор объектов ис-

пользует эту информацию очень интенсивно. Изучив информацию о типе времени исполнения, мы найдем для нее множество применений помимо нужд времени конструирования.

Рассмотрим теперь структуру объекта, которая представлена на рисунке 6.3. Объект создается с помощью конструктора, действия которого можно представить как копирование структуры класса и расширение полученной копии реализацией полей, свойств и методов. Результаты построения присваиваются объектной переменной.

Реализация полей представляет собой выделенную в соответствии с типом поля область динамической памяти. Отметим одну важную особенность: полями объекта могут быть как данные обычных типов данных (целое, строка, массив и т. п.), так и объекты. В случае объектных типов данных реализация поля представляет собой структуру объекта, аналогичную рисунку 6.3.

Реализация методов осуществляется традиционно для функций языка программирования: в виде исполняемого текста. Мы не будем затрагивать вопросы трансляции программ, поэтому для простоты будем считать, что реализация метода представлена его исходным текстом.

В структуре объекта появился деструктор. Как уже говорилось, с помощью конструктора объекты создаются в динамической памяти, а с помощью деструктора — уничтожаются. Более подробная информация об этих процессах приведена далее.

Функция обработки сообщений Windows предназначена для поддержания контакта с операционной системой, которая отправляет приложению послания (сообщения) в ответ на действия пользователя и другие события в вычислительной системе. Примерами действия пользователя могут быть нажатия на клавишу мыши или клавиатуры. Сообщение операционной системы принимает объект — приложение (класса TApplication), который ретранслирует полученное сообщение своим компонентам. У компонентов имеется богатый набор своих внутренних сообщений. Система сообщений компонентов получила название «динамический поток сообщений». Динамический поток сообщений — сложное понятие, однако его успешное освоение дает богатый набор профессиональных приемов событийного программирования.

Таким образом, выделение и сопоставление структуры класса и структуры объекта позволяет увидеть отличие в построении класса и объекта. Принципиальным является то, что информация о классе не содержит описание реализации, в то время как объект — это экземпляр класса с реализацией полей, свойств и методов. Объект создается с помощью конструктора путем копирования в свободную область динамической памяти структуры класса и расширения полученной структуры реализацией полей, свойств и методов. Для полей и свойств этот принцип абсолютен, а для методов имеется исключение: реализация конструктора и методов класса представлена в структуре класса.

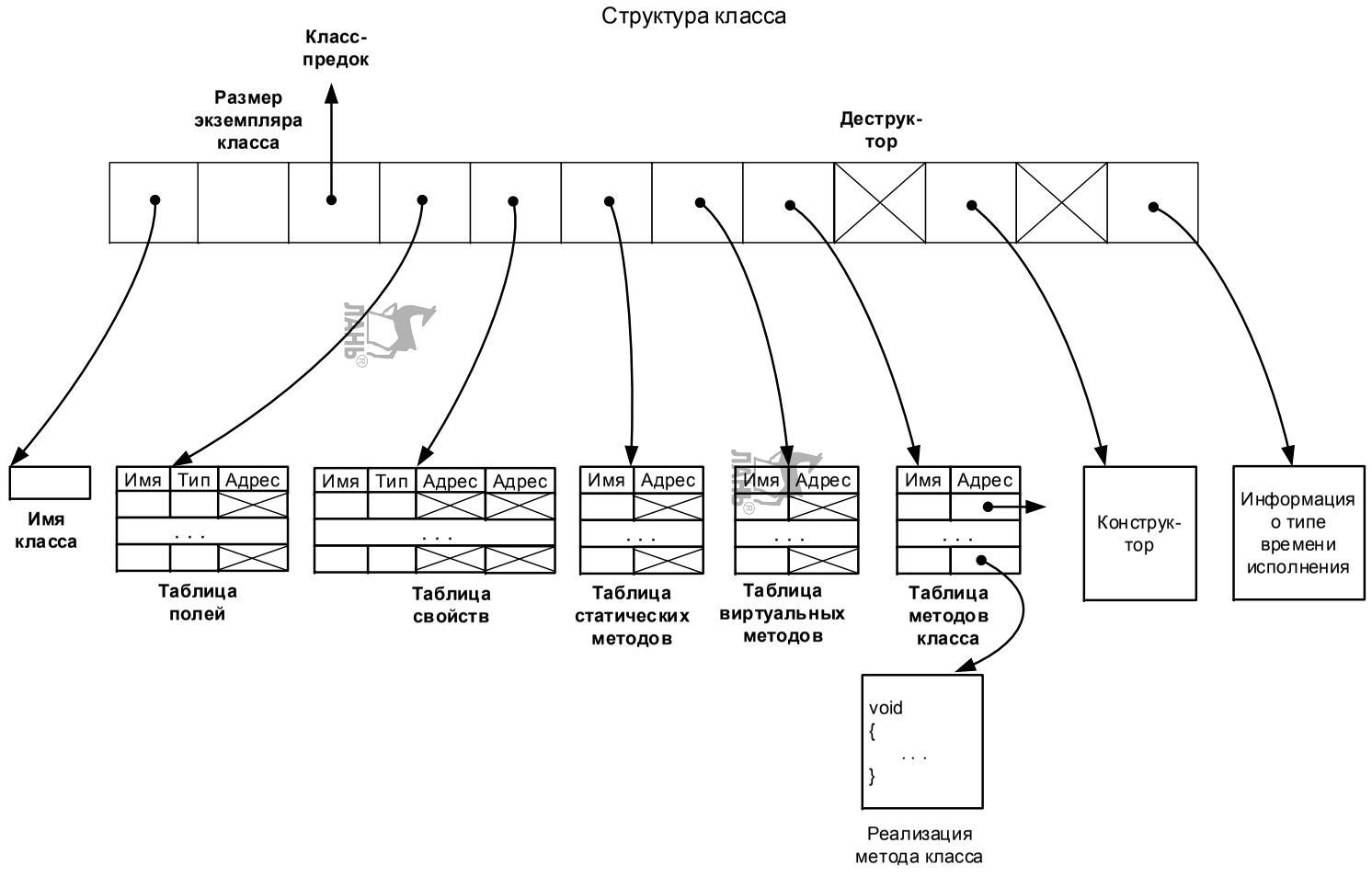

Структура класса
Класс-
предок
Размер
экземпляра
класса
Деструк-
тор
Имя
класса
Имя
Тип
Адрес
Таблица
полей
Имя
Тип
Адрес
Адрес
Таблица
свойств
Имя
Адрес
Таблица
статических
методов
Имя
Адрес
Таблица
виртуальных
методов
Имя
Адрес
Таблица
методов
класса
Конструк-
тор
Информация
о типе
времени
исполнения
void
{
}
Реализация
метода класса

Рис. 6.2

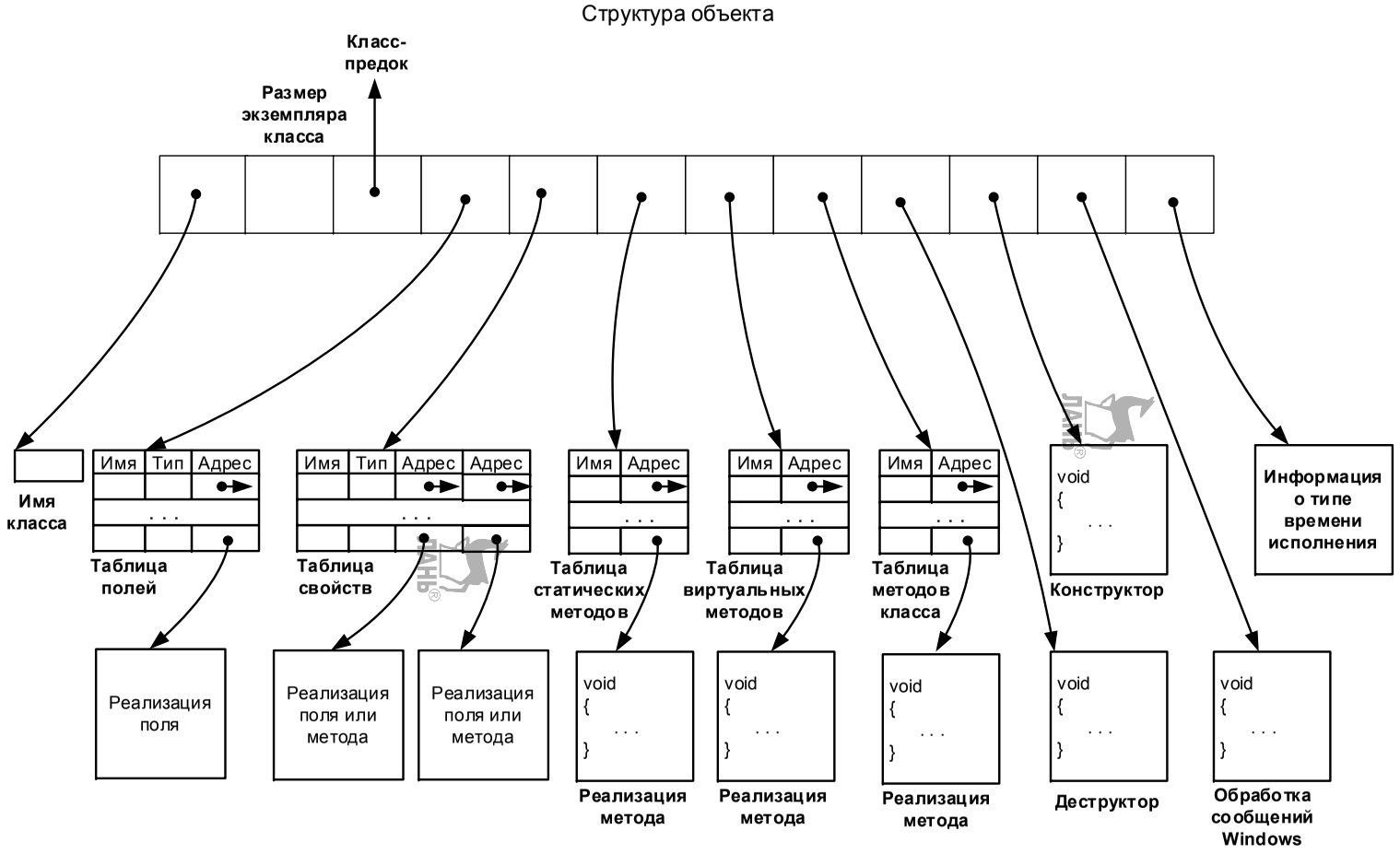

Структура объекта
Класс-предок
Размер экземпляра класса
Имя класса
Имя
Тип
Адрес
Таблица полей
Таблица свойств
Таблица статических методов
Таблица виртуальных методов
Таблица методов класса
Конструктор
Информация о типе времени исполнения
Реализация поля
Реализация поля или метода
Реализация поля или метода
void
{
. . .
}
Реализация метода
Реализация метода
Реализация метода
Деструктор
Обработка сообщений Windows

Рис. 6.3

Контрольные вопросы

1. В чем заключается принципиальное отличие структуры класса и структуры объекта?
2. Какие методы можно использовать в структуре класса?
3. Какие методы можно использовать в структуре объекта?

6.2. Класс TObject

Описание класса **TObject** приведено в пространстве имен **System** модуля ...**\Embarcadero\Studio\14.0\include\windows\systobj.h** стандартной поставки RAD Studio XE6[24].

Класс **TObject** является основообразующим классом, в нем сосредоточены все свойства «быть классом», «быть объектом». Он реализует и предоставляет потомкам функции, обязательно наследуемые любым объектом, который может быть создан в среде разработки. Учитывая гигантское разнообразие возможных областей применения объектов в процессе создания приложений, можно сказать, что круг общих для всех классов операций весьма невелик.

В первую очередь это создание экземпляра класса и его уничтожение. Любой программист выполняет эти две операции в обязательном порядке[25]. Процесс создания объекта включает выделение области адресного пространства, реализацию полей и методов, установку указателя на экземпляр объекта, задание начальных значений свойств и выполнение установочных действий, связанных с предназначением объекта. В общем случае две последние операции могут не выполняться. Например, пусть у нас имеется описание класса **TAnyClass**, тогда можно построить описание объектной переменной и создать объект:

```
TAnyClass * AnyObject;
AnyObject=new TAnyClass();
```

В первой строке примера выделяется память для хранения ссылки на структуру объекта. Структура класса создается ранее при обработке описания класса **TAnyClass**. Во второй строке примера происходит создание экземпляра класса **TAnyClass**. Указатель на экземпляр класса (объект) передается в переменную объектного типа **AnyObject**, которая в дальнейшем будет идентифицировать объект в программном коде приложения. При создании экземпляра класса конструктор **TAnyClass()** возвращает в объектную переменную указатель на выделенную для нового объекта область памяти, в которой располагается структура объекта. Для этого применяется метод **NewInstance()**, который вызывается в конструкторе автоматически. Описание метода **NewInstance()** в классе **TObject** имеет вид:

```
__classmethod virtual TObject* __fastcall NewInstance();
```

где служебное слово **__classmethod** свидетельствует о том, что данный метод является методом класса; служебное слово **virtual** объявляет метод виртуаль-

[24] В других версиях RAD Studio расположение модуля необходимо уточнить.

[25] Как увидим далее, функции создания и ликвидации объекта в потомках класса **TComponent** будут автоматизированы.

ным; описатель **TObject*** указывает на тип возвращаемого функцией значения (указатель на созданный конструктором объект); служебное слово **__fastcall** предписывает компилятору передачу параметров функции через быстрые регистры процессора (для регламентации некоторого стандарта использования регистров процессора и ускорения процесса вызова функции).

Класс **TObject** обеспечивает выполнение процесса создания объекта для любого порожденного от него класса. А уже внутри конструктора, который унаследован от класса **TObject**, можно предусмотреть инициализацию переменных и выполнение дополнительных операций. Объявление конструктора класса **TObject** выглядит следующим образом:

```
__fastcall TObject();
```

Как видно из описания, имя конструктора совпадает с именем класса. Объясняется это принятым в языке программирования C++ правилом: имя конструктора должно совпадать с именем класса.

Для ликвидации объекта класса **TObject** предназначены методы **~TObject** и **Free**:

```
virtual __fastcall ~TObject();
__fastcall Free();
```

Настоящим деструктором является метод **~TObject()**. Он обеспечивает освобождение всех занимаемых экземпляром класса ресурсов. Обычно при создании новых классов деструктор всегда переопределяется для того, чтобы корректно завершить работу со структурой объекта. Поэтому он объявлен с атрибутом **virtual** — переопределяемый.

Обратим внимание, что в потомках класса **TObject** унаследованный конструктор **TObject()** вызывается по умолчанию. Это необходимо для того, чтобы выполнить в соответствии с иерархией наследования все нужные операции при создании объекта до инициализации его собственных свойств. При уничтожении объекта обычно обратные созданию объекта действия осуществляются в обратном порядке: сначала выполняются завершающие операции в порядке от потомка к предку, и только в самом конце вызывается унаследованный от класса **TObject** деструктор, чтобы выполнить собственно уничтожение объекта и освободить занимаемую память.

При уничтожении объектов часто рекомендуется вместо деструктора использовать метод **Free()**, который просто вызывает деструктор, но перед этим проверяет, чтобы указатель на экземпляр объекта был не пустым (не был равен **NULL**). Считается, что это позволяет избежать серьезных ошибок, но, с другой стороны, использование метода **Free()** вместо метода **~TObject()** может скрыть существующую в программе ошибку.

Если объект является владельцем других объектов (например, форма владеет всеми размещенными на ней компонентами), то его метод **Free()** автоматически вызовет эти же методы для всех подчиненных объектов. Поэтому при закрытии формы разработчик избавлен от необходимости заботиться об уничтожении всех размещенных на форме компонентов.

Для освобождения занимаемой объектом памяти деструктор автоматически вызывает метод:

virtual void __fastcall FreeInstance();.

Каждый объект должен содержать некоторую информацию о себе, которая используется приложением и средой разработки. Поэтому класс **TObject** содержит ряд методов, обеспечивающих представление этой информации в потомках.

Метод

__classmethod void* __fastcall ClassInfo();

возвращает указатель на таблицу информации о типе времени исполнения (RTTI — Run Time Type Information). Эта информация используется в среде разработки и в приложении.

Функция

__classmethod String __fastcall ClassName();

возвращает имя типа объекта, которое может быть использовано для идентификации (рис. 6.3). Например, один метод-обработчик щелчка мыши на кнопке может работать с несколькими типами компонентов кнопок:

```
void __fastcall TMyForm::BtnClick(TObject *Sender)
{
    if (Sender –> ClassName() == «TBitBtn»)
       {
         ((TBitBtn*)Sender) –> Enabled=False;
       }
    else   if (Sender –> ClassName() == «TSpeedButton»)
              {
                 ((TSpeedButton*) Sender) –> Down=True;
              }
}.
```

В последнем примере параметр **Sender** содержит указатель на вызвавший метод **BtnClick** объект. Содержанием примера является выяснение с помощью операции **ClassName()** имени класса вызвавшего метод объекта с последующим исполнением для каждого типа индивидуальных действий. В примере можно увидеть также прием приведения типов.

Метод

__classmethod bool __fastcall ClassNameIs (const String string);

позволяет определить, является ли данный объект того типа, имя которого передано в параметре **string**. В случае положительного ответа функция возвращает **True**.

Как известно, программирование в операционной системе Windows основано на событиях. Каждое приложение и каждый программный объект должны уметь реагировать на сообщение о событиях и, в свою очередь, рассылать сообщения. В выполнении этих операций заключается общая для всех объектов функция. Метод

virtual void __fastcall Dispatch (void *Message);

осуществляет обработку сообщений, поступающих объекту. Он определяет, сможет ли объект обработать сообщение при помощи собственных обработчиков событий. В случае отсутствия таких методов сообщение передается аналогичному методу **Dispatch()** класса-предка (если он есть).

Класс **TObject** имеет предопределенный обработчик событий:

virtual void __fastcall DefaultHandler (void* Message);

Кроме рассмотренных здесь методов, класс **TObject** имеет еще несколько методов, которые в основном применяются для взаимодействия объекта со средой разработки.

В целом класс **TObject** может служить для создания на его основе некоторых простых классов для использования в приложениях.

Изучив тему «Наследование», мы убедимся, что класс **TObject** является фундаментом иерархии наследования. Это означает, что любые классы и объекты являются потомками класса **TObject** и наследуют все его свойства. Следовательно, механизм наследования позволяет любому объекту обладать свойством «быть объектом». Класс **TObject** является также родоначальником всей иерархии классов библиотеки визуальных компонентов.

В таблице 6.1 приведено содержание класса **TObject** в сокращенном виде. В средней колонке таблицы указаны заголовки методов, в левой колонке дается ссылка на фрагмент структуры класса (рис. 6.2) или структуры объекта (рис. 6.3), которые используют соответствующий метод; правая колонка содержит комментарий к рассматриваемому методу. Мы не вправе рассматривать реализацию методов, так как эта информация не документируется разработчиками, однако если взглянуть на структуру класса и структуру объекта, то реализация большинства методов становится очевидной.

Таблица 6.1

Содержание класса TObject

Название структуры на рис. 6.2, 6.3	Метод в классе TObject	Назначение метода
Имя класса	**__classmethod String __fastcall ClassName ();**	Возвращает строку с именем класса вызвавшего метод объекта или класса
Имя класса	**__classmethod bool __fastcall ClassNameIs (const String string);**	Проверяет, является ли класс вызвавшего метод объекта или класса равным значению параметра **string**
Размер экземпляра класса (объекта)	**__classmethod long __fastcall InstanceSize ();**	Возвращает размер экземпляра класса в байтах
Класс-предок	**__classmethod TClass __fastcall ClassParent ();**	Возвращает указатель специального вида **TClass** на аналогичную структуру класса-предка

Продолжение табл. 6.1

Название структуры на рис. 6.2, 6.3	Метод в классе TObject	Назначение метода
Потомок класса	**__classmethod bool __fastcall InheritsFrom (TClass aClass);**	Проверяет, является ли вызвавший метод класс или объект потомком класса **aClass**
Таблица полей	**void * __fastcall FieldAddress (const String Name);** **void * __fastcall FieldAddress (const ShortString &Name);**	Возвращает адрес реализации поля с именем **Name** вызвавшего данный метод объекта. Для ссылки на поля произвольного типа используется бестиповый указатель **void ***
Таблицы методов	**__classmethod void* __fastcall MethodAddress (const ShortString &Name);** **__classmethod void* __fastcall MethodAddress (const String Name);**	Возвращает адрес реализации метода с именем **Name** вызвавшего объекта или класса. Из класса допускается получать информацию только о методах класса
Таблицы методов	**__classmethod String __fastcall MethodName (void *Address);**	Возвращает имя метода с адресом реализации **Address** вызвавшего класса или объекта. Из класса допускается получать информацию только о методах класса
Конструктор	**__fastcall TObject ();**	Создает новый экземпляр класса (объект) и инициирует обработчик исключительных ситуаций
Деструктор	**virtual __fastcall ~TObject ();**	Производит действия по уничтожению объекта
Не показана	**__fastcall Free ();**	Эта процедура может использоваться вместо деструктора. Отличается от деструктора тем, что перед попыткой уничтожить объект проверяет передаваемый указатель на реальность существования объекта
Обработчик сообщений Windows	**virtual void __fastcall DefaultHandler (void* Message);**	Обработчик сообщений Windows по умолчанию
Информация о типе времени исполнения	**__classmethod __void* __fastcall ClassInfo ();**	Возвращает указатель на структуры данных с информацией о свойствах типа данных
Вся структура	**TClass __fastcall ClassType ();**	Возвращает указатель на структуру класса вызвавшего объекта

Как видно из таблицы 6.1, класс **TObject** содержит только методы. Здесь демонстрируется основополагающий принцип объектного программирования: допускать пользователя к свойствам объекта только путем использования методов.

Сформулируем главный вывод, который можно сделать, изучив содержание класса TObject.

Класс TObject *предназначен для обработки структуры класса, а также создания, обработки и ликвидации структуры объекта.*

Контрольные вопросы

1. Сформулируйте назначение класса TObject.
2. Сформулируйте правила именования конструктора и деструктора.
3. Каким образом в программах осуществляется наследование конструктора и деструктора?
4. Перечислите методы класса TObject и опишите их назначение.
5. Поясните назначение служебного слова __fastcall.

6.3. Жизнь классов и объектов

Рассмотрим примеры, с помощью которых будет продемонстрированы принципы использования в программе структуры класса и структуры объекта. Пусть имеется описание класса:

```
class TMyClass
        {
            Integer Field;
            void Method();
        };
```

ЛАНЬ®

В соответствии с этим описание в динамической памяти будет построена следующая структура класса:

Удобно считать, что базовая запись структуры класса расположена в статической памяти, а отсылаемые поля, свойства и методы — в динамической памяти. В этом случае имя класса является именем базовой записи и имеет статус константы. На последнем рисунке информация о классе представлена в краткой форме в виде прямоугольника, для восстановления подробностей необходимо обратиться к рисунку 6.2. Информация о классе будет содержать все необходимые таблицы. В нашем случае реализация поля **Field** и реализация метода **Method()** будут отсутствовать. Приемы обработки структуры класса будут рассмотрены в следующем параграфе.

Продолжим наш пример. Чтобы от описания класса перейти к объекту, следует сначала выполнить соответствующее объявление объектной переменной. Пусть имеется описание переменной:

```
TMyClass * MyObject;
```

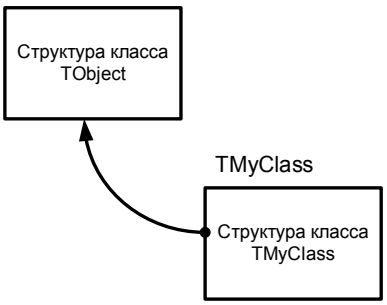
ЛАНЬ®

Обработка компилятором этого описания приведет к созданию переменной, имеющей тип адрес экземпляра класса **TMyClass**:

Переменная **MyObject** является именем ссылки, которой можно присвоить объект. Сам объект пока не существует, для его создания необходимо вызвать конструктор, однако в исходном описании класса **TMyClass** конструктор отсутствует. Эта проблема решается использованием принципа умолчания для выбора предка: если описание предка отсутствует, то им является класс **TObject**. Поэтому исходное описание класса **TMyClass** автоматически преобразуется в следующее:

```
class TMyClass : public TObject
        {
            Integer Field;
            void Method();
        };
```

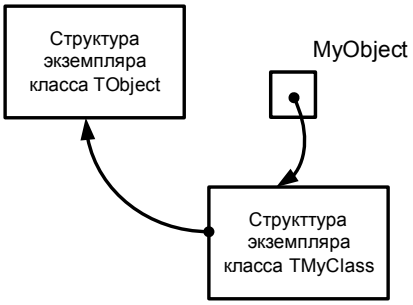
Как видим, в заголовке описания класса появилось выражение **public TObject**, что означает: класс **TMyClass** является потомком класса **TObject** и наследует все его содержание, в том числе и конструктор. В структуре класса **TMyClass** в поле «Класс-предок» появится ссылка на структуру класса **TObject**:



Появившаяся в поле «Класс-предок» ссылка (рис. 6.2) связала структуру класса **TMyClass** со структурой класса **TObject**. Наследование состоялось. Теперь вызов конструктора будет законной операцией:

```
MyObject=new TMyClass();
```

Через имя класса **TMyClass** получаем доступ к структуре класса **TMyClass**, затем через поле «Класс-предок» попадаем в структуру класса **TObject** и, наконец, совместными усилиями конструкторов создается объект **MyObject**:



Действия конструктора следующие: в динамическую память копируется структура класса **TMyClass**, добавляется реализация поля **Field** и метода **Method**, аналогичные действия выполняются над предками **TMyClass**, наконец, адрес полученной структуры объекта записывается в переменную **MyObject**. Удобно считать, что имя **MyObject** является средством идентификации объекта, а структура объекта вместе с присоединенными структурами (в нашем случае — структура объекта **TObject**) представляет сам объект.

После создания объект можно использовать в программе: получать и устанавливать значения его полей, вызывать его методы. С помощью этих действий устанавливается состояние объекта и обеспечивается поведение объекта. Доступ к полям и методам объекта происходит с помощью имени объекта, например,

```
MyObject–>Field=5;
MyObject–>Method();
```

Через имя объекта **MyObject** осуществляется доступ к структуре объекта с помощью операции –> (стрелка), затем с помощью таблицы полей (рис. 6.3) программа получает доступ к реализации поля, и, наконец, выполняется соответствующее действие: в поле **Field** записывается значение 5. Аналогично, через имя объекта **MyObject** и таблицы статических методов программа получает доступ к реализации метода и функция **Method** исполняется. Рекомендуем проследить выполнение рассмотренных в примере действий с помощью рисунка 6.3.

Если объект становится ненужным, то он должен быть удален вызовом деструктора, в нашем случае:

```
MyObject–>~TMyClass();
```

Через имя объекта программа получает доступ к структуре экземпляра класса **TMyClass**, затем через ссылку на предка — к структуре экземпляра класса **TObject**.

Совместными усилиями деструкторов объект будет ликвидирован в динамической памяти:

Вызов деструктора для несуществующих объектов недопустим и приведет к ошибке. Считается, что для избавления программистов от лишних ошибок в состав класса **TObject** введен метод **Free**, который следует вызывать вместо деструктора. Метод **Free** сам вызывает соответствующий деструктор, но только в том случае, если значение объектной переменной не равно **NULL**. Поэтому вместо **MyObject–>~TMyClass()** в приведенном выше примере можно использовать **MyObject–>Free();**.

Заметим, что использование метода **Free()** вместо деструктора снижает надежность программы, так как скрывает попытку разрушения несуществующего объекта, а если объекта нет, то и разрушать нечего.

После уничтожения объекта переменная **MyObject** сохраняет свое значение, продолжая ссылаться на место в динамической памяти, в которой объекта

уже нет. Если эту переменную предполагается еще использовать, то желательно присвоить ей значение **NULL**, чтобы в дальнейшем иметь возможность проверить, существует объект или нет. Таким образом, при предполагаемом повторном использовании объекта наиболее правильная последовательность действий при уничтожении объекта должна быть следующая:

```
MyObject–>~TMyClass();
MyObject=NULL;
```

Объекты могут выступать в программе не только в качестве переменных, но также элементов массивов, полей записей, параметров процедур и функций. Кроме того, они могут служить полями других объектов. Во всех этих случаях программист фактически оперирует указателями на экземпляры классов в динамической памяти. Следовательно, объекты изначально приспособлены для создания сложных динамических структур данных, таких как списки и деревья. Дополнительные указатели на объекты для этого не всегда нужны.

В некоторых случаях требуется, чтобы объекты разных классов содержали ссылки друг на друга. Возникает проблема: объявление первого класса будет содержать ссылку на еще не определенный класс. Она решается с помощью упреждающего объявления:

```
class TSecondClass;  // упреждающее объявление класса
class TFirstClass
      {
            TSecondClass * Second;
                        ...
        };

class TSecondClass
      {
             TFirstClass * Objects[10];
                        ...
        };
```

Первое объявление класса **TSecondClass** называется упреждающим (от *англ*. forward). Оно необходимо для того, чтобы компилятор нормально воспринял объявление поля **Second** в классе **TFirstClass**.

Подведем теперь итоги.

1. Класс, реализуемый с помощью структуры класса, существует как на этапе проектирования программы, так и во время исполнения программы.

2. Доступ к классу производится с помощью имени класса и «методов класса» класса **TObject**, при этом необходимо учитывать, что в структуре класса отсутствует реализация всех полей и реализация большинства методов.

3. Объект, реализуемый с помощью структуры объекта, появляется после вызова конструктора и существует до вызова деструктора.

4. Доступ к объекту осуществляется с помощью имени объекта с учетом правил установления идентичности объекта. В структуре объекта присутствует реализация всех полей и методов, поэтому не существует никаких ограничений доступа.

Контрольные вопросы

1. Объясните название этого параграфа.
2. Сформулируйте назначение конструктора.
3. Сформулируйте назначение деструктора.
4. Как осуществляется доступ к содержимому объекта?
5. Объясните назначение упреждающего объявления класса.

6.4. Доступ к структуре класса и структуре объекта

В современных системах программирования появилась новая возможность: программисту доступен тип данных для выполнения операций. Указанная возможность реализуется доступом к структуре класса. Однако заметим, что в соответствии с концепцией операционной системы Windows прямой доступ к структуре класса реализуется, опосредовано, путем использования методов класса **TObject**.

Доступ к структуре класса можно получить двумя способами:

1) через имя класса с помощью методов класса **TObject**. Например, пусть имеется описание класса:

```
class  TMyClass
{
      …
};,
```

тогда можно узнать размер экземпляра класса в байтах, вызвав предназначенный для этой цели метод класса

n =(int) TMyClass::InstanceSize();

где **n** — переменная целого типа. Как видно из примера, доступ к методу класса осуществляется операцией «**::**»;

2) с помощью специальной переменной — указателя на класс. В стандартной поставке системы программирования имеется описание:

typedef TMetaClass * TClass;,

где класс **TMetaClass** определен в одном с классом **TObject** пространстве имен и является некоторым промежуточным классом для контролируемого взаимодействия с классом **TObject**[26]. Когда мы рассматривали методы класса **TObject**, то тип **TClass** упоминался в качестве указателя на структуру класса **TObject**. Для программиста это, по сути, одно и то же.

Зная о существовании рассмотренного типа **TClass**, можно записать:

TClass P;

Возникает законный вопрос: какая потребность в таких указателях, если доступ к структуре класса можно получить с помощью имени класса? Для отве-

[26] Здесь проявляются возможности и назначение инкапсуляции: защитить программные компоненты от несанкционированного использования.

та на этот вопрос рассмотрим пример. Пусть у нас имеется экранная форма класса **TForm2**, на которой расположены различные компоненты. Наша цель заключается в том, чтобы узнать «родословную» как самой формы, так и расположенных на форме визуальных компонентов. Для получения указанной информации свяжем с событием **OnClick** каждого интересующего нас компонента обработчик события следующего вида:

```
void __fastcall TForm2::OnClick(TObject *Sender)
{
     TClass  ClassRef = Sender –> ClassType();
      ListBox1–>Clear();
      while (ClassRef  !=  NULL)
              {
              ListBox1 –> Items –> Add (ClassRef –> ClassName());
              ClassRef = ClassRef –> ClassParent();
              }
}
```

Например, щелкнув на поле ввода класса **TEdit** левой клавишей мыши, мы увидим в компоненте **ListBox1** следующую последовательность строк:

```
TEdit
TCustomEdit
TWinControl
TControl
TComponent
TPersistent
TObject
```

Рекомендуется проследить за использованием переменной **ClassRef**. Более подробную информацию о содержании выходной информации можно получить в главе «Библиотека визуальных компонентов».

Рассмотрим еще один пример. Пусть имеются описания классов:

```
class  TFirstClass
   {
     …
   };
class TSecondClass : public TFirstClass
   {
      …
   };.
```

Особенностью приведенных описаний классов является указание на то, что класс **TSecondClass** является потомком (наследником) класса **TFirstClass**. Введем теперь описания переменных с одновременной их инициализацией с помощью соответствующих конструкторов:

```
TFirstClass  * First=new TFirstClass();
TSecondClass * Second=new TSecondClass();
```

Наконец, рассмотрим два оператора присваивания:

```
First=Second;          // так присваивать можно
Second=First;          // так присваивать нельзя
```

Для прояснения ситуации рассмотрим укрупненные структуры объектов **First** и **Second** и результат первого присваивания (рис. 6.4).

Класс **TSecondClass** является потомком класса **TFirstClass**, поэтому он наследует все содержание предка (заштрихованная на рисунке область) и обладает собственным содержанием (незаштрихованная область). Во время исполнения правильного присваивания переменная **First** будет требовать только то, что относится к классу **TFirstClass** (заштрихованную на рисунке часть). Поэтому никаких неприятностей не ожидается. Во время исполнения неправильного присваивания переменная **Second** будет претендовать на свой законный объем памяти, который больше, чем размер **TFirstClass**, а это недопустимо.

Правило. *Переменные-указатели на класс могут ссылаться на любые классы-потомки*.

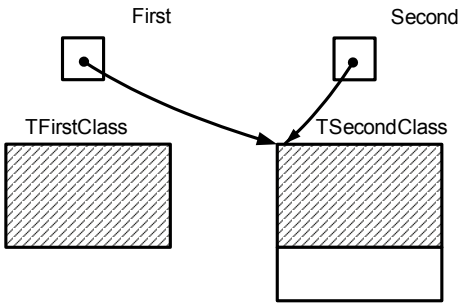


Рис. 6.4
Использование присваивания

Совместимость по присваиванию. Рассмотренный прием присваивания широко используется для передачи в метод адреса вызывающего этот метод объекта. Например, рассмотрим заголовок метода:

```
void __fastcall Form1::MyMethod (TObject * Sender);.
```

Переменной **Sender** можно сопоставлять любой объект, класс которого является потомком класса **TObject**, и это будет корректно. Если учесть, что любой класс является потомком класса **TObject**, то фактическим параметром рассмотренного метода **MyMethod** может быть экземпляр любого класса. Иногда внутри метода возникает необходимость узнать имя класса объекта, вызвавшего метод. Для пояснения рассмотрим пример. На рисунке 6.5 приведен фрагмент иерархии наследования стандартных классов RAD Studio. Отметим, что между классами не везде изображено отношение «непосредственный потомок», кроме того, о назначении каждого класса сейчас рассуждать не будем. Наша задача состоит в том, чтобы определить, какие действия можно осуществить с параметром **Sender**. Для этой цели введем метод **Inherited**, который отвечает на вопрос: является ли класс вызвавшего метод объекта потомком класса, указанного параметром?

Пусть имеется метод:

```
bool  __fastcall TForm2::Inherited (TObject * Sender, String Parent)
     {
       TClass  ClassRef = Sender–>ClassType ();
       while (ClassRef  !=  NULL)
             {
               if (ClassRef–>ClassNameIs (Parent)) return true;
               else ClassRef=ClassRef–>ClassParent ();
             }
        return false;
    }
```

В приведенном методе используются методы класса **TObject**. Первый, **ClassType()**, возвращает ссылку на структуру вызвавшего метод объекта. Второй метод, **ClassNameIs(Parent)**, возвращает истину, если вызвавший метод объект является экземпляром класса с именем **Parent**. Третий метод, **ClassParent()**, возвращает адрес непосредственного предка, вызвавшего этот метод объекта.

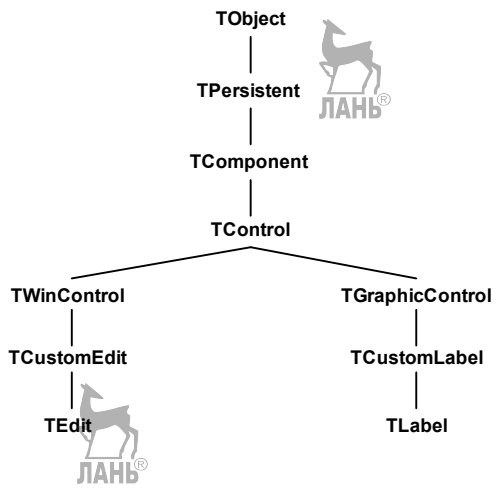


Рис. 6.5
Фрагмент иерархии наследования RAD Studio

Введем теперь метод, который сопоставим щелчку мыши над интересующими компонентами формы.

```
  void __fastcall TForm2::OnClick(TObject * Sender)
      {
      if (this–>Inherited(Sender, «TButton»)) ShowMessage(«ДА»);else ShowMes-
sage(«НЕТ»);
      }
```

Здесь метод **ShowMessage(«Текст сообщения»)** выводит на экран окно, в котором размещается **«Текст сообщения»**. Нетрудно сообразить, что только щелчок мыши на любой кнопке, являющейся потомком класса **TButton**, приведет к появлению сообщения **«ДА»**.

Рассмотрим примеры возможного использования операции **Inherited** в методе **MyMethod.** Пусть имеется вызов:

MyMethod (edtProjectName),

где **edtProjectName —** имя визуального компонента класса **TEdit,** предназначенного для ввода и редактирования названия проекта. Тогда, учитывая содержание рисунка 6.5:

Inherited (Sender, «TComponent») равно значению **True**,

Inherited (Sender, «TGraphicControl») равно значению **False**,

Inherited (Sender, «TObject») равносильно **True** при любом значении **Sender**.

Приведение типов. Приведение типов позволяет рассматривать экземпляр объекта как принадлежащий другому совместимому типу. Попытка приведения к несовместимому типу приводит к возникновению исключительной ситуации. После применения операции приведения объект остается неизменным, но вызываются те его методы, как если бы он принадлежал к присваиваемому классу. Продолжим наш пример с учетом иерархии (рис. 6.5).

Пусть имеется описание объектной переменной:

TObject * objMy;

тогда после инициализации объектной переменной **objMy** возможны присваивания:

(TLabel)(objMy)–>Caption=«Фамилия»;

если переменная **objMy** обозначает экземпляр класса, который является потомком класса **TLabel**, и

(TEdit) (objMy)–>Text=«Петров»;

если переменная **objMy** обозначает экземпляр класса, который является потомком класса **TEdit**.

Объединим теперь рассмотренные примеры в один:

```
void TMyClass::MyMethod(TObject * Sender)
{
 if (Inherited (Sender, «TLabel»)) (TLabel) (objMy)–>Caption=«Фамилия»;
 else if (Inherited (Sender, «TEdit»)) (TEdit) (objMy)–>Caption=« Петров «;
};.
```

В дальнейшем мы рассмотрим более элегантное решение подобной задачи с помощью приемов полиморфизма.

Методы класса. Отличительной особенностью методов класса является то, что реализация методов класса появляется в структуре класса. Синтаксическим признаком метода класса является служебное слово **__classmethod**, которое необходимо поставить перед описанием заголовка метода. Примерами методов класса является большинство методов класса **TObject**. Программист имеет возможность вводить свои методы класса, например,

__classmethod void __fastcall TMyClass::MyClassMethod(TObject * Sender);.

Методы класса можно вызывать без создания экземпляра класса (объекта). Рассмотрим пример:

```
class TMyClass
    {
        __classmethod Word __fastcall GetSize ();
    }
```

Рассмотрим теперь соответствующий последнему примеру фрагмент структуры класса **TMyClass**.

Пусть имеется описание переменной:

Word Size;,

тогда можно выполнить оператор:

Size=TMyClass::GetSize ();.

Через имя класса **TMyClass** получаем доступ к таблице методов класса, затем из строки таблицы методов класса, соответствующей методу **GetSize()**, получаем доступ к реализации метода. Метод **GetSize** выполняется. Обратим внимание на использование операции «::», в данном контексте эта операция обозначает переход от имени класса **TMyClass** к структуре класса, в которой автоматически осуществляется поиск таблицы методов класса и входящего в таблицу метода **GetSize()**.

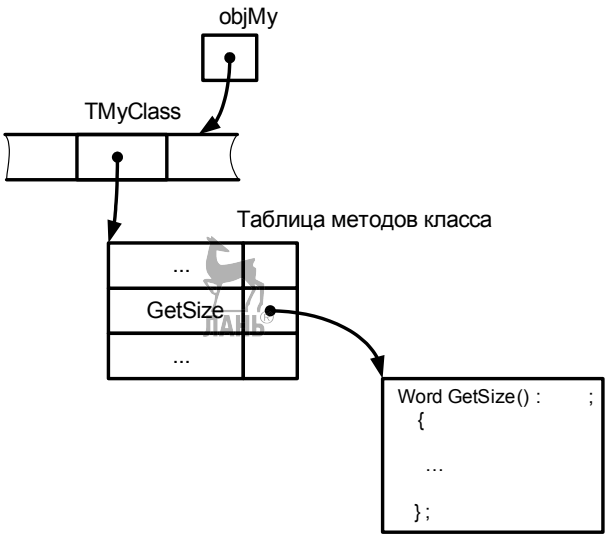


Рис. 6.6
Структура объекта **objMy**

Введем теперь переменные:

```
TMyClass * objMy;
Word Size1;
```

и выполним конструктор:

objMy=new TMyClass();

После исполнения конструктора получим структуру объекта, в которой интересующий нас фрагмент имеет вид, приведенный на рисунке 6.6.

Рассмотрим выполнение оператора:

Size1=objMy–>GetSize();

Через имя переменной **objMy** получаем доступ к экземпляру класса **TMyClass**, содержащему строку **GetSize** в таблице методов класса. Из таблицы методов класса вызывается реализация метода **GetSize**. Метод исполняется. Таким образом, любой метод класса может быть вызван как из структуры класса, так и из структуры объекта.

Рассмотрим еще один пример:

```
class TMyClass
     {
       …
     };
TClass AddrClass;
String   s;.
```

Выполним присваивания:

```
AddrClass=TMyClass::ClassParent();
s=AddrClass–>ClassName();
```

и рассмотрим представленный на рисунке 6.7 фрагмент структуры класса.

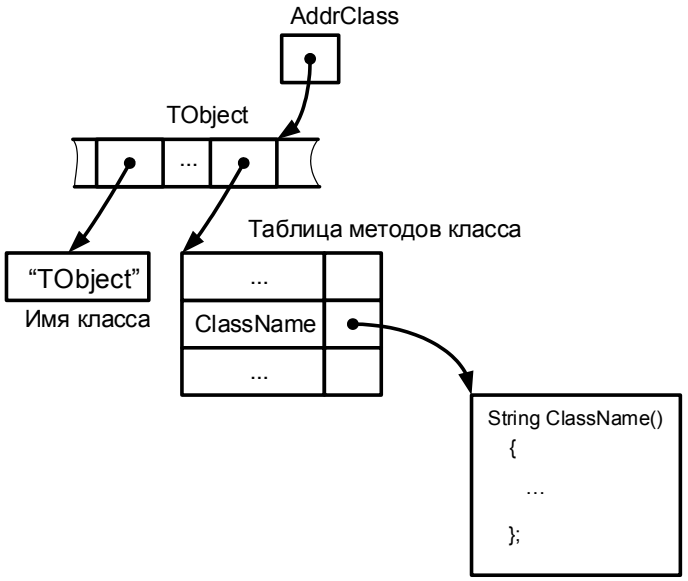


Рис. 6.7
Фрагмент структуры класса

Через ссылочную переменную **AddrClass** получаем доступ к структуре класса **TObject**, затем происходит переход к таблице методов класса и, наконец, переход к реализации метода **ClassName**. Метод исполняется, и переменная **s** получает значение **«TObject»**, выбранное из поля «Имя класса».

Контрольные вопросы

1. Какими двумя способами можно получить доступ к структуре класса?
2. Могут ли переменные-указатели на класс ссылаться на класс-предок?
3. Могут ли переменные-указатели на класс ссылаться на класс-потомок?
4. Поясните назначение распространенного параметра методов: **TObject *Sender**.
5. Назовите главную особенность метода класса.

6.5. Конструкторы и деструкторы

Конструктор — это особый метод, имеющий имя, совпадающее с именем класса. Назначением конструктора является создание и инициализация объектов. Количество и типы конструктора могут быть любые; конструктор может не иметь аргументов; аргументам разрешается присваивать значения по умолчанию; аргументы конструктора обычно используются для инициализации полей объекта. В описании конструктора не пишется тип возвращаемого результата — нельзя писать даже **void**.

Заголовок конструктора, реализуемого вне класса, как и для обычного метода, должен сопровождаться префиксом:

<класс>::<класс>(<параметры>).

Конструкторы, так же как и обычные методы, можно переопределять. Обычно различают три вида конструкторов:

— конструктор без параметров (конструктор по умолчанию);
— конструктор инициализации;
— конструктор копирования (с одним параметром определяемого класса).

При отсутствии в классе явно определенных конструкторов автоматически создаются конструктор без параметров (конструктор по умолчанию) и конструктор копирования. Создаваемый автоматически конструктор без параметров имеет вид:

<класс>::<класс>(){}.

Конструктор копирования автоматически создается в таком виде:

<класс>**::**<класс>**(const** <класс> **&**<ссылка>**) {*this=**<ссылка>**;}**.

Параметром конструктора копирования всегда является объект своего класса. Эти конструкторы обеспечивают объявление объектов без инициализации и с инициализацией другим объектом того же типа. В комментариях следующих примеров записано, какие конструкторы будут созданы автоматически:

```
TMyDate d;  // конструктор без параметров
TMyDate  t (d);     // конструктор копирования
TMyDate  r=d;       // конструктор копирования.
```

При наличии хотя бы одного явно определенного конструктора конструктор без аргументов не создается. Конструктор копирования создается всегда, если он не определен явно.

При явном определении конструктора копирования аргумент должен передаваться по ссылке — передавать его по значению нельзя. В классе может быть объявлено несколько конструкторов копирования, и разрешается определять несколько аргументов, если им присваиваются значения по умолчанию.

Любой конструктор с параметрами не своего класса является конструктором инициализации. Конструктор инициализации предназначен для задания начальных значений полям класса. Рассмотрим пример:

```
class TMyTime
{
   private:
      int intFHours, intFMinutes, intFSeconds;
   public:
      TMyTime();                          // конструктор без параметров
      TMyTime(int h, int m=0, int s=0)          // конструктор инициализации
        {
           intFHours=h; intFMinutes=m; intFSeconds=s;
        }
};.
```

При реализации конструктора вне класса можно записать:

```
TMyTime::TMyTime() { intFHours= intFMinutes=intFSeconds=0;}.
```

Конструктор инициализации обеспечивает объявление и инициализацию объектных переменных:

```
TMyTime t(10,30,20), s(13,45), u=14;.
```

Конструктор инициализации может вызываться явно:

```
TMyTime r=TTime(10);.
```

Конструктор инициализации со всеми аргументами по умолчанию играет роль конструктора без аргументов (конструктора по умолчанию).

Конструкторы вызываются и при создании динамических переменных:

```
TMyTime *a=new TMyTime;              // конструктор без параметров
TMyTime *b=new TMyTime();            // конструктор без параметров
TMyTime *c=new TMyTime(12,29);       // конструктор инициализации
TMyTime *d=new TMyTime(*c);   // конструктор копирования.
```

Конструкторы обеспечивают привычную форму объявления констант:

```
const TMyTime t1(23,34);      // конструктор инициализации
const TMYTime t2=10;          // конструктор инициализации
const TMyTime t3=TTime(10);.      // явный вызов.
```

Конструктор инициализации обеспечивает инициализацию массива:

```
TMyTime T[3]={1, 2, 3};.
```

Для каждого целого числа, представленного в списке, вызывается конструктор инициализации, поэтому инициализация массива представляет собой следующее:

TMyTime T[0]=TMyTime(1);
TMyTime T[1]=TMyTime(2);
TMyTime T[2]=TMyTime(3);.

Если массив не инициализируется, то для каждого элемента массива вызывается конструктор без параметров, если он определен явно или неявно (или вызывается конструктор инициализации с параметрами по умолчанию).

Деструктор — это метод, имеющий имя класса, с тем отличием, что его первым символом должен быть символ ~ (тильда). Деструктор не возвращает результата и не имеет аргументов. Деструктор в классе может быть только один. Деструктор вызывается автоматически при уничтожении объекта. При отсутствии в классе явно определенного деструктора он создается автоматически. Вид автоматически создаваемого деструктора следующий:

<класс>::~<класс>(){}.

Если конструкторы класса не запрашивают ресурсов, которые нужно возвращать (например, динамическую память), или не выполняют инициализирующих действий (например, если файл открывается, его нужно закрывать), то деструктор можно не описывать.

Контрольные вопросы

1. Каково назначение конструктора?
2. Какие три вида конструкторов различают в объектном программировании?
3. Какие конструкторы создаются автоматически?
4. Какие конструкторы необходимо описывать и вызывать явным образом?
5. Каково назначение деструктора?
6. Какие деструкторы создаются автоматически?
7. Какие деструкторы необходимо описывать и вызывать явным образом?

6.6. Поля

Полями класса называют переменные, объявленные внутри класса. Они предназначены для хранения значений данных во время функционирования экземпляра класса (объекта). Ограничений на тип полей в классе не предусмотрено. В описании класса поля должны (рекомендуется) предшествовать методам и свойствам. Обычно поля используются для внутренних потребностей операций внутри класса.

В соответствии с правилами именования идентификаторов (Приложение 2) при объявлении имен полей принято к названию добавлять характеризующий тип префикс и заглавную букву F. Соблюдая правила именования, мы из одного названия переменной можем извлечь информацию о типе, месте описания переменной, назначении переменной. Например, **uintFLineSize** — поле класса (**F**) имеет тип **unsigned int**, обозначает размер какой-то линии или какой-то строки; **lstFSections** — поле класса (**F**) имеет тип **TList**, обозначает список

каких-то секций. Словосочетание «какой-то» используется, чтобы подчеркнуть полное отсутствие в наших примерах сведений о проблемной области переменных, тем не менее тщательное именование переменных уже вносит немалую определенность в назначение переменных, что в итоге сыграет положительную роль в понимании незнакомой или забытой программы.

Напомним некоторые относящиеся к полям понятия главы 1.

1. Объект имеет свойства, характеризуется состоянием, обладает некоторым хорошо определенным поведением и уникальной идентичностью.

2. Свойством объекта называется характеристика, которая может принимать некоторые значения.

3. Объект характеризуется перечнем (обычно статическим) всех свойств данного объекта и состоянием — текущими (обычно динамическими) значениями каждого из этих свойств.

Рассмотрим пример. Пусть имеется определение класса:

```
class TMyClass
   {
      public:
         Byte   byteFSize;
              …
   };
```

Введем объектную переменную

TMyClass * objMy;,

создадим экземпляр класса

objMy= new TMyClass();

и выполним присваивание

objMy–> byteFSize=5;.

Для выяснения происходящих во время присваивания процессов рассмотрим представленный на рисунке 6.8 фрагмент структуры объекта **objMy**.

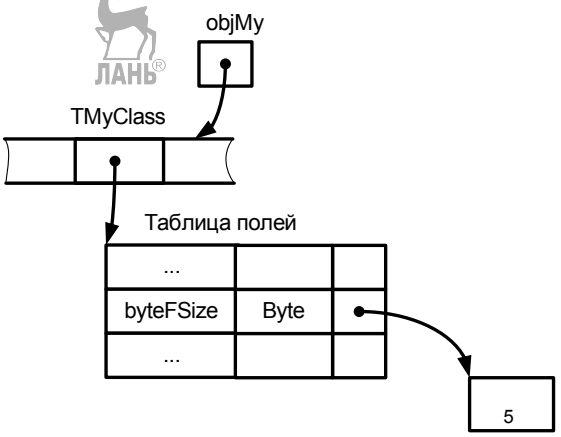


Рис. 6.8
Фрагмент структуры объекта **objMy**

Через переменную **objMy** выполняется идентификация объекта, затем по ссылке получаем доступ к таблице полей. Далее выполняется поиск в таблице полей строки с именем **byteFSize** и из найденной строки — переход на реализацию поля. Наконец, запись числа 5 в ячейку типа Byte выполняется традиционно. Заметим, что поиск реализации поля выполняется еще на этапе компиляции программы.

Контрольные вопросы

1. Опишите понятие «поле».
2. Чем определяется перечень свойств объекта?
3. Как можно изменить перечень свойств объекта?
4. Чем определяется состояние объекта?
5. Как можно изменить состояние объекта?
6. Опишите содержание операции «–>» (стрелка).

6.7. Методы

Метод — это функция, которая входит в состав объекта и может быть вызвана.

В зависимости от ряда факторов (доступ к методу, связывание адресов) методы делятся на статические и виртуальные. Синтаксическим признаком отличия методов является записываемое в определении метода служебное слово **virtual**, например,

void A(); — статический метод,
void virtual B(); — виртуальный метод.

Как видно из приведенного примера, при определении методов используется принцип умолчания: при отсутствии в описании метода признака разновидности метода нужно считать его статическим.

От обычных функций методы отличаются тем, что им при вызове передается указатель на тот объект, который их вызвал. Поэтому обрабатываться будут данные именно того объекта, который вызвал метод.

Рассмотрим пример. Пусть имеется определение класса:

```
class TMyClass
   {
      public:
         Byte   byteFField;
          void Method();
   };,
```

где метод Method реализован следующим образом:

```
void TMyClass::MyMethod()
   {
      ShowMessage(IntToStr(byteFField));
   }
```

Введем объектную переменную

TMyClass * objMy;,

создадим экземпляр класса

objMy= new TMyClass();.

Выполним действия:

objMy–>byteFField=5;
objMy–>MyMethod();

Для выяснения происходящих во время выполнения действий процессов рассмотрим фрагмент структуры объекта **objMy**, представленный на рисунке 6.9.

Выполнение присваивания **objMy–>byteFField=5;** осуществляется следующим образом. Через переменную **objMy** выполняется идентификация объекта, затем по ссылке получаем доступ к таблице полей. Далее выполняется поиск в таблице полей строки с именем **byteFField** и из найденной строки — переход на реализацию поля. Наконец, число 5 записывается в ячейку памяти типа Byte. Поиск реализации поля выполняется на этапе компиляции программы. Во время исполнения программы происходит прямое обращение по вычисленным на этапе компиляции адресам.

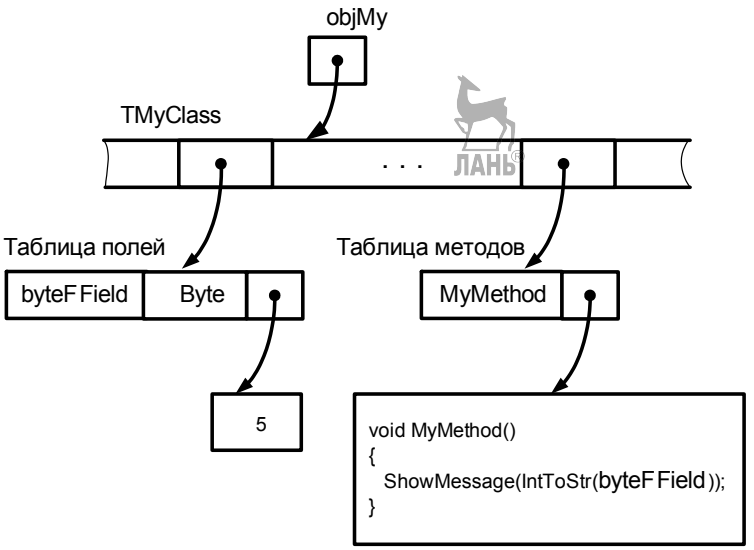


Рис. 6.9
Фрагмент структуры объекта **objMy**

Рассмотрим теперь выполнение вызова метода **objMy–>MyMethod();**. Через переменную **objMy** выполняется идентификация объекта, затем по ссылке получаем доступ к таблице статических методов. Далее выполняется поиск в таблице методов строки с именем **MyMethod** и из найденной строки таблицы статических методов — переход на реализацию метода. Метод начинает исполняться и на экран выводится форма, в которой находится текст «5». Заметим,

что у статических методов поиск реализации метода выполняется на этапе компиляции программы, поэтому в исполняемой программе связь вызова метода с реализацией метода установлена напрямую.

На некоторых важных особенностях использования методов мы остановимся далее в главах «Наследование» и «Полиморфизм».

Контрольные вопросы

1. Опишите понятие «метод».
2. Чем определяется поведение объекта?
3. Как можно изменить поведение объекта?
4. Как строится описание метода?
5. Как осуществляется вызов метода?
6. Опишите содержание операции «–>» (стрелка).

6.8. Свойства

Свойством называется особый вид полей, придуманных специально для объектно-ориентированного программирования, в котором действует правило: не разрешать пользователю произвольно изменять состояние объекта, а делать это посредством и под контролем методов.

Синтаксическим признаком объявления свойства в классе является служебное слово **__property**. Обычно свойство определяется тремя элементами: идентификатором свойства и двумя опциями (**read** и **write**), которые осуществляют чтение/запись значения свойства. Рассмотрим пример:

```
class  TColorClass
   {
       private:
    TColor colFColor;
    TColor  GetColor();
              void SetColor(TColor New);
       public:
              __property TColor colPColor={read=GetColor, write=SetColor};
  };
```

Если в разделах **read** и **write** объявления свойства записано имя поля, то предполагается прямое чтение или запись данных в указанное поле. Если в разделе **read** указано имя метода (как в нашем примере), то чтение значения свойства будет осуществляться функцией с этим именем. Метод чтения — это функция без параметров, возвращающая значение того типа, который объявлен для свойства. Имя функции принято начинать с префикса **Get**, после которого следует имя свойства. Если в разделе **write** записано имя метода, то запись значения в поле будет осуществляться указанной функцией. Метод записи — это функция с одним параметром того типа, который объявлен для свойства. Имя метода записи принято начинать с префикса **Set**, после которого следует имя свойства. Если раздел **write** отсутствует в объявлении свойства, значит это свойство только для чтения и пользователь класса-владельца свойства не может задавать его значение.

Реализация методов в нашем случае может выглядеть следующим образом:

```
TColor TColorClass::GetColor()
      {
         return colFColor;
      }
void TColorClass::SetColor(TColor New)
      {
         colFColor=New;
      }.
```

В данном примере значение цвета хранится в недоступном для пользователя класса поле **colFColor**, а свойство **colPColor** предоставляет возможность использования его значения для нужд создаваемой программы.

Доступ к значению свойства **colPColor** осуществляется через вызовы методов **GetColor** и **SetColor**. Однако программист использует свойство как обычную переменную:

```
ColorObject–>colPColor=clRed;
Variable=ColorObject–>colPColor;
```

Компилятор самостоятельно оттранслирует обращение к свойству **colPColor** в вызовы методов **GetColor** или **SetColor**. Таким образом, внешне свойство выглядит в точности как обычное поле, но за всяким обращением к нему могут стоять нужные нам действия.

Частичная структура объекта **ColorObject** приведена на рисунке 6.10.

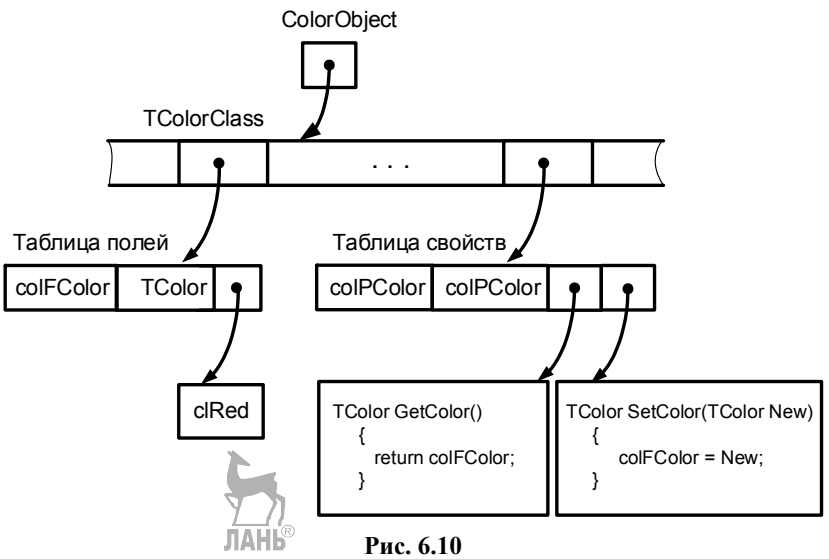


Рис. 6.10
Структура объекта **ColorObject**

Рассмотрим присваивание **ColorObject–>colPColor=clRed;**. Через имя объекта **ColorObject** получаем доступ к свойству **colPColor**. Так как свойство **colPColor** находится в левой части оператора присваивания, с ним связан ме-

тод **SetColor**, в который передается фактическое значение **clRed** параметра **New**. В результате вызова метода **SetColor** в поле **colFColor** записывается значение **clRed**.

Переходим к присваиванию **Variable=ColorObject–>colPColor;**. Через имя объекта **ColorObject** получаем доступ к свойству **colPColor**. В этом случае со свойством связывается метод **GetColor**, который возвращает в результате вызова значение **clRed** поля **colFColor**. Наконец, переменная **Variable** принимает значение **clRed**.

Заканчивая рассмотрение примера, еще раз обратим внимание на то, что поле **colFColor** явно не участвует в рассмотренных операторах присваивания. Вся работа с ним осуществляется в методах **SetColor**, **GetColor**.

В методах, входящих в состав свойств, может осуществляться проверка устанавливаемой величины на попадание в допустимый диапазон значений и вызов других процедур, зависящих от вносимых изменений. Если же потребности в специальных процедурах чтения и/или записи нет, можно вместо имен методов применять имена полей. Рассмотрим следующую конструкцию:

```
class TMyClass
   {
     private:
        TSomeType FValue;
        void Processing();
        Boolean Correct(TSomeType AValue);
        void SetValue(TSomeType NewValue);
     public:
        __property TSomeType PValue={read=FValue, write SetValue};
  };
                ...
void TMyClass::SetValue(TSomeType NewValue);
  {
     if (Correct(NewValue) FValue=NewValue;
     Processing();
  };
```

В этом примере чтение значения свойства **PValue** означает просто чтение поля **FValue**. Зато при присвоении значения внутри **SetValue** вызывается сразу два метода: вначале проверяется корректность нового значения свойства, затем выполняется обработка.

Свойство может быть и векторным; в этом случае оно внешне выглядит как массив:

```
__property TPoint pntPPoints [Integer Index]={read=GetPoint,
                                write=SetPoint};.
```

На самом деле в классе может и не быть соответствующего поля — массива. Напомним, что вся обработка обращений к внутренним структурам класса может быть замаскирована.

Для векторного свойства необходимо описать не только тип элементов массива, но также имя и тип индекса. После ключевых слов **read** и **write** в этом

случае должны стоять имена методов — использование здесь полей массивов недопустимо. Метод, читающий значение векторного свойства, должен быть описан как функция, возвращающая значение того же типа, что и элементы свойства, и имеющая единственный параметр того же типа и с тем же именем, что и индекс свойства:

TPoint GetPoint(Integer Index);.

Аналогично, метод, помещающий значения в такое свойство, должен первым параметром иметь индекс, а вторым — переменную нужного типа (которая может быть передана как по ссылке, так и по значению):

void SetPoint(Integer Index, TPoint NewPoint);.

В заключение приведем пример, в котором используется векторное свойство, но отсутствует соответствующее поле-массив для хранения значений.

ЛАНЬ®

```
class TVecClass
   {
      private:
         int GetValue(Integer Index);
      public:
         __property int intPSeriesOfNembers[Integer Index]={read=FValue};
   };
                  ...
void TVecClass::GetValue(Integer Index);
   {
      if (1 <= Index && Index <= 100) return Index * Index;
      else return 0;
   };
```

В данном примере с помощью свойства мы получаем в диапазоне от 1 до 100 ряд чисел, каждое из которых является квадратом порядкового номера числа в ряду. Каждое значение элемента векторного свойства вычисляется при получении этого значения. Вычисление осуществляется в методе **GetValue**. Записывать куда-то значения элементов вектора нет необходимости, поэтому в описании свойства отсутствует опция **write**.

О роли свойств в объектном программировании красноречиво говорит следующий факт: у всех имеющихся в распоряжении программиста (RAD Studio) стандартных классов 100% полей недоступны и заменены базирующимися на них свойствами. Рекомендуем при разработке собственных классов придерживаться этого же правила.

ЛАНЬ®

Контрольные вопросы

1. Опишите понятие «свойство».
2. Как строится описание свойства?
3. Чем использование свойства отличается от использования поля?
4. Как строится описание векторного свойства?

6.9. Программирование классов и объектов

6.9.1. Определение класса

Класс определяется с помощью конструкции

<тип класса><имя класса> {<компоненты класса>};.

Точка с запятой в конце описания ставится обязательно. В этом определении:

<тип класса> — одно из служебных слов **class**, **struct**, **union**[27];

<имя класса> — идентификатор класса;

<компоненты класса> — определения свойств, операций, событий, принадлежащих классу.

Имя класса является по умолчанию именем типа объектов. Данные — это поля объекта, определяющие его структуру. Совокупность значений полей определяет состояние объекта. С помощью методов класса определяются возможные операции над объектами. События — это явления, происшествия, возникающие внутри или вне приложения. Приложение должно иметь возможность реагирования на происходящие события.

Разрешается определять класс с любым содержанием: только с полями, с полями и методами, с методами и событиями, пустым.

Доступ к содержимому класса можно задать спецификатором доступа: **public**, **private** и др. Ключевое слово **public** объявляет элемент класса доступным вне класса, а ключевое слово **private** закрывает доступ снаружи. После ключевого слова требуется поставить знак «**:**».

Элементы класса типа **class** по умолчанию закрыты и недоступны вне данного класса. Элементы класса типа **struct** по умолчанию открыты и доступны в других частях программы. В остальном свойства классов типа **class** и **struct** одинаковы. Выбор типа класса из перечисленных — это дело пристрастий программиста.

И в классе, и в структуре можно написать столько спецификаторов **public** и **private**, сколько необходимо, и в том порядке, в каком требуется. Очередной спецификатор действует до следующего. Допускается индивидуально объявлять каждый элемент класса либо открытым, либо закрытым. Открытая часть класса называется интерфейсом. Сокрытие информации о внутренней структуре — это один из инструментов инкапсуляции. Как уже говорилось ранее, инкапсуляция является основополагающим принципом объектно-ориентированного программирования (к этой теме мы вернемся в главе 8).

Рассмотрим ряд примеров определения класса. Напомним, что имена классов в C++Builder рекомендуется начинать с символа «T»[28].

```
class TEmptyType{};    // пустые классы
struct TEmptyType{};
struct TPair {int intFFirst; double dubFSecond};  // класс содержит только
открытые поля
```

[27] Использование **union** рекомендуем изучить самостоятельно.

[28] Правила именования приведены в Приложении 2.

```
struct TDate                                    // класс содержит только открытые
поля и методы
{
   int intFMonth, intFDay, intFYear;
   void SetDate(int d, int m, int y);
   void GetDate(int &d, int &m, int &y);
    void NextDate();
    void ShowDate();
}
struct TMoney                    // класс содержит закрытое поле и открытые методы
{
     private: long longFRubls;
     public:
         void Add(const long &r);
         void Display();
}
class TFraction                  // класс содержит закрытые поля и открытый метод
{
   int intFNum; int intFDenum;
   public:
         void Reduce();
}
class TLibrary                   // класс содержит только открытые методы
{
    public : void CopyFile(String strAFrom, String strATo);
    public: void ShowFile(String strAFile);
}
```

6.9.2. Использование класса

Имея определение класса, можно объявить переменные типа «имя класса», т. е. объекты. Класс объявляется один раз, а вот объектов можно создать столько, сколько необходимо. Объекты объявляются аналогично переменным:

```
<имя класса> <имя объекта>;                    // скалярный объект
<имя класса> * <имя объекта>;                  // указатель
<имя класса> <имя объекта>[<количество>];      // массив
```

Разрешается совмещать определение класса и определение объекта:

```
class <имя класса> {<тело класса>} <имя объекта>;                  // скалярный
объект
class <имя класса> {<тело класса>}* <имя объекта>;                 // указатель
struct <имя класса> {<тело класса>}<имя объекта>[<количество>];    // массив
```

Эти переменные получат тип **<имя класса>**. В дальнейшем можно объявлять переменные без записи определения класса, используя только **<имя класса>**. Объявленные переменные подчиняются правилам видимости и время их жизни зависит от места объявления.

Следующие переменные-объекты могут быть объявлены для приведенных в предыдущем параграфе классов:

```
TEmptyType objEmpty;                    // переменная
TPair objPair[10];                  // массив
TDate * objDate;                    // указатель
class TMoney objMoney[5];               // массив
class TFraction * objFraction;          // указатель
TLibrary objLibrarys[10], * objLibrary;         // массив и указатель
```

Переменную можно инициализировать другой переменной того же класса. Разрешается объявлять и ссылки на объекты с обязательной инициализацией. Например:

```
class TAny{public:int i;} *p1=new m();          // инициализация конструктором
TAny p2=m();                            // инициализация переменной
TAny p3(p2);                            // инициализация переменной
```

Если поля класса открыты, разрешается обычная инициализация полей инициализатором структуры, например:

```
class TPerson {public: String strFFio; double dubFSalary;};
TPerson objPerson={“Иванов И.”,15000};.
```

Указатели могут быть инициализированы так же, как и указатели на переменные стандартных типов:

```
TPerson * ps=new TPerson(), *pp=ps, *pq;
ps=new TPerson();
```

Экземпляры класса разрешается определять в качестве полей другого класса:

```
class TAccount                          // счет
{
    TMoney  * objFSumma;                // сумма на счете
    unsigned  long ulongFAccountNumber; // номер счета
    TDate * objFAccountDate;            // дата открытия счета
  public:
    void Display();                     // вывод на экран
};
```

Экземпляры класса можно передавать в качестве аргументов любыми способами (по значению, по ссылке, по указателю) и получать как результат из функции:

```
TMoney AddMoney(const TMoney &a, const TMoney &b); // по константной ссылке
TMoney Init(const long double &r);                  // возврат значения
void F1(TDate d);                           // по значению
void F2(TDate &d);                              // по ссылке
void F1(TDate *d);                          // по указателю
```

Функция, возвращающая объект, может быть использована для инициализации объектной переменной:

```
TMoney m1=Init(100.21);
TMoney m2=AddMoney(x, y);
```

Можно объявлять объекты-константы с обязательной инициализацией:

```
const TMoney k=p;
const TDate=d(t);
const TMoney t=Init(234.5);
```

Доступ к открытым полям выполняется обычными для структуры способами:

<объект> . <поле>

<указатель> –> <поле>

Вызов открытых методов осуществляется аналогично

<объект> . <метод> (<параметры>)

<указатель> –> <метод>(<параметры>).

Для массива объектов обращение к открытому полю выполняется так:

<имя массива> [<индекс>] . <поле>.

Аналогично выполняется и вызов метода для элемента массива

<имя массива> [<индекс>] . <метод>(<аргументы>).

Примеры.

```
TDate d;
d.day=25; d.month=10; d.year=2016;
d.print();
d.set(15,6,1985);
TDate DD[3]; DD[0].day=14;
DD[0].next();
TLibrary t;
int a=t.function(7);
TLibrary *pt=new TLibrary();
int b=pt –> function(7);
TCount A[5];
for (int i=0; i < 5; i++)  A[i].Display();
```

ЛАНЬ®

Контрольные вопросы

1. Какой синтаксис имеет определение класса?
2. Какое содержание может иметь класс?
3. Какие имеются разновидности объектов?
4. Какие приемы используются для инициализации объектов?

Задачи

В заданиях этой темы необходимо создать новый проект в режиме «VCL Forms Application-C++Builder», на экранную форму поместить кнопку «TButton» из раздела «Standard» панели инструментов «Tool Palette». При выделенной кнопке выбрать в инспекторе объектов «Object Inspector» закладку «Events» и создать обработчик «OnClick»:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{

}.
```

Все дальнейшие построения реализовать внутри обработчика события нажатия на кнопку.

В каждом упражнении требуется реализовать в том или ином виде определение нового класса. Во всех заданиях необходимо реализовать конструктор инициализации (один или несколько) и конструктор без аргументов. Необходимо также предусмотреть деструктор.

Продемонстрировать использование методов класса TObject*:* ClassName, ClassNameIs, InstanceSize, ClassParent, InheritsFrom, FieldAddress, MethodAddress, MethodName.

Все доступные вне объекта поля реализовать с помощью свойств.

Для решения приведенных задач рекомендуется использовать три приема решения:

а) нарисовать структуры объектов и проследить процессы установления связи между вызовами методов и реализацией методов;

б) мысленно представить структуры объектов и установить связь вызова каждого метода с его реализацией;

в) набрать в системе программирования RAD Studio приведенные программы и проследить за их выполнением.

Массивы и константы в классе

Максимально возможный размер массива задать константой. В отдельном поле size должно храниться максимальное для данного объекта количество элементов массива; реализовать метод size(), возвращающий установленную длину. Если количество элементов массива изменяется во время работы, определить в классе поле count. Первоначальные значения size и count устанавливаются конструктором.

Список заданий.

1. Создать класс TBitString для работы с битовыми строками не более чем из 100 бит. Битовая строка должна быть представлена массивом типа unsigned char, каждый элемент которого принимает значение 0 или 1. Реальный размер массива задается как аргумент конструктора инициализации. Должны быть реализованы все традиционные операции для работы с битовыми строками: and, or, xor, not. Реализовать сдвиг влево и сдвиг вправо на заданное количество битов.

2. Создать класс TDecimal для работы с беззнаковыми целыми десятичными числами, используя для представления числа массив из 100 элементов типа unsigned char, каждый из которых является десятичной цифрой. Младшая цифра имеет меньший индекс (единицы — в нулевом элементе массива). Реальный размер массива задается как аргумент конструктора инициализации. Реализовать арифметические операции, аналогичные встроенным для целых в C++, и операции сравнения.

3. Создать класс THex для работы с беззнаковыми целыми шестнадцатеричными числами, используя для представления числа массив из 100 элементов типа unsigned char, каждый из которых является шестнадцатеричной цифрой. Младшая цифра имеет меньший индекс. Реальный размер массива задается как

аргумент конструктора инициализации. Реализовать арифметические операции, аналогичные встроенным для целых в C++, и операции сравнения.

4. Создать класс TMoney для работы с денежными суммами (задание III.4 из предыдущего параграфа). Сумма должна быть представлена массивом, каждый элемент которого — десятичная цифра. Максимальная длина массива — 100 цифр, реальная длина задается конструктором. Младший индекс соответствует младшей цифре денежной суммы. Младшие две цифры — копейки.

5. Создать класс TPolinom для работы с многочленами до 100-й степени. Коэффициенты должны быть представлены массивом из 100 элементов-коэффициентов. Младшая степень имеет меньший индекс (нулевая степень — нулевой индекс). Размер массива задается как аргумент конструктора инициализации. Реализовать арифметические операции и операции сравнения, вычисление значения полинома для заданного значения *x,* дифференцирование, интегрирование.

6. Создать класс TFraction для работы с беззнаковыми дробными десятичными числами. Число должно быть представлено двумя массивами типа unsigned char: целая и дробная часть, каждый элемент — десятичная цифра. Для целой части младшая цифра имеет меньший индекс, для дробной части старшая цифра имеет меньший индекс (десятые — в нулевом элементе, сотые — в первом и т. д.). Реальный размер массивов задается как аргумент конструктора инициализации. Реализовать арифметические операции сложения, вычитания и умножения, и операции сравнения.

7. Реализовать класс TRational (задание III.8 из предыдущего параграфа), используя два массива из 100 элементов типа unsigned char для представления числителя и знаменателя. Каждый элемент является десятичной цифрой. Младшая цифра имеет меньший индекс (единицы — в нулевом элементе массива). Реальный размер массива задается как аргумент конструктора инициализации.

8. Реализовать класс TMoney (задание III.13 из предыдущего параграфа), используя для представления суммы денег массив структур. Структура имеет два поля: номинал купюры и количество купюр данного достоинства. Номиналы представить как перечислимый тип nominal. Элемент массива структур с меньшим индексом содержит меньший номинал.

9. Реализовать класс TData (задание III.9 из предыдущего параграфа), используя для представления месяцев массив структур. Структура имеет два поля: название месяца (строка), количество дней в месяце. Индексом в массиве месяцев является перечислимый тип month. Реализовать два варианта класса: с обычным массивом и статическим массивом месяцев.

10. Карточка иностранного слова представляет собой структуру, содержащую иностранное слово и его перевод. Для моделирования электронного словаря иностранных слов реализовать класс TDictionary. Данный класс имеет поле-название словаря и содержит массив структур TWordCard, представляющих собой карточки иностранного слова. Название словаря задается при создании нового словаря, но должна быть предоставлена возможность его изменения во время работы. Карточки добавляются в словарь и удаляются из него. Реализовать поиск определенного слова как отдельный метод. Аргументом операции

индексирования должно быть иностранное слово. В словаре не должно быть карточек-дублей. Реализовать операции объединения, пересечения и вычитания словарей. При реализации должен создаваться новый словарь, а исходные словари не должны изменяться. При объединении новый словарь должен содержать без повторений все слова, содержащиеся в обоих словарях-операндах. При пересечении новый словарь должен состоять только из тех слов, которые имеются в обоих словарях-операндах. При вычитании новый словарь должен содержать слова первого словаря-операнда, отсутствующие во втором.

11. Один тестовый вопрос представляет собой структуру TTask со следующими полями: вопрос, пять вариантов ответа, номер правильного ответа, начисляемые баллы за правильный ответ. Для моделирования набора тестовых вопросов реализовать класс TTestContent, содержащий массив тестовых вопросов. Реализовать методы добавления и удаления тестовых вопросов, а также метод доступа к тестовому заданию по его порядковому номеру в списке. В массиве не должно быть повторяющихся вопросов. Реализовать операцию слияния двух тестовых наборов, операцию пересечения и вычисления разности (см. задание 10). Дополнительно реализовать операцию генерации конкретного объекта Test объемом не более *K* вопросов из объекта типа TTestContent.

12. Карточка персоны содержит фамилию и дату рождения. Реализовать класс TListPerson для работы с картотекой персоналий. Класс должен содержать массив карточек персон. Реализовать методы добавления и удаления карточек персон, а также метод доступа к карточке по фамилии. Фамилии в массиве должны быть уникальны. Реализовать операции объединения двух картотек, операцию пересечения и вычисления разности (см. задание 10). Реализовать метод, выдающий по фамилии знак зодиака. Для этого в классе должен быть объявлен массив структур TZodiac с полями: название знака зодиака, дата начала и дата окончания периода. Индексом в массиве должен быть перечислимый тип zodiac.

13. Товарный чек содержит список товаров, купленных покупателем в магазине. Один элемент списка представляет собой пару: товар-сумма. Товар — это класс TGoods с полями кода и наименования товара, цены за единицу товара, количества покупаемых единиц товара. В классе должны быть реализованы методы доступа к полям для получения и изменения информации, а также метод вычисления суммы оплаты за товар. Для моделирования товарного чека реализовать класс TReceipt, полями которого являются номер товарного чека, дата и время его создания, список покупаемых товаров. В классе Receipt реализовать методы добавления, изменения и удаления записи о покупаемом товаре, метод поиска информации об определенном виде товара по его коду, а также метод подсчета общей суммы, на которую были осуществлены покупки. Методы добавления и изменения принимают в качестве аргумента объект класса TGoods. Метод поиска возвращает объект класса TGoods в качестве результата.

14. Информационная запись о книге в библиотеке содержит следующие поля: автор, название, год издания, издательство, цена. Для моделирования учетной карточки абонента реализовать класс TSubscriber, содержащий фамилию абонента, его библиотечный номер и список взятых в библиотеке книг. Один элемент списка состоит из информационной записи о книге, даты выдачи,

требуемой даты возврата и признака возврата. Реализовать методы добавления книг в список и удаления книг из него; метод поиска книг, подлежащих возврату; методы поиска по автору, издательству и году издания; метод вычисления стоимости всех подлежащих возврату книг. Реализовать операцию слияния двух учетных карточек, операцию пересечения и вычисления разности (см. задание 10). Реализовать операцию генерации конкретного объекта Debt (долг), содержащего список книг, подлежащих возврату из объекта типа TSubscriber.

15. Информационная запись о файле в каталоге содержит поля: имя файла, расширение, дата и время создания, атрибуты «только чтение», «скрытый», «системный», размер файла на диске. Для моделирования каталога реализовать класс TDirectory, содержащий название родительского каталога, количество файлов в каталоге, список файлов в каталоге. Один элемент списка включает в себя информационную запись о файле, дату последнего изменения, признак выделения и признак удаления. Реализовать методы добавления файлов в каталог и удаления файлов из него; метод поиска файла по имени, по расширению, по дате создания; метод вычисления полного объема каталога. Реализовать операцию объединения и операцию пересечения каталогов (см. задание 10). Реализовать операцию генерации конкретного объекта Group (группа), содержащего список файлов, из объекта типа TDirectory. Должна быть возможность выбирать группу файлов по признаку удаления, по атрибутам, по дате создания (до или после), по объему (меньше или больше).

16. Используя класс TTelephonePainment (задание III.16 из предыдущего параграфа), реализовать класс TListPayer. Класс содержит список плательщиков за телефонные услуги, дату создания списка, номер списка. Поля одного элемента списка — это: плательщик (класс TTelephonePainment), признак оплаты, дата платежа, сумма платежа. Реализовать методы добавления плательщиков в список и удаления их из него; метод поиска плательщика по номеру телефона и по фамилии, по дате платежа. Метод вычисления полной стоимости платежей всего списка. Реализовать операцию объединения и операцию пересечения списков (см. задание 10). Реализовать операцию генерации конкретного объекта Group (группа), содержащего список плательщиков, из объекта типа TListPayer. Должна быть возможность выбирать группу плательщиков по признаку оплаты, по атрибутам, по дате платежа, по номеру телефона.

17. Учебный план специальности является списком дисциплин, которые студент должен изучить за время обучения. Одна дисциплина представляет собой структуру с полями: номер дисциплины в плане, тип дисциплины (федеральная, региональная, по выбору), название дисциплины, семестр, в котором дисциплина изучается, вид итогового контроля (зачет или экзамен), общее количество часов, необходимое для изучения дисциплины, количество аудиторных часов, которые состоят из лекционных часов и часов практики. Реализовать класс TPlanEducation для моделирования учебного плана специальности. Класс должен содержать код и название специальности, дату утверждения, общее количество часов специальности по стандарту и список дисциплин. Один элемент списка дисциплин должен содержать запись о дисциплине, количество часов

для самостоятельной работы (разность между общим количеством часов и аудиторными часами), признак наличия курсовой работы, выполняемой по данной дисциплине. Реализовать методы добавления и удаления дисциплин; метод поиска дисциплины по семестру, по типу дисциплины, по виду итогового контроля; метод вычисления суммарного количества часов всех дисциплин; метод вычисления количества экзаменов и зачетов по семестрам. Реализовать операцию объединения, операцию вычитания учебных планов и операцию пересечения учебных планов (см. задание 10). Реализовать операцию генерации конкретного объекта Group (группа дисциплин), содержащего список дисциплин, из объекта типа TPlanEducation. Должна быть возможность выбирать группу дисциплин по типу, по семестру, по виду итогового контроля, по наличию курсовой работы. Должен осуществляться контроль за суммарным количеством часов, за количеством экзаменов в семестре (не более пяти и не менее трех).

18. Нагрузка преподавателя за учебный год представляет собой список дисциплин, преподаваемых им в течение года. Одна дисциплина представляется информационной структурой с полями: название дисциплины, семестр проведения, количество студентов, количество часов аудиторных лекций, количество аудиторных часов практики, вид контроля (зачет или экзамен). Реализовать класс TWorkTeacher, моделирующий бланк назначенной преподавателю нагрузки. Класс содержит фамилию преподавателя, дату утверждения, список преподаваемых дисциплин, объем полной нагрузки в часах и в ставках. Дисциплины в списке не должны повторяться. Объем в ставках вычисляется как частное от деления объема в часах на среднюю годовую ставку, одинаковую для всех преподавателей кафедры. Элемент списка преподаваемых дисциплин содержит дисциплину, количество часов, выделяемых на зачет (0,35 ч на одного студента) или экзамен (0,5 ч на студента), сумму часов по дисциплине. Реализовать добавление и удаление дисциплин; вычисление суммарной нагрузки в часах и ставках. Должен осуществляться контроль за превышением нагрузки более полутора ставок.

19. Создать класс TOctal для работы с беззнаковыми целыми восьмеричными числами, используя для представления числа массив из 100 элементов типа unsigned char, каждый элемент которого является восьмеричной цифрой. Младшая цифра имеет меньший индекс (единицы — в нулевом элементе массива). Реальный размер массива задается как аргумент конструктора инициализации. Реализовать арифметические операции, аналогичные встроенным для целых в C++, и операции сравнения.

20. Создать класс TMyString для работы со строками, аналогичными строкам Turbo Pascal (строка представляется как массив 255 байт, длина — в первом байте). Максимальный размер строки должен задаваться. Обязательно должны быть реализованы: определение длины строки, поиск подстроки в строке, удаление подстроки из строки, вставка подстроки в строку, сцепление двух строк.

21. Одна запись в списке запланированных дел представляет собой структуру TDailyItem, которая содержит время начала и окончания работы, описание и признак выполнения. Реализовать класс TDailySchedule, представляющий со-

бой план работ на день. Реализовать методы добавления, удаления и изменения планируемой работы. При добавлении проверять корректность временных рамок (они не должны пересекаться с уже запланированными мероприятиями). Реализовать метод поиска свободного промежутка времени. Условие поиска задает размер искомого интервала, а также временные рамки, в которые он должен попадать. Метод поиска возвращает структуру TDailyItem с пустым описанием вида работ. Реализовать операцию генерации объекта Redo (еще раз), содержащего список дел, не выполненных в течение дня, из объекта типа TDailySchedule.

22. Прайс-лист компьютерной фирмы включает в себя список моделей продаваемых компьютеров. Одна позиция списка (структура TModel) содержит марку компьютера, тип процессора, частоту работы процессора, объем памяти, объем жесткого диска, объем памяти видеокарты, цену компьютера в условных единицах и количество экземпляров, имеющихся в наличии. Реализовать класс TPriceList, полями которого являются дата его создания, номинал условной единицы в рублях и список продаваемых моделей компьютеров. В списке не должно быть двух моделей одинаковой марки. В классе TPriceList реализовать методы добавления, изменения и удаления записи о модели, метод поиска информации о модели по марке компьютера, по объему памяти, диска и видеокарты (равно или не меньше заданного), а также метод подсчета общей суммы. Реализовать методы объединения и пересечения прайс-листов (см. задание 10). Методы добавления и изменения принимают в качестве входного параметра объект класса TModel. Метод поиска возвращает объект класса TModel в качестве результата.

23. Реализовать класс TSet (множество) не более чем из 256 элементов-символов. Решение должно обеспечивать включение элемента в множество, исключение элемента из множества, объединение, пересечение и разность множеств, вычисление количества элементов в множестве, проверку присутствия элемента в множестве, проверку включения одного множества в другое.

ЛАНЬ®

7. НАСЛЕДОВАНИЕ

На С я могу просто делать ошибки, на С++ я могу их наследовать!

С сайта:
http://www.aforizmov.net/tema/tags/programmist/page/4/

В данной главе нам предстоит изучить технику программирования наследования. Тема наследования в отдельных аспектах уже обсуждалась в п. 1.4.3, 2.2.2, 3.3.

Наследование — это один из «трех китов» (наряду с инкапсуляцией и полиморфизмом), на котором стоит объектно-ориентированное программирование. При наследовании обязательно имеется класс-предок (родитель, порождающий класс) и класс-наследник (потомок, порожденный класс).

Механизм наследования позволяет определять новые классы на основе уже имеющихся классов. Класс, на основе которого создается новый класс, обычно называют базовым или родительским классом, а новый — производным классом или классом-потомком. Любой производный класс может, в свою очередь, стать базовым для других создаваемых классов. Таким образом, формируется иерархия классов, а при создании объектов — иерархия объектов. В иерархии объектов производный объект имеет возможность доступа к элементам данных и методам родительских объектов.

Наследование — важная часть объектно-ориентированного программирования. Выигрыш от него состоит в том, что наследование позволяет использовать существующий код многократно. Имея написанный и отлаженный базовый класс, мы можем его больше не модифицировать, при этом механизм наследования позволит нам приспосабливать его для работы в различных ситуациях.

Мы рассмотрим эти возможности наследования более детально после того, как познакомимся с некоторыми тонкостями, которые могут встретиться при использовании наследования.

7.1. Назначение и способы наследования

7.1.1. Метафора наследования

Понятие наследования введено в параграфе 1.4. Для лучшего понимания смысла наследования введем понятия.

Экстенсиональная составляющая (семантика) описания характеризует объем описания (от *лат*. extensio — объем понятия).

Интенсиональная составляющая (семантика) описания характеризует содержание описания (от *лат*. intensio — содержание понятия).

Рассмотрим эти понятия применительно к наследованию.

Пример. Пусть имеется описание классов:

```
class  TClass1
    {
      public:
```

```
    Integer A;
    void B();
};
    class TClass2 : public TClass1
        {
            public:
             double C;
             void D();
          };.
```

В заголовке класса **class TClass2 : public TClass1** указано, что класс **TClass2** является наследником класса **TClass1**. Об использовании ключевого слова **public** речь пойдет в следующем параграфе. Класс **TClass1**, в соответствии с договоренностями по умолчанию (не указана информация о наследовании), является непосредственным потомком класса **TObject**. В свою очередь, класс **TClass2** является непосредственным потомком класса **TClass1** и потомком класса **TObject**. Экстенсиональной составляющей описания классов будет весь текст, который мы видим. Рассмотрим теперь интенсиональную составляющую описания класса **TClass2**, которая означает, как мы понимаем приведенное описание классов. Речь идет о содержании класса-потомка:

```
class TClass2
{
        Integer A;
        double C;
        void B();
        void D();
};.
```

Как видно из последнего описания, в класс **TClass2** добавлено содержимое класса-предка. Это и есть наследование. На самом деле картина еще сложнее: к описанию класса **TClass1** необходимо добавить все доступное содержание класса **TObject**, которое, в свою очередь, добавится и в описание класса **TClass2**.

Таким образом, экстенсиональная семантика будет означать то, что мы фактически пишем. Интенсиональная семантика означает то, что мы имеем в виду, когда пишем.

Договоримся для простоты измерять объем семантики количеством знаков и обозначим через $О_э$ — объем экстенсионального описания и через $О_и$ — объем интенсионального описания. Тогда можно рассмотреть три случая:

1. $О_э=О_и$ — означает «что пишем, то и мыслим».
2. $О_э<О_и$ — предусматривает существование некоторых договоренностей, некоторого механизма извлечения смысла (интенсиональной семантики) того, что на самом деле записано неявно.
3. $О_э>О_и$ — также предусматривает существование механизма извлечения смысла; заметим, что в русском языке для этого случая существует понятие «словоблудие», т. е. пустая бессодержательная болтовня, преподносимая как серьезное рассуждение о пустяке.

Нас будет интересовать случай $О_э<О_и$. Естественно, что теперь нужно хорошо разобраться в механизме наследования, т. е. в сопровождающих наследование договоренностях. Прагматические правила разумного использования или неиспользования наследования классов были рассмотрены в главе 1.

Мы будем рассматривать два случая:

1. Названия полей и методов у потомка и предка разные, что демонстрируется последним примером.
2. Названия полей и методов у потомка и предка совпадают, например,

```
class TClass1
    {
      public:
        Integer A;
        double B;
        String C;
        void D();
        void E();
    };
class TClass2  : public TClass1
    {
      public:
        double A;
        String C;
        void D();
        void F();
     };
```

В соответствии с механизмом наследования в классе **TClass2** будут доступны:

```
class TClass2
    {
      double A;              // из TClass2
      double B;              // из TClass1
      String C;              // из TClass2
      void D();              // из TClass2
      void E();              // из TClass1
      void F();              // из TClass2
    };
```

Здесь, как и в предыдущем примере, для краткости не приведены результаты наследования класса **TObject**.

Сформулируем и прокомментируем теперь наши наблюдения.

1. Использование несовпадающих у потомка и предка названий полей и методов позволяет сократить экстенсиональную составляющую описания, т. е. имеем случай $О_э<О_и$, т. е. пишем $О_э$, а имеем в виду $О_и$. Однако не стоит преувеличивать практическое значение этого приема, так как выигрыш в количестве введенных знаков сопровождается снижением наглядности описания.

2. Использование совпадающих у потомка и предка названий полей и методов не сокращает визуально экстенсиональную составляющую описания, но тем не менее мы имеем случай $О_э<О_и$, т. е. пишем одно, а имеем в виду другое. Здесь заложен очень мощный механизм программистских приемов, который называется перекрытие, переопределение, пересмотр.

Ограничение. *Поля могут перекрываться в классе-потомке только полями, а методы — только методами.*

Таким образом, нам нужно рассмотреть, как наследуются и перекрываются поля и как наследуются и перекрываются (переопределяются, пересматриваются) методы. Свойства мы договорились рассматривать как разновидность полей, тем не менее в использовании свойств имеются свои очень интересные приемы наследования.

ЛАНЬ®

Контрольные вопросы

1. Объясните понятие «экстенсиональная составляющая описания».
2. Объясните понятие «интенсиональная составляющая описания».
3. Как происходит наследование полей с несовпадающими названиями?
4. Как происходит наследование полей с совпадающими названиями?
5. Как происходит наследование методов с несовпадающими названиями?
6. Как происходит наследование методов с совпадающими названиями?

7.1.2. Одиночное наследование классов

Простым, или одиночным, называется наследование классов, при котором производный класс имеет только одного родителя. Синтаксис класса-потомка имеет вид:

class <имя класса-потомка>[<модификатор наследования>]
<имя базового класса> **{**<тело класса>**}**.

Модификатор наследования определяет степень доступности элементов базового класса в классе-наследнике. Квадратные скобки не являются элементом синтаксиса, а показывают, что модификатор может отсутствовать.

При разработке классов для управления доступом к элементам класса также используются модификаторы, будем называть их модификаторами доступа. Модификаторы наследования и модификаторы доступа используют одни и те же ключевые слова. Основными модификаторами являются: **public** (общий), **private** (личный), **protected** (защищенный).

В литературе модификаторы наследования (доступа) называют также спецификаторами наследования (доступа). Какое название выбрать для повседневного использования — это дело вкуса, однако употребление чем-либо отличного от выбранного названия модификатора не должно заводить в тупик.

При разработке отдельных (не предполагающих их наследование) классов используются модификаторы доступа **public** и **private**. Первый модификатор предполагает неограниченную доступность полей и методов за пределами экземпляра класса. Второй модификатор ограничивает область видимости элементов класса только внутренними методами экземпляра класса. К теме использования модификаторов доступа полей и методов класса мы вернемся

в главе «Инкапсуляция». Здесь речь пойдет об использования модификаторов применительно к наследованию.

Рассмотрим пример. Задание: смоделировать цилиндр, который задается радиусом основания и высотой; из всех полезных свойств цилиндра нас интересует значение объема цилиндра. Тогда описание класса **TCilinder** может иметь вид:

```
class TCilinder
{
   private:
      double dubFCilinderRadius;
      double dubFCilinderHeight;
      void CircleArea();
   public:
      void SetCilinderRadius(double R);
      double GetCilinderRadius();
      void SetCilinderHeight (double H);
      double GetCilinderHeight ();
      double GetCilinderVolume();
};
```

В нашем цилиндре имеются поля **dubFCilinderRadius, dubFCilinderHeight** для хранения соответственно значений радиуса основания и высоты цилиндра. Как видим, указанные поля недоступны вне экземпляра класса. Доступ к полям осуществляется с помощью методов **SetCilinderRadius**, **GetCilinderRadius** и **SetCilinderHeight**, **GetCilinderHeight**. Эти методы доступны пользователям нашего класса без ограничений. Также неограниченным доступом обладает метод **GetCilinderVolume**, предназначенный для получения значения объема цилиндра. Остался нерассмотренным метод **CircleArea** (вычисление площади основания цилиндра), находящийся в приватной области доступа. Решение отнести данный метод к приватной области обусловлено следующими соображениями: во-первых, предполагается, что метод **CircleArea** удобно использовать при вычислении объема цилиндра в реализации метода **GetCilinderVolume**; во-вторых, в задании речь не идет о свойстве цилиндра «площадь основания», поэтому нет никакого смысла включать соответствующий метод в область неограниченной доступности. Есть еще одно соображение в полезности приведенного решения. Опыт нашего общения с заказчиками подсказывает, что потребность в вычислении площади основания пока не осознается нашим заказчиком. Если в процессе разработки проекта площадь основания цилиндра понадобится в явном виде, то реализация такой возможности будет заключаться в простом переносе метода **CircleArea** из области доступа **private** в область доступа **public**.

Контрольные вопросы

1. Что называется модификатором наследования?
2. Что называется модификатором доступа?

3. Какие ключевые слова используются для обозначения модификаторов наследования и модификаторов доступа?

4. Расскажите подробно о каждом модификаторе доступа.

7.1.3. Комбинация модификаторов наследования и модификаторов доступа

Доступность элементов базового класса из классов-наследников изменяется в зависимости от модификаторов доступа в базовом классе и модификатора наследования. Каким бы ни был модификатор наследования, приватные элементы базового класса недоступны в его потомках. В таблице 7.1 приведены все варианты доступности элементов базового класса в производном классе при любых вариантах доступности модификатора наследования. Модификатор **protected** означает доступность только в классах-потомках.

Таблица 7.1

Доступ к элементам базового класса в классе-наследнике

№	Модификатор доступа в базовом классе	Модификатор наследования	Доступ в производном классе	
			struct	class
1	public	отсутствует	public	private
2	protected	отсутствует	public	private
3	private	отсутствует	недоступны	недоступны
4	public	public	public	public
5	protected	public	protected	protected
6	private	public	недоступны	недоступны
7	public	protected	protected	protected
8	protected	protected	protected	protected
9	private	protected	недоступны	недоступны
10	public	private	private	private
11	protected	private	private	private
12	private	private	недоступны	недоступны

В основном приведенные в таблице 7.1 правила доступа в комментариях не нуждаются. Поясним содержание строк 10, 11. В них модификатор наследования **private** переводит доступность элементов класса в **private**. Поэтому при построении производных классов от класса-наследника приведет к тому, что все элементы класса-наследника, который будет играть здесь роль базового класса, станут недоступными.

Существует много возможностей для доступа, и полезно будет изучить программу примера, показывающую, какая комбинация будет работать, а какая нет. Рассмотрим листинг программы, предназначенной для испытания модификаторов наследования и модификаторов доступа.

```
class A                          // базовый класс
{
 int missingModA;
    private:
 int privateModA;      // модификатор доступа private
    protected:
  int protectedModA;    // модификатор доступа protected
```

```
    public:
  int publicModA;        // модификатор доступа public
};
class B : A             // модификатор наследования отсутствует
{
    public:
 void fun()
       {
         int a;
         a=missingModA;         // ошибка, нет доступа
         a=privateModA;   // ошибка, нет доступа
         a=protectedModA;         // так можно
         a=publicModA;    // так можно
                }
};
class C : public A             // модификатор наследования public
{
    public:
void fun()
      {
          int a;
          a=missingModA;          // ошибка, нет доступа
                  a=privateModA;        // ошибка, нет доступа
          a=protectedModA;        // так можно
          a=publicModA;   // так можно
      }
};
class D : protected A                  // модификатор наследования protected
{
    public:
void fun()
      {
         int a;
         a=missingModA;          // ошибка, нет доступа
         a=privateModA;    // ошибка, нет доступа
         a=protectedModA;          // так можно
         a=publicModA;     // так можно
      }
};
class E : private A           // модификатор наследования private
{
    public:
 void fun()
       {
         int a;
         a=missingModA;          // ошибка, нет доступа
         a=privateModA;    // ошибка, нет доступа
         a=protectedModA;          // так можно
```

```
        a=publicModA;     // ошибка, нет доступа
       }
};
int main()
{
    int a;
    B * missingB=new B();
    a=missingB–>missingModA;        // ошибка, нет доступа
    a=missingB–>privateModA;        // ошибка, нет доступа
    a=missingB–>protectedModA;      // ошибка, нет доступа
    a=missingB–>publicModA;         // ошибка, нет доступа
    C * publicC=new C();
    a=publicC–>missingModA;         // ошибка, нет доступа
    a=publicC–>privateModA;         // ошибка, нет доступа
    a=publicC–>protectedModA;       // ошибка, нет доступа
    a=publicC–>publicModA;          // так можно
    D * protectedD=new D();
    a=protectedD–>missingModA;      // ошибка, нет доступа
    a=protectedD–>privateModA;      // ошибка, нет доступа
    a=protectedD–>protectedModA;    // ошибка, нет доступа
    a=protectedD–>publicModA;       // ошибка, нет доступа
    E * privateE=new E();
    a=privateE–>missingModA;        // ошибка, нет доступа
    a=privateE–>privateModA;        // ошибка, нет доступа
    a=privateE–>protectedModA;      // ошибка, нет доступа
    a=privateE–>publicModA;         // ошибка, нет доступа
}.
```

В приведенном листинге для придания смысла (семантики) именам объектов и полей интенсивно используются префиксы: public, protected, private, missing (модификатор отсутствует). Содержание функции **fun()** показывает возможности доступа к компонентам класса A, внутри классов-потомков. Содержание функции **main()** демонстрирует возможности доступа к компонентам класса A во внешней функции, использующей объекты классов-наследников класса A. Доступность компонента класса A в каждом случае поясняется комментарием. Рекомендуем сравнить наши результаты с приведенными в таблице 7.1. Для принимающих на веру наши рассуждения читателей скажем: они совпадают.

Контрольные вопросы

1. Какие возможности доступа к компонентам некоторого класса во внешней функции?

7.1.4. Использование конструкторов и деструкторов при наследовании

Конструкторы не наследуются — они создаются в производном классе (если не определены программистом явно). Система программирования поступает с конструкторами следующим образом:

— если в базовом классе нет конструкторов или есть конструктор без аргументов (или аргументы присваиваются по умолчанию), то в производном

классе конструктор можно не писать — будут созданы конструктор копирования и конструктор без аргументов;

— если в базовом классе все конструкторы с параметрами, производный класс обязан иметь конструктор, в котором явно должен быть вызван конструктор базового класса;

— при создании объекта производного класса сначала вызывается конструктор базового класса, затем — производного.

Рассмотрим примеры.

```
// Базовый класс TObject с конструкторами по умолчанию
// Класс самолет
class TPlane
{
   private:
      double dubFFlightSpeed;
   public:
     TPlane(const double &s=0.0) {dubFFlightSpeed =s;} // конструктор без аргументов
}
// класс авиалайнер
class TAirliner : public TPlane {};
```

Класс **TPlane** является потомком класса **TObject**, в котором имеется конструктор. В классе **TPlane** определен конструктор инициализации, который играет роль конструктора без аргументов, так как аргумент задан константой. В классе-наследнике **TAirliner** конструкторы можно не писать.

```
class TNumberPairs
{
              int intFFirst, intFSecond;
            public:
              TNumberPairs(int f, int s) { intFFirst=f; intFSecond=s;}
}
class TPairSums : public TNumberPairs
         {
             int intFPairSum;
           public:
             TPairSums(int f, int s): TNumberPairs(f, s) { intFPairSum=f+s; }
         }
```

В последнем примере в классе **TNumberPairs** определен только конструктор инициализации (конструктор копирования создается автоматически, а конструктор с параметрами — нет). В классе-наследнике **TPairSums** конструктор базового класса явно вызывается в списке инициализации.

Деструктор класса, как и конструкторы, не наследуется, а создается. С деструкторами система программирования поступает следующим образом:

— при отсутствии деструктора в производном классе система создает деструктор по умолчанию;

— деструктор базового класса вызывается в деструкторе производного класса автоматически независимо от того, определен он явно или создан системой;

— деструкторы вызываются (для уничтожения объектов) в порядке, обратном вызову конструкторов.

Создание и уничтожение объектов выполняется по принципу: последним создан — первым уничтожен.

Контрольные вопросы

1. Какая особенность у конструкторов в процессе наследования?
2. Как нужно поступать с классом-потомком, если в базовом классе конструктор отсутствует?
3. Как нужно поступать с классом-потомком, если в базовом классе конструктор без параметров?
4. Как нужно поступать с классом-потомком, если в базовом классе все конструкторы с параметрами?
5. В каком порядке вызываются конструкторы базового класса и производного класса?
6. Какая особенность у деструкторов в процессе наследования?
7. Как нужно поступать с классом-потомком, если в базовом классе деструктор отсутствует?
8. Какие принципы вызова деструкторов?

7.2. Наследование полей и свойств

При изучении главы «Инкапсуляция» мы познакомимся с правилом хорошего стиля объектно-ориентированного программирования: поля рекомендуется размещать только в разделе **private** класса. Поэтому тема наследования полей не очень актуальна, так как содержимое раздела **private** не наследуется. Тем не менее в языке такая возможность предусмотрена и существуют правила наследования полей. Наследование свойств и полей осуществляется одинаково. Поэтому в настоящем параграфе мы не будем отделять свойства от полей.

Рассмотрим пример. Пусть имеется описание классов с использованием наследования:

```
class TClass1
    {
      public:
        Integer A;
        double B;
        String C;
    };
class TClass2  : public TClass1
    {
      public:
        double A;
        String C;
     };
```

Построим на основе этого описания объект **Object2**. После выполнения конструктора **TClass2 * Object2=new TClass2()**; получим структуру объекта, представленного на рисунке 7.1.

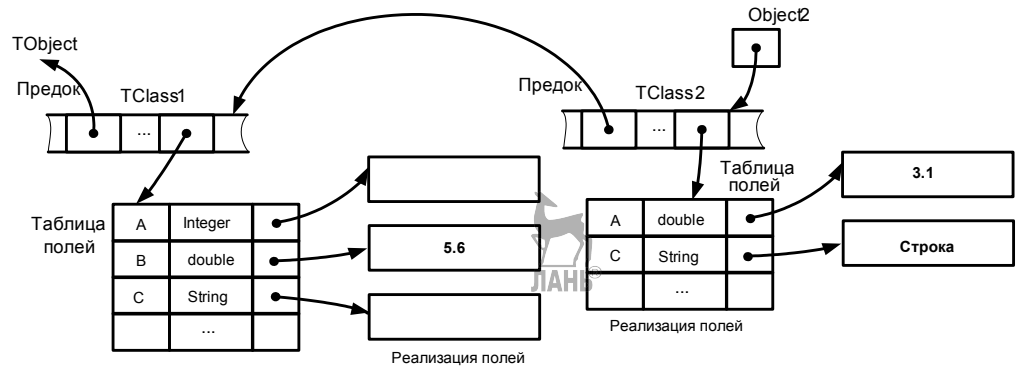


Рис. 7.1
Наследование полей

Структура приведена в сокращенном виде. На рисунке 7.1 не представлена часть иерархии объекта **Object2**, унаследованная от класса **TObject**. Кроме того, из всех таблиц приведены только таблицы полей, о которых пойдут дальнейшие рассуждения.

Рассмотрим присваивания:

Object2–>A=3.1;
Object2–>B=5.6;
Object2–>C=“Строка”;

Центральным моментом в этих присваиваниях является поиск реализации поля.

В первом присваивании через имя объекта **Object2** получаем доступ к структуре **TClass2**. В структуре объекта имеется таблица полей, в которой присутствует строка с переменной ***A***. Выбираем найденную строку таблицы и получаем доступ к реализации поля ***A***. В память записывается значение **3.1**, как это показано на рисунке 7.1.

Для второго присваивания ситуация сложнее. В таблице полей структуры **TClass2** переменная ***B*** не обнаруживается. Через ссылку на предка переходим к структуре **TClass1**. Из соответствующей переменной ***B*** строки таблицы полей попадаем в реализацию поля ***B***. Присваиваемое значение записывается. На рисунке 7.1 представлен результат операции присваивания.

Перейдем теперь к третьему присваиванию. Нетрудно заметить, что значение «Строка» попадет в реализацию поля ***C***, соответствующего структуре **TClass2**.

Наконец, видно, что через имя **Object2** мы никогда не сможем достигнуть реализации полей ***A*** и ***C*** структуры **TClass1**.

Отметим, что рассмотренные процессы поиска реализации полей ***A***, ***B***, ***C*** в структуре объекта **Object2** происходят еще на стадии компиляции программы, хотя в это время реализация указанных полей отсутствует. Тем не менее в структурах соответствующих классов уже присутствуют таблицы полей, в ко-

торых для каждого поля имеется ячейка, предназначенная для хранения ссылки на реализацию поля.

Рассмотрим теперь алгоритм поиска полей (свойств) в структуре объекта. Пусть объект имеет имя *A*, а поле (свойство) объекта *A* называется *b*, т. е. рассматривается конструкция *A–>b*.

Алгоритм 7.1.

Шаг 1. Используя имя *A* объекта, выбрать структуру объекта.

Шаг 2. Выполнить поиск поля (свойства) b в таблице полей (свойств).

Шаг 3. Если найдено, то стоп: успех, иначе выполнить шаг 4.

Шаг 4. Если выбрана структура экземпляра класса TObject, то стоп: неуспех, иначе выполнить шаг 5.

Шаг 5. Перейти к структуре экземпляра класса непосредственного предка и выполнить шаг 2.

Пояснение к алгоритму 7.1. На шаге 4 поиск в таблице полей не выполняется, так как в структурах класса TObject доступная таблица полей отсутствует.

Из алгоритма 7.1 следует, что введение в класс-потомок одноименного с базовым классом поля приводит к недоступности одноименного поля предка. Для доступа к одноименному полю предка рекомендуется использовать квалификаторы, например, пусть имеется описание классов:

```
class A
{
 public:
        int i;
        A(const int &x=5) { i=x; };
};
class B : public A
{
        public:
         int i;
         B(const int &x=10) { i=x; };
         void m() {i=i+A::i;};
};.
```

В операторе присваивания **i=i+A::i;** используется квалификатор **A::i**, который позволяет получить доступ к полю **i** класса **A**. Если выполнить следующие действия

```
B * b=new B();
b–>m();
```

то поле b–>i получит значение 15.

Выводы.

1. Поле или свойство обнаруживается в ближайшей на пути от текущего объекта структуре к структуре экземпляра класса TObject.

2. Поле потомка перекрывает одноименное поле предка.

3. Для доступа к перекрытому полю предка необходимо использовать квалификатор.

Контрольные вопросы

1. Объясните механизм поиска реализации поля.
2. Объясните, почему поле потомка перекрывает одноименное поле предка.
3. Как получить доступ к перекрытому полю класса-предка с помощью квалификатора?

7.3. Наследование методов

7.3.1. Статическое и динамическое наследование методов

Существует два принципиально различных механизма наследования методов:

— статическое наследование;

— динамическое наследование.

Методы, использующие механизм статического наследования, принято называть статическими.

При определении статических методов используется принцип умолчания: при отсутствии в описании метода признака механизма наследования нужно считать его статическим, например,

void A(); — статический метод.

Методы, использующие механизм динамического наследования, получили название динамических, или виртуальных, методов. Синтаксическим признаком отличия динамических и виртуальных методов является записываемое в определении метода служебное слово **virtual**, например,

void virtual B(); — виртуальный метод.

Служебное слово **virtual** в описании виртуальных методов опускать не разрешается (если мы не хотим изменять возможности их наследования).

Принципиальное отличие статических и виртуальных методов проявляется в специфике процессов установления связи вызова метода с реализацией метода.

Механизм статического наследования методов предполагает одновременную передачу классам-потомкам интерфейса класса и реализации метода. Реализуется это установлением связи вызова метода с реализацией метода во время трансляции программы: отсюда и название методов — статические.

Механизм динамического наследования передает потомкам класса только интерфейс метода. Реализуется это тем, что связь вызова метода с реализацией метода устанавливается только в процессе выполнения программы: отсюда и название методов — динамические.

Контрольные вопросы

1. Что является признаком статического метода?
2. Что является признаком виртуального метода?

7.3.2. Статическое наследование методов

Механизм наследования статических методов рассмотрим на примерах.

Пример 1. Пусть дано описание классов:

```
class TClass1
    {
       public:
          void Method1();
          void Method2();
    };
class  TClass2  : public TClass1
    {
        public:
            void Method2();
    };.
```

Отметим особенности приведенного описания классов:

1) класс **TClass2** является потомком класса **TClass1**;

2) описание метода **Method1** присутствует только в классе **TClass1**;

3) описание метода **Method2** присутствует как в классе **TClass1**, так и в классе **TClass2**.

Теперь рассмотрим реализацию методов классов **TClass1** и **TClass2**:

```
void TClass1::Method1()
 {
    Method2();
 }
void TClass1::Method2()
 {
     ShowMessage("Я из класса 1");
 }
void TClass2::Method2()
 {
    ShowMessage("Я из класса 2");
 }.
```

Пусть теперь имеется объявление двух объектов:

```
TClass1 * Object1;
TClass2 * Object2;.
```

После вызова конструкторов:

```
Object1 =new TClass1();
Object2 =new TClass2();
```

в динамической памяти будут созданы два объекта, доступ к которым производится с помощью переменных **Object1** и **Object2**. Рассмотрим устройство этих объектов.

Структура первого объекта представлена на рисунке 7.2, а структура второго объекта изображена на рисунке 7.3.

Пусть имеется вызов метода: **Object1–>Method1();**.

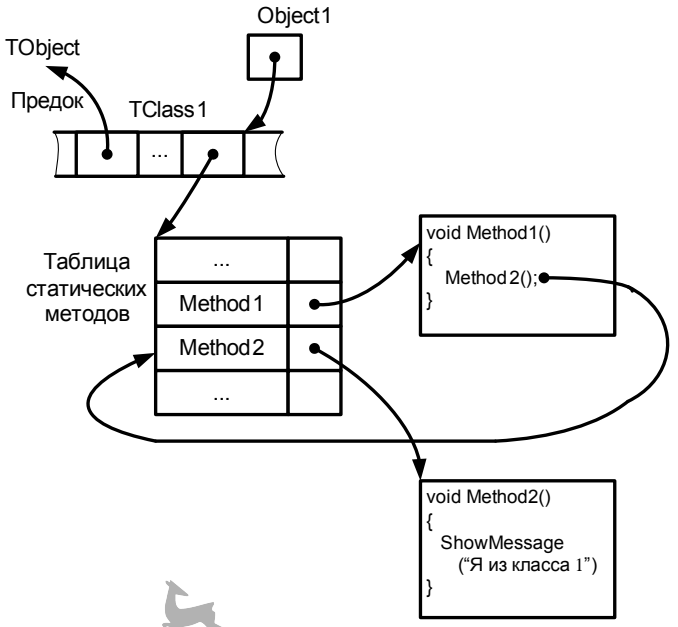


Рис. 7.2
Структура объекта **Object1**

Центральным моментом в рассматриваемой теме является установление связи вызова метода и реализации метода.

Шаг 1. Через имя **Object1** получаем доступ к экземпляру класса **TClass1** (рис. 7.2).

Шаг 2. В таблице статических методов находим строку с именем метода **Method1**.

Шаг 3. Разыменованием ссылки получаем доступ к реализации **Method1**.

В методе **Method1** имеется вызов **Method2**, связь **Method2** с реализацией осуществляется также с помощью таблицы статических методов. Таким образом, вызов **Object1–>Method1()** завершится выводом на экран сообщения «Я из класса 1».

Связь вызова метода: **Object1–>Method2()** с реализацией **Method2** устанавливается также за три шага (рис. 7.2): доступ к объекту, поиск в таблице статических методов строки с именем **Method2**, доступ к реализации **Method2**. Вызов метода **Object1–>Method1()** завершится выводом на экран сообщения «Я из класса 1».

Проведем теперь эксперименты с объектом **Object2**.

Пусть имеется вызов метода: **Object2–>Method1()**. Связь вызова **Object2–>Method1()** с реализацией **Method1** устанавливается следующим образом (рис. 7.3).

Шаг 1. Через имя **Object2** получаем доступ к экземпляру класса **TClass2**;

Шаг 2. В таблице статических методов **TClass2** осуществляется поиск строки с именем метода **Method1**. Как видно из рисунка 7.3, поиск завершился неуспехом.

Шаг 3. По ссылке на предка переходим к структуре экземпляра класса **TClass1**.

Шаг 4. В таблице статических методов **TClass1** осуществляется поиск строки с именем метода **Method1**. Поиск завершился успехом.

Шаг 5. Разыменованием ссылки получаем доступ к реализации **Method1**.

Шаг 6. В теле метода **Method1** имеется вызов метода **Method2**.

Шаг 7. В результате успешного поиска метода **Method2** в таблице статических методов экземпляра класса **TClass1** получаем доступ к реализации **Method2**.

Таким образом, вызов **Object2–>Method1();** завершится выводом на экран сообщения «Я из класса 1».

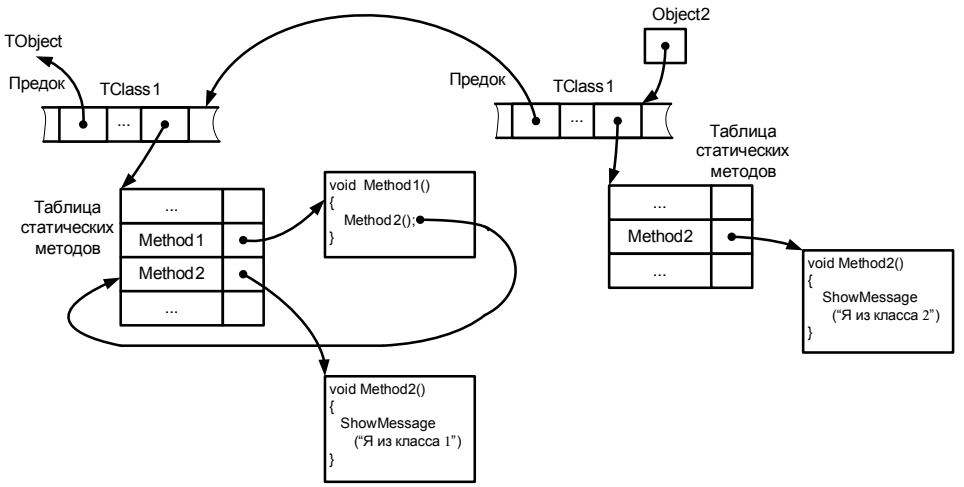


Рис. 7.3
Структура объекта **Object2**

Связь вызова **Object2–>Method2()** с реализацией **Method2** предлагаем установить самостоятельно. Подсказкой будет результат исполнения этого вызова: на экране появится сообщение «Я из класса 2».

Следующим экспериментом будет изучение вызова:

((TClass1 *) Object2)–>Method1();.

В этом вызове использовано приведение: объект **Object2** класса **TClass2** приводится к безымянному экземпляру класса **TClass1**. Связь вызова с реализацией **Method1** будет установлена следующим образом: в таблице статических методов **TClass1** осуществляется поиск строки с именем метода **Method1**. Поиск завершился успехом. Разыменованием ссылки получаем доступ к реализации **Method1**.

Таким образом, исполнение вызова **((TClass1 *) Object2)–>Method1()** приведет к выводу на экран сообщения «Я из класса 1».

Вызов **((TClass1 *) Object2)–>Method2();** предлагаем рассмотреть самостоятельно.

В заключение обратим еще раз внимание на очень важное обстоятельство: установление связи вызова метода с реализацией метода осуществляется во время трансляции программы. Этим и объясняется название рассматриваемых методов — «статические».

Выводы.

1. В статических методах связь вызова метода и реализации метода устанавливается в статике, т. е. во время трансляции программы, во время исполнения программы эта связь не меняется.

2. Методы дочерних классов перекрывают одноименные методы предка.

3. Для принудительного вызова перекрытого метода предка используется операция приведения к классу предка.

Рассмотрим теперь пример использования в классе полей и статических методов.

Пусть дано описание классов:

```
class TClass1
    {
       public:
          double Field;
          void SetData (double Value);
    };
class  TClass2  : public TClass1
    {
       public:
          String Field;
          void SetData (String Value);
    };
```

и пусть реализация методов выглядит следующим образом:

```
void TClass1::SetData(double Value)
   {
      Field=Value;
   }
void TClass2::SetData(String Value)
   {
      Field=Value;
      ((TClass1 *) this)–>SetData(StrToFloat(Value));
   }
```

Рассмотрим структуры объектов после введения описаний объектных переменных

TClass1 * Object1; TClass2 * Object2;

и исполнения конструкторов

Object1=new TClass1(); Object2=new TClass2();.

Структура объекта **Object1** представлена на рисунке 7.4, структура объекта **Object2** представлена на рисунке 7.5.

Пусть имеется оператор присваивания:

Object1–>Field=0.5;

Связь составного имени **Object1–>Field** с реализацией поля **Field** будет установлена следующей последовательностью действий (рис. 7.4).

Шаг 1. Через имя **Object1** получаем доступ к экземпляру класса **TClass1**.

Шаг 2. В таблице полей **TClass1** осуществляется поиск строки с именем **Field**. Поиск завершается успехом.

Шаг 3. Разыменованием ссылки получаем доступ к реализации поля **Field**, т. е. к выделенной по формату **double** области памяти.

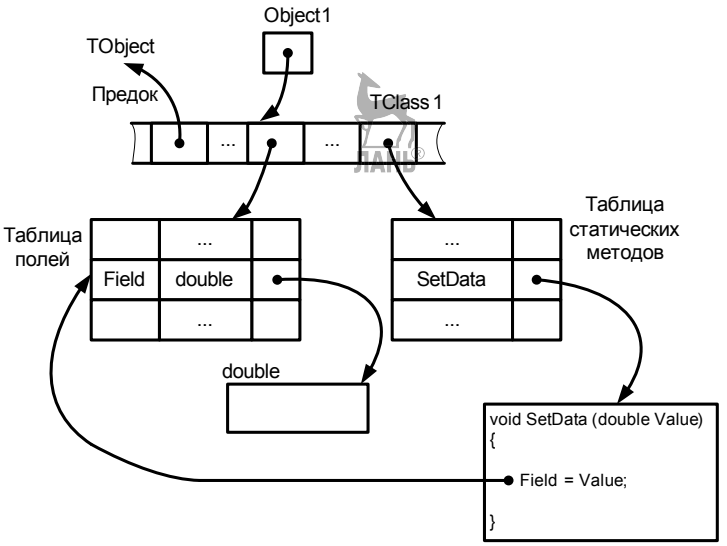


Рис. 7.4
Структура объекта **Object1**

В операторе присваивания

Object2–>Field=“Строка”;

те же действия будут выглядеть следующим образом (рис. 7.5).

Шаг 1. Через имя **Object2** получаем доступ к экземпляру класса **TClass2**.

Шаг 2. В таблице полей **TClass2** осуществляется поиск строки с именем **Field**. Поиск завершается успехом.

Шаг 3. Разыменованием ссылки получаем доступ к реализации поля **Field**, т. е. к выделенной по формату **String** области памяти.

Рассмотрим теперь три вызова.

Связь вызова **Object1–>SetData(0.5)** с реализацией будет установлена следующим образом (рис. 7.4).

Шаг 1. Через имя **Object1** получаем доступ к экземпляру класса **TClass1**.

Шаг 2. В таблице статических методов **TClass1** осуществляется поиск строки с именем метода **SetData**. Поиск завершился успехом.

Шаг 3. Разыменованием ссылки получаем доступ к реализации метода **SetData**.

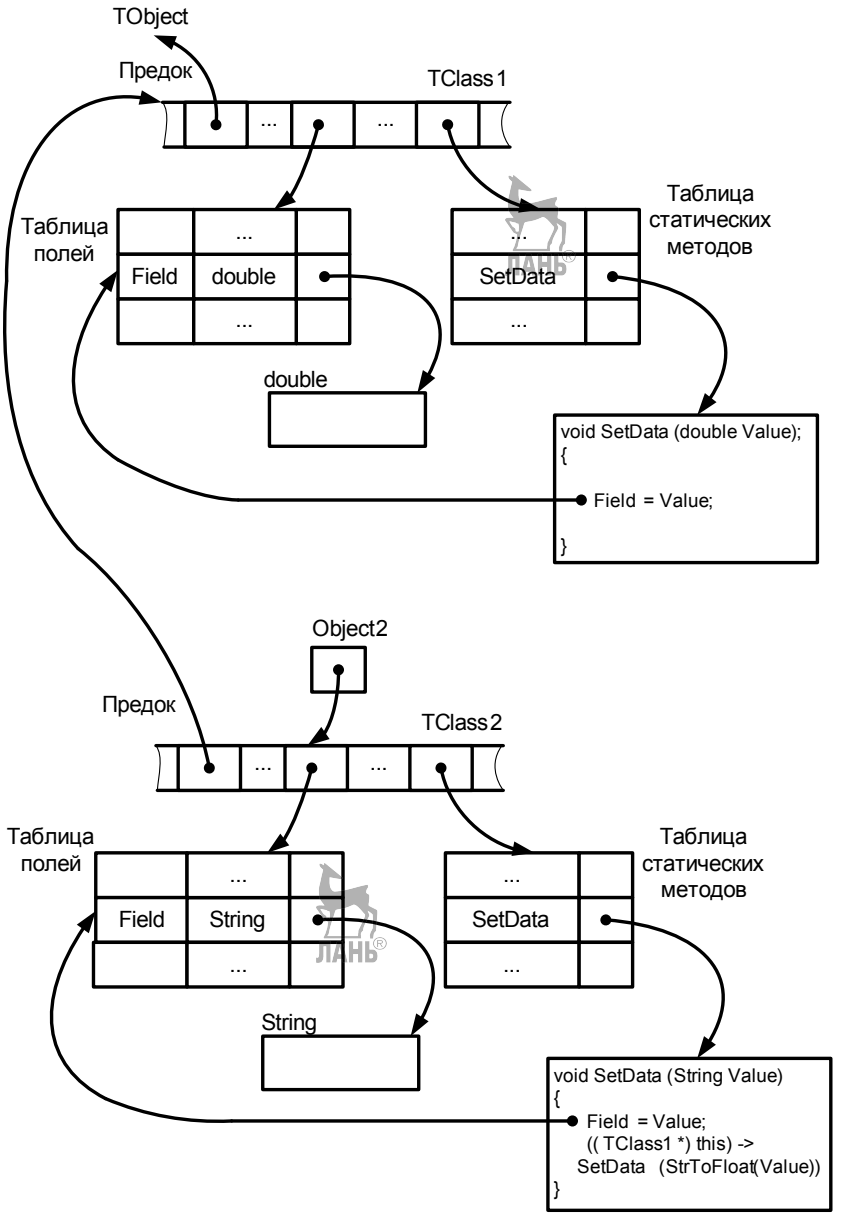


Рис. 7.5
Структура объекта **Object2**

В реализации метода **SetData** связь имени **Field** с реализацией поля в структуре объекта **Object1** (рис. 7.4) показана через ссылку на строку **Field** таблицы полей и последующей ссылки из строки таблицы полей на реализацию поля.

Связь вызова **Object2–>SetData("0.777");** с реализацией будет установлена следующим образом (рис. 7.5).

Шаг 1. Через имя **Object2** получаем доступ к экземпляру класса **TClass2**.

Шаг 2. В таблице статических методов **TClass2** осуществляется поиск строки с именем метода **SetData**. Поиск завершился успехом.

Шаг 3. Разыменованием ссылки получаем доступ к реализации метода **SetData**.

В реализацию метода **SetData** входит присваивание **Field=Value**, связь имени **Field** с реализацией показана на рисунке 7.5. Кроме того, имеется вызов:

((TClass1 *) this)–>SetData(StrToFloat(Value));

связь которого с реализацией будет установлена следующим образом (рис. 7.5).

Шаг 1. В заголовке реализации метода **void TClass2::SetData(String Value)** имеется указание на класс **TClass2**, поэтому через служебное слово **this** (этот) получаем доступ к экземпляру класса **TClass2**.

Шаг 2. С помощью приведения **(TClass1 *) this** получаем доступ к экземпляру класса **TClass1**.

Шаг 3. В таблице статических методов **TClass1** осуществляется поиск строки с именем метода **SetData**. Поиск завершился успехом.

Шаг 4. Разыменованием ссылки получаем доступ к реализации метода **SetData**.

Таким образом, в результате исполнения вызова **Object2–>SetData(«0.777»)** значение параметра будет записано два раза: в виде строки “**0.777”** в реализацию поля типа **String** объекта, непосредственно идентифицируемого именем **Object2**, а также — в виде вещественного числа **0.777** в реализацию экземпляра класса **TClass1**.

Выводы.

1. Одноименные поля перекрываются и недоступны в потомках.

2. Одноименные статические методы перекрываются, но доступны в потомках с помощью приведения к экземпляру класса-предка.

3. Доступ к перекрытым полям предка возможен только с помощью методов.

4. Установление связи вызова статического метода с реализацией метода устанавливается в статике, т. е. во время трансляции программы. Во время исполнения программы установленная связь не изменяется.

В заключение сформулируем алгоритм поиска реализации статического метода для установления связи вызова метода с реализацией метода.

Шаг 1. Если в вызове присутствует приведение к экземпляру класса-предка, то перейти к предку.

Шаг 2. Просмотреть текущую таблицу статических методов.

Шаг 3. Если метод найден в таблице, то стоп — успех, иначе выполнить шаг 4.

Шаг 4. Если имеется предок, то выполнить переход к предку и перейти к шагу 2, иначе стоп — неуспех.

Замечание. Содержание шага 4 можно переформулировать следующим образом.

Шаг 4. Если текущим является экземпляр класса **TObject**, то стоп — неуспех, иначе выполнить переход к предку и перейти к шагу 2.

Контрольные вопросы

1. Объясните механизм установления связи вызова статического метода с реализацией метода.
2. Объясните механизм установления связи вызова статического метода с использованием приведения к экземпляру класса-предка с реализацией метода.
3. Каким образом можно получить доступ к перекрытому статическому методу предка?
4. Каким образом можно получить доступ к перекрытому полю предка?

7.3.3. Динамическое наследование методов

Виртуальные (динамические) методы отличаются от статических методов тем, что связь вызова и реализации метода устанавливается в динамике, т. е. во время исполнения программы.

Механизм наследования виртуальных методов рассмотрим на примере.

Пусть дано описание классов:

```
class TClass1
    {
        public:
            void  virtual Method1();
      };
class  TClass2 : public TClass1
    {
        public:
            void Method1();
     };
```

Отметим особенности приведенного описания классов:

1) класс **TClass2** является потомком класса **TClass1**;
2) метод **Method1** в классе **TClass1** объявлен как виртуальный;
3) метод **Method1** в классе **TClass2** переопределяет одноименный метод из класса **TClass1**.

Теперь рассмотрим реализацию методов классов **TClass1** и **TClass2**:

```
void TClass1::Method1()
   {
     ShowMessage("Я из класса 1");
   }
void TClass2::Method1()
   {
     ShowMessage("Я из класса 2");
   }
```

Пусть теперь имеется объявление двух объектов:

```
TClass1 * Object1;
TClass2 * Object2;
```

После вызова конструкторов:

Object1=new TClass1();

Object2=new TClass2();

в динамической памяти будут созданы два объекта, доступ к которым производится с помощью переменных **Object1** и **Object2**. Рассмотрим устройство этих объектов.

Структура первого объекта изображена на рисунке 7.6.

Структура второго объекта представлена на рисунке 7.7.

Пусть имеется вызов метода: **Object1–>Method1();**.

Центральным моментом в рассматриваемой теме является установление связи вызова метода **Object1–>Method1()** с реализацией метода **Method1()** (рис. 7.6).

Шаг 1. Через имя **Object1** получаем доступ к объекту (экземпляру класса **TClass1**).

Шаг 2. В таблице виртуальных методов находим строку с именем метода **Method1**.

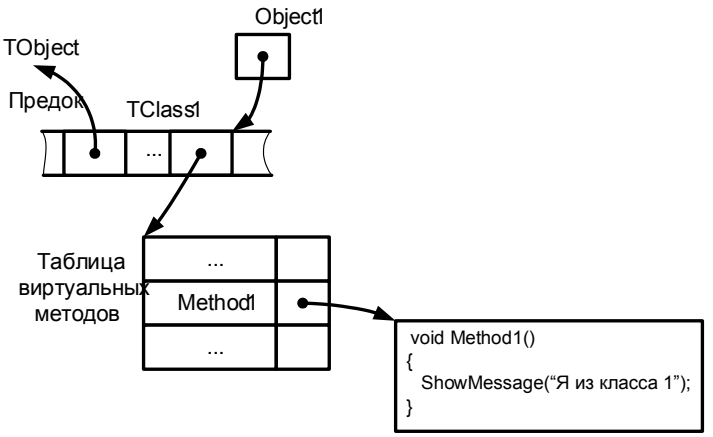


Рис. 7.6
Структура объекта **Object1**

Шаг 3. Разыменованием ссылки получаем доступ к реализации **Method1**.

Таким образом, вызов **Object1–>Method1()** завершится выводом на экран сообщения «Я из класса 1».

Проведем теперь эксперименты с объектом **Object2**.

Пусть имеется вызов метода: **Object2–>Method1()**.

Связь вызова **Object2–>Method1** с реализацией **Method1** устанавливается следующим образом (рис. 7.7).

Шаг 1. Через имя **Object2** получаем доступ к экземпляру класса **TClass2**.

Шаг 2. В таблице виртуальных методов **TClass2** находим строку с именем метода **Method1**.

Шаг 3. Осуществляем переход к реализации **Method1**.

Таким образом, вызов **Object2–>Method1();** завершится выводом на экран сообщения «Я из класса 2».

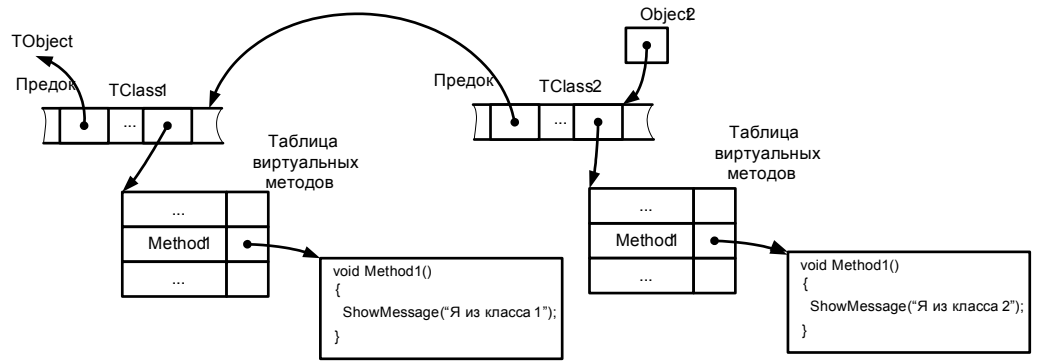


Рис. 7.7
Структура объекта **Object2**

Следующим экспериментом будет изучение вызова:

((TClass1 *) Object2)–>Method1();

Шаг 1. Через имя **Object2** получаем доступ к экземпляру класса **TClass2**.

Шаг 2. Выполняется операция приведения к экземпляру класса **TClass1**.

Шаг 3. В таблице виртуальных методов **TClass1** находим строку с именем метода **Method1**.

Шаг 4. Разыменованием ссылки получаем доступ к реализации **Method1**.

Таким образом, исполнение вызова **((TClass1 *) Object2)–>Method1()** приведет к выводу на экран сообщения «Я из класса 1».

Попытаемся теперь сравнить поведение динамических и виртуальных методов с поведением статических методов. В приведенных примерах никаких отличий не прослеживается. Однако имеется принципиальное отличие: установление связи вызова динамического или виртуального метода с реализацией метода осуществляется во время исполнения программы. Этим объясняется и общее название рассматриваемых методов — «динамические». Более содержательные примеры будут приведены в главе «Полиморфизм».

Выводы.

1. В виртуальных методах связь вызова метода и реализации метода устанавливается в динамике, т. е. во время исполнения программы.

2. Методы дочерних классов переопределяют одноименные методы предка.

3. Для принудительного вызова перекрытого метода предка используется приведение для доступа к структуре экземпляра класса-предка.

Основное назначение виртуальных методов состоит в том, чтобы облегчить модификацию уже разработанной программы. Программу можно реализовать как с помощью статических методов, так и с помощью динамических или виртуальных методов. Если в дальнейшем могут появиться новые структуры (типы) обрабатываемых данных, то во втором случае переделок программы будет, как правило, меньше. Основным приемом программирования с использованием виртуальных методов является полиморфизм. Полиморфизму посвящена отдельная глава.

Контрольные вопросы

1. Объясните механизм установления связи вызова виртуального метода с реализацией метода.
2. Объясните механизм установления связи вызова виртуального метода с использованием приведения к экземпляру класса-предка с реализацией метода.
3. Каким образом можно получить доступ к перекрытому виртуальному методу предка?
4. Каким образом можно получить доступ к перекрытому полю предка?

7.4. Использование наследования в разработке программ

Наследование — важная часть объектно-ориентированного программирования. Выигрыш от него состоит в том, что наследование позволяет использовать существующий код многократно. Имея написанный и отлаженный базовый класс, мы можем его больше не модифицировать, при этом механизм наследования позволяет нам приспособить его для работы в различных ситуациях. Используя уже написанный код, мы экономим время и деньги, а также увеличиваем надежность программы. Наследование может помочь и при начальной постановке задачи программирования, разработке общей структуры программы.

Важным результатом повторного использования кода является упрощение распространения библиотек классов. Программист может использовать классы, созданные кем-то другим, без модификации их кода, просто создавая производные классы, подходящие для частной ситуации.

Процесс разработки программы был основательно изменен с появлением объектно-ориентированного программирования. Это связано не только с использованием классов, но и с использованием наследования классов. Рассмотрим проектную ситуацию.

Пусть программист А, исходя из поставленных перед ним задач, создает класс. Пусть это будет класс TGramsKilograms, моделирующий работу с мерами веса: грамм, килограмм. Методы класса предназначены для выполнения арифметических операций с определенными разработчиком мерами веса.

Программисту Б нравится этот класс, но он считает, что класс может быть улучшен путем введения меры веса миллиграмм. Решением будет создание нового класса TMilligramsGramsKilograms (можно придумать более короткое, но более емкое по смыслу название). Новый класс является производным от класса TGramsKilograms, но включает в себя расширения, необходимые для реализации функциональности класса.

Программисты В и Г затем создают приложения, использующие класс TMilligramsGramsKilograms.

Программист Б может не иметь доступа к исходному коду, реализующему класс TGramsKilograms, а программисты В и Г могут не иметь исходного кода класса TMilligramsGramsKilograms. Но в C++ существует возможность повторного использования кода, поэтому программист Б может использовать и дополнять работу программиста А, а В и Г могут использовать работы Б (и А).

Заметим, что различия между разработчиками инструментов программного обеспечения становятся расплывчатыми. Программист А создает общецелевой программный продукт, класс TGramsKilograms. Программист Б, в свою очередь, создает более специализированный класс TMilligramsGramsKilograms. Программисты В и Г пишут приложения. А — разработчик инструмента, а В и Г — разработчики приложений. Б находится где-то посередине. В любом случае объектно-ориентированное программирование становится более гибким, но в то же время и более сложным.

ЛАНЬ®

Задачи

Для решения приведенных задач рекомендуется использовать три приема решения:

а) нарисовать структуры объектов и проследить процессы установления связи между вызовами методов и реализацией методов;

б) мысленно представить структуры объектов и установить связь вызова каждого метода с его реализацией;

в) набрать в системе программирования RAD Studio приведенные программы и проследить за их выполнением.

Все доступные вне объекта поля реализовать с помощью свойств.

I. Наследование и использование методов

Список заданий.

I.1. Дано описание классов:

```
class TA : public TObject
   {
            public:
            void A() {ShowMessage(«TA::A()»);};
   };
class TB : public TA
   {
            public:
                  void A() {ShowMessage(«TB::A()»);};
                  void B() {ShowMessage(«TB::B()»);};
   };
class TC : public TB
   {
            public:
                  void A() {ShowMessage(«TC::A()»);};
                  void B() {ShowMessage(«TC::B()»);};
                  void C() {ShowMessage(«TC::C()»);};
    };
```

ЛАНЬ®

После создания объектов:

```
TA* objA=new TA;
TB* objB=new TB;
TC* objC=new TC;
```

нужно сопоставить выходное сообщение каждому вызову:

```
objA–>A();
objB–>A();
objB–>B();
objC–>A();
objC–>B();
objC–>C();
((TA*) objB)–>A();
((TA*) objC)–>A();
((TB*) objC)–>A();
((TB*) objC)–>B();
```

I.2. Для объектов из задачи I.1 нужно сопоставить выходное сообщение (ShowMessage) каждому вызову метода:

```
objA=objB;
objA–>A();
objB–>A();
objB–>B();
objC–>A();
objC–>B();
objC–>C();
((TA*) objB)–>A();
((TA*) objC)–>A();
((TB*) objC)–>A();
((TB*) objC)–>B();
```

I.3. Для объектов из задачи I.1 нужно сопоставить выходное сообщение (ShowMessage) каждому вызову метода:

```
objA=objC;
objA–>A();
objB–>A();
objB–>B();
objC–>A();
objC–>B();
objC–>C();
((TA*) objB)–>A();
((TA*) objC)–>A();
((TB*) objC)–>A();
((TB*) objC)–>B();
```

II. Статическое наследование

Во всех заданиях реализовать операторы вывода сообщений на экран, методы получения значений полей и методы установки значений полей, а также необходимые конструкторы (если это не указано в задании явно). Конструкторы и методы обязательно должны проверять параметры на допустимость; в случае неправильных данных — выводить сообщение об ошибке и заканчивать работу.

Программа должна демонстрировать использование всех функций и методов.

Список заданий.

II.1. Создать базовый класс TCar (машина), характеризуемый торговой маркой (строка), числом цилиндров, мощностью. Определить методы переназначения и изменения мощности. Создать производный класс TLorry (грузовик), характеризуемый также грузоподъемностью кузова. Определить функции переназначения марки и изменения грузоподъемности.

II.2. Создать класс TPair (пара чисел); определить методы изменения полей и сравнения пар: пара p1 больше пары p2, если (p1–>first>p2–>first) или (p1–>first=p2–>first) и (p1–>second>p2–>second). Определить класс-наследник TFraction с полями: целая часть числа и дробная часть числа. Определить полный набор методов сравнения.

II.3. Создать класс TLiquid (жидкость), имеющий поля названия и плотности. Определить методы переназначения и изменения плотности. Создать производный класс T*A*lcohol (спирт), имеющий крепость. Определить методы переназначения и изменения крепости.

II.4. Создать класс TPair (пара чисел); определить методы изменения полей и вычисления произведения чисел. Определить производный класс TMyRectangle (прямоугольник) с полями-сторонами. Определить методы вычисления периметра и площади прямоугольника.

II.5. Создать класс TMan (человек) с полями: имя, возраст, пол и вес. Определить методы переназначения имени, изменения возраста и изменения веса. Создать производный класс TStudent, имеющий поле года обучения. Определить методы переназначения и увеличения года обучения.

II.6. Создать класс TTriad (тройка чисел); определить методы изменения полей и вычисления суммы чисел. Определить производный класс TTriangle с полями-сторонами. Определить методы вычисления углов и площади треугольника.

II.7. Создать класс TTriangle с полями-сторонами. Определить методы изменения сторон, вычисления углов, вычисления периметра. Создать производный класс TEquilateral (равносторонний), имеющий поле площади. Определить метод вычисления площади.

II.8. Создать класс TTriangle с полями-сторонами. Определить методы изменения сторон, вычисления углов, вычисления периметра. Создать производный класс TRightAngled (прямоугольный), имеющий поле площади. Определить метод вычисления площади.

II.9. Создать класс TPair (пара чисел); определить методы изменения полей и вычисления произведения чисел. Определить производный класс TRightAngled с полями-катетами. Определить методы вычисления гипотенузы и площади треугольника.

II.10. Создать класс TTriad (тройка чисел); определить метод сравнения триад (см. задание II.2). Определить производный класс TDate с полями: год, месяц и день. Определить полный набор методов сравнения дат.

II.11. Создать класс TTriad (тройка чисел); определить метод сравнения триад (см. задание II.2). Определить производный класс Time с полями: час, минута и секунда. Определить полный набор методов сравнения моментов времени.

II.12. Реализовать класс-оболочку TNumber для числового типа вещественное. Реализовать методы сложения и деления. Создать производный класс TReal, в котором реализовать метод возведения в произвольную степень, и метод для вычисления логарифма числа.

II.13. Создать класс TTriad (тройка чисел); определить методы увеличения полей на 1. Определить производный класс TMyDate с полями: год, месяц и день. Переопределить методы увеличения полей на 1 и определить метод увеличения даты на n дней.

II.14. Реализовать класс-оболочку TNumber для числового типа double. Реализовать методы умножения и вычитания. Создать производный класс TReal, в котором реализовать метод, вычисляющий корень произвольной степени, и метод для вычисления числа n в данной степени.

II.15. Создать класс TTriad (тройка чисел); определить методы увеличения полей на 1. Определить класс-наследник TTime с полями: час, минута, секунда. Переопределить методы увеличения полей на 1 и определить методы увеличения на n секунд и минут.

II.16. Создать базовый класс TPair (пара целых чисел) с операциями проверки на равенство и перемножения полей. Реализовать операцию вычитания пар по формуле $(a, b) - (c, d) = (a - b, c - d)$. Создать производный класс TRational; определить новые операции сложения $(a, b)+(c, d)=(ad+be, bd)$ и деления $(a, b)/(c, cl)=(ad, be)$; переопределить операцию вычитания $(a, b) - (c, d)=(ad - be, bd)$.

II.17. Создать класс TPair (пара чисел); определить метод перемножения полей и операцию сложения пар $(a, b)+(c, d)=(a+b, c+d)$. Определить производный класс TComplex с полями: действительная и мнимая части числа. Определить методы умножения $(a, b)*(c, d)=(ac - bd, ad+be)$ и вычитания $(a, b) - (c, d)=(a - b, c - d)$.

II.18. Создать класс TPair (пара целых чисел); определить методы изменения полей и операцию сложения пар $(a, b)+(c, d)=(a+b, c+d)$. Определить класс-наследник TLong с полями: старшая часть числа и младшая часть числа. Переопределить операцию сложения и определить методы умножения и вычитания.

II.19. Создать базовый класс TTriad (тройка чисел) с операциями сложения с числом, умножения на число, проверки на равенство. Создать производный класс TVector3D, задаваемый тройкой координат. Должны быть реализованы: операция сложения векторов, скалярное произведение векторов.

II.20. Создать класс TPair (пара целых чисел); определить метод умножения на число и операцию сложения пар $(a, b)+(c, d)=(a+b, c+d)$. Определить класс-наследник TMoney с полями: рубли и копейки. Переопределить операцию сложения и определить методы вычитания и деления денежных сумм.

III. Динамическое наследование

Создать базовый класс TArray с полями: массив типа unsigned char и поле для хранения количества элементов у текущего объекта-массива. Максимально возможный размер массива задается статической константой. Реализовать конструктор инициализации, задающий количество элементов и начальное значение (по умолчанию 0).

Реализовать в классе TArray виртуальную функцию поэлементного сложения массивов. Реализовать два класса, переопределив виртуальную функцию сложения. Вызывающая программа должна продемонстрировать все варианты вызова виртуальных функций.

III.1. Создать класс TFraction (задание I.6 из раздела 6) и класс TBitstring (задание I.1 из раздела 6).

III.2. Создать класс TBitstring (задание I.1 из раздела 6) и класс THex (задание I.3 из раздела 6).

III.3. Создать класс TDecimal (задание I.2 из раздела 6) и класс TBitstring (задание I.1 из раздела 6).

III.4. Создать класс TMoney (задание I.4 из раздела 6) и класс TBitstring (задание I.1 из раздела 6).

III.5. Создать класс TDecimal (задание I.2 из раздела 6) и класс TBitstring (задание I.1 из раздела 6).

III.6. Создать класс THex (задание I.3 из раздела 6) и класс TBitstring (задание I.1 из раздела 6).

III.7. Создать класс TOctal (задание I.19 из раздела 6) и класс TBitstring (задание I.1 из раздела 6).

III.8. Создать класс THex (задание I.3 из раздела 6) и класс TOctal (задание I.19 из раздела 6).

III.9. Создать класс Hex (задание I.3 из раздела 6) и класс Money (задание I.4 из раздела 6).

III.10. Создать класс Hex (задание I.3 из раздела 6) и класс Fraction (задание 29).

III.11. Создать класс Decimal (задание I.2 из раздела 6) и класс Hex (задание I.3 из раздела 6).

III.12. Создать класс Octal (задание I.19 из раздела 6) и класс Hex (задание I.3 из раздела 6).

III.13. Создать класс Decimal (задание I.2 из раздела 6) и класс Octal (задание I.19 из раздела 6).

III.14. Создать класс Decimal (задание I.2 из раздела 6) и класс TMyString (задание I.20 из раздела 6).

III.15. Создать класс Bitstring (задание 2.1) и класс TMyString (задание I.20 из раздела 6).

III.16. Создать класс Fraction (задание I.6 из раздела 6) и класс TMyString (задание I.20 из раздела 6).

III.17. Создать класс Hex (задание I.3 из раздела 6) и класс TMyString (задание I.20 из раздела 6).

III.18. Создать класс Octal (задание I.19 из раздела 6) и класс TMyString (задание I.20 из раздела 6).

III.19. Создать класс Money (задание I.4 из раздела 6) и класс TMyString (задание I.20 из раздела 6).

III.20. Создать класс Money (задание I.4 из раздела 6) и класс Hex (задание I.3 из раздела 6).

ЛАНЬ®

ЛАНЬ®

8. ИНКАПСУЛЯЦИЯ

Когда кто-то говорит: «Мне нужен язык программирования, в котором достаточно только сказать, что мне нужно сделать», — дайте ему леденец.

Алан. Дж. Перлис

Инкапсуляция — это механизм программирования, который связывает код (действия) и данные (состояния), которыми он манипулирует, и при этом предохраняет их от вмешательства извне и неправильного использования. При связывании кода и данных создается объект, т. е. объект — это элемент, который поддерживает инкапсуляцию.

В самом общем виде понятие «инкапсуляция» определяется как результат скрытия, помещения в капсулу, ограничения пространства. В программировании инкапсуляция скрывает внутреннюю реализацию программных элементов за внешней оболочкой-интерфейсом. Инкапсуляция освобождает программиста, использующего в своих проектах существующие программные компоненты, от детального их изучения, ограничиваясь ознакомлением только с интерфейсом. В структурном программировании единицей инкапсуляции является функция. В объектно-ориентированных языках программирования единица инкапсуляции — это объект.

В первой главе объект был определен как осязаемая реальность, обладающая индивидуальным поведением. Капсула, или объект, содержит данные состояния объекта, операции и протоколы сообщений, которые обрабатывает объект. Инкапсуляция позволяет выделить объект в отдельную программную единицу и создать условия реализации его индивидуальности. Это приводит к упрощению программы, поскольку из нее исключаются второстепенные детали. Инкапсуляцию можно сравнить с работой автомобиля с точки зрения типичного водителя. Многие водители не разбираются в устройстве авто, но управляют им. Существует специальный интерфейс, который упрощает сложные операции управления двигателем или тормозной системой. Сказанное относится к инкапсуляции в объектно-ориентированной программе — многие подробности внутреннего устройства скрываются от пользователя-программиста, что позволяет ему сосредоточиться на решении поставленных задач. Таким образом, инкапсуляция создает условия для использования и оформления принципа абстрагирования в программировании.

В настоящей главе рассматриваются приемы обеспечения инкапсуляции на уровне данных и операций. Также включен материал по очень близкой и тесно связанной с инкапсуляцией темой: особенности использования программных модулей.

8.1. Назначение и свойства инкапсуляции

Хороший интерфейс класса скрывает внутренние детали его реализации, открывая внешнему миру только «лицо», которое отделено от «внутренностей».

Определим назначение инкапсуляции четырьмя направлениями:

1) соединить вместе и локализовать все элементы описания типа данных;

2) оградить программиста от подробностей реализации используемого класса;

3) защитить данные от несанкционированного использования;

4) обеспечить возможность тиражирования программных решений.

Необходимо отметить, что перечисленные возможности автоматически не обеспечиваются объектной технологией и даже не являются бесконфликтными. Поэтому придание программной системе указанных свойств требует творческого подхода и соблюдения определенной дисциплины.

Потребность в использовании инкапсуляции обусловлена желанием упростить совместную работу проектировщиков программного обеспечения, а также потребностью создания комфортных условий сопровождения и развития программ.

В объекте необходимо инкапсулировать все свойства объектной абстракции, включая значения абстракций данных (переменные состояния) и функциональные абстракции (сообщения и методы). Поскольку капсула охватывает одну связанную группу подобъектов, ее интерфейс должен быть связанным — все его части должны иметь непосредственное отношение друг к другу. Таким образом, мы говорим, что объект инкапсулирует абстракции, создавая условия придания объекту определенного поведения.

Для объектов (инкапсулированных абстракций) желательны следующие свойства:

— четкая граница; эта граница определяет область видимости инкапсуляции. В C++ область видимости инкапсуляции для объекта распространяется на описание его класса, включая описания родительских классов;

— хорошо разработанный интерфейс; интерфейс описывает то, как объект взаимодействует с другими объектами или сегментами программы. В C++ интерфейс задается с помощью свойств и прототипов методов. Другие объекты могут осуществлять доступ к объекту только через такой контролируемый интерфейс;

— скрытая реализация объекта; внутреннее представление объекта невидимо для других объектов и не может быть ими изменено. В C++ эта защита является гибкой и может управляться благодаря концепции закрытых (private), защищенных (protected) и открытых (public) разделов описания класса (модификаторов доступа).

Контрольные вопросы

1. Сформулируйте назначение инкапсуляции.
2. Какими свойствами должна обладать инкапсулированная абстракция (объект)?

8.2. Средства инкапсуляции

В настоящем параграфе рассмотрены средства языка программирования C++Builder, использование которых позволяет обеспечить полноценную инкапсуляцию объектов.

1. Средство построения специальных описаний типов данных — классов, в которых объединены все компоненты типа данных: поля, свойства, методы, сообщения. Одной из целей введения классов в C++ является поддержка практики объектно-ориентированного программирования и устранение недостатков, свойственных применению более мелких единиц модульности — функций. Новые типы данных C++ представляют собой нечто большее, чем просто структуры языка C с функциями. В частности, эти новые типы данных получили и новое качество: объекты теперь обладают поведением. Бесспорно, это полностью оправдывает введение нового ключевого слова **class**. Переход на классы был настолько важен, что исключению ключевого слова **struct** из языка не произошло только из-за требований совместимости с языком C. Поэтому в языке программирования C++ слово **class** практически полностью идентично ключевому слову **struct**, существует только одно отличие: по умолчанию элементы класса являются закрытыми, а элементы структуры — открытыми. Закрытость (открытость) элементов класса означает недоступность (доступность) их за пределами соответствующего экземпляра класса.

Первая цель введения конструкции класса состоит в связывании данных и операций в один синтаксический блок и обозначении общей принадлежности типу данных этих элементов программы.

Следующая цель заключается в устранении конфликтов имен, благодаря чему данные и функции в разных классах могут без проблем использовать одни и те же имена.

Третья важная цель — позволить разработчику предоставляющего услугу объекта-сервера управлять внешним доступом получающих услуги классов-клиентов.

Четвертой целью введения конструкции класса является поддержка сокрытия информации, переноса обязанностей с клиента на сервер, создание раздельных областей ответственности, устранение необходимости кропотливой координации действий между программистами, работающими над разными частями программы.

Напомним, неполный синтаксис описания класса имеет вид:

```
class <имя класса> : <спецификация наследования>
   {
public: // доступны всем
      <свойства, методы, события>
__published // доступны всем, видны в Инспекторе Объекта и изменяемы
     < свойства, методы, события>
protected: // доступны только потомкам
    <данные, методы, свойства, события>
private: // доступны только в контексте данного класса
    <данные, методы, свойства, события>
```

```
};.
```
Например,
```
class TMyClass : public TParentClass
 {
     private:
          int A;
          double B, C;
     public:
          TMyClass();
          void SetA(int A);
          int GetA(void);
     protected:
          int D(int A);
};
```

Если рассматривать классы C++ как еще одну синтаксическую конструкцию безотносительно к описанным выше целям, то применение классов не улучшит качества программного кода. При проектировании класса необходимо уделять достаточно внимания перечисленным четырем целям и стараться достичь их каждый раз, когда очередной класс включается в программу.

Классы являются основой объектно-ориентированного программирования на C++. Они предоставляют программистам инструменты для создания новых типов данных, более соответствующих поведению реальных объектов, чем типы, реализуемые в программах с функциями. Каждая большая программа C++ получает преимущества от применения классов, если, конечно, эти классы используются корректно и создавались с учетом четырех описанных выше целей. Правильно сконструированная программа C++ представляет собой комбинацию построенных на классах объектов, совместно выполняющих общую задачу, но при этом независимых и раздельно сопровождаемых.

2. Средства раздельного описания интерфейса и реализации объекта. При проектировании класса предполагается разбиение содержания соответствующего классу объекта на доступную из окружающей среды область (интерфейс) и недоступную из окружающей среды область (реализация). На рисунке 8.1 направленные в область интерфейса стрелки символизируют возможность использования интерфейса для воздействия на объект с целью управления его поведением. Поведение объекта осуществляется заложенными в реализацию объекта процедурами. В свою очередь, объект, реализуя свое поведение, может оказывать управляющие воздействия на другие объекты. Эта возможность демонстрируется исходящими стрелками в нижней части рисунка 8.1.

Для наших рассуждений удобно использовать понятия клиентской функции и сервисной функции.

Клиентская функция решает возложенную на нее задачу, обращаясь к помощи доступных объектов, используя их интерфейсы. Сервисная функция предназначена для предоставления объектом — владельцем функции некоторых услуг внешним объектам. Заметим, что такое разделение является каноническим, потому что клиентская функция, принадлежащая некоторому объекту,

может выполняться по инициативе другого объекта и пользоваться при решении своей задачи услугами третьих объектов. Аналогично, сервисная функция, вызванная по инициативе внешнего объекта, может использовать функции других объектов; в этом случае сервисная функция будет одновременно клиентской функцией.

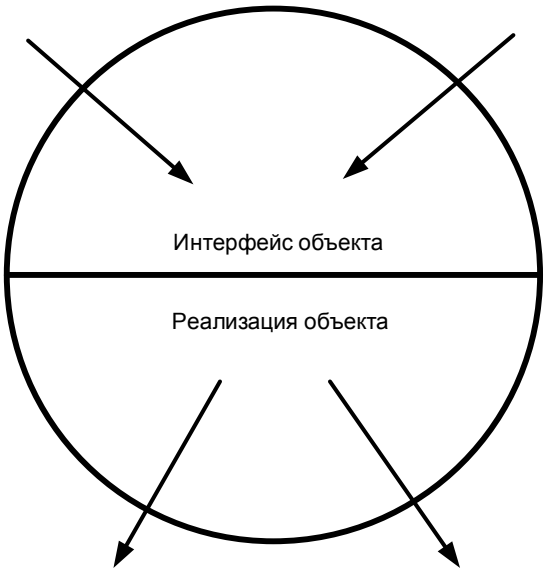


Рис. 8.1
Интерфейс и реализация объекта

3. Средства ограничения видимости элементов описания класса. Степень доступности к объявляемым элементам класса определяется тем, в каком разделе класса они объявлены. Эти средства частично уже рассматривались в главе «Наследование», тем не менее перечислим их кратко здесь. Раздел **public** (открытый) предназначен для объявлений, которые доступны для внешнего использования. В этом отношении открытые элементы класса ничем не отличаются от элементов структуры. Это открытый интерфейс класса. Раздел **published** (публикуемый) содержит открытые свойства, которые появляются в процессе проектирования на странице свойств Инспектора Объектов и значения которых пользователь может устанавливать в процессе проектирования. Раздел **private** (закрытый) содержит объявления полей и функций, используемых только внутри данного класса, т. е. в текущем программном модуле. Обращение к этим элементам возможно только из методов данного класса. Фактически здесь происходит разделение «зоны ответственности» создателя класса и прикладного программиста. Раздел **protected** (защищенный) содержит объявления, доступные только для потомков объявляемого класса. Как и в случае закрытых элементов, детали реализации защищенных элементов скрыты от конечного пользователя. Однако в отличие от закрытых элементов, защищенные элементы остаются доступны для программистов, которые захотят создавать на базе этого

класса производные классы, причем не требуется, чтобы производные классы объявлялись в этом же программном модуле.

Необходимость ограничения доступа к элементам объекта объясняется двумя причинами.

Во-первых, нужно оградить от прикладных программистов то, что им трогать не положено, инструменты, необходимые для внутренней работы типа данных, но не входящие в его интерфейс, используемый клиентом для решения своих задач. На самом деле такие ограничения только помогают прикладным программистам разобраться, что действительно важно для их работы, а от чего лучше держаться подальше.

Во-вторых, ограничение доступа позволяет разработчику класса изменять внутренние механизмы работы объекта, не беспокоясь о том, как это отразится на прикладных программистах. Так, например, при работе со стеком или очередью можно для повышения быстродействия программы перейти к выделению памяти большими блоками вместо выделения памяти при каждом добавлении элемента. Четкое разделение интерфейса и реализации позволяет решить эту задачу, после чего прикладному программисту останется только заново скомпоновать программу.

В любых отношениях важно установить границы, которые должны соблюдаться всеми участниками этих отношений. При создании класса или библиотеки мы должны установить отношения с прикладным программистом, который будет использовать нашу библиотеку для построения приложений или других библиотек.

4. Средства придания модульности программным файлам. Программы на C++ состоят из объявлений (переменных, констант, типов, классов, функций) и описаний реализации функций. Среди функций всегда имеется главная — **main** для консольных приложений или **WinMain** для приложений Windows. Именно с вызова этой главной функции начинается работа программы. Обычно в C++Builder эта функция очень короткая и выполняет только некоторые подготовительные операции, необходимые для инициализации создаваемого программой приложения. Далее работа приложения определяется происходящими событиями и реакцией на них объектов. Подробнее эти механизмы рассмотрены в главе, посвященной обработке событий и сообщений.

Как правило, программы строятся по модульному принципу и состоят из множества модулей. Принцип модульности очень важен для создания надежных и относительно легко модифицируемых и сопровождаемых приложений. Четкое соблюдение принципов модульности в сочетании с принципом скрытия информации позволяет внутри любого модуля проводить какие-то модификации, не затрагивая при этом остальные модули и головную программу.

Согласно принципам скрытия информации, в C++Builder текст модуля разделяется на заголовочный файл интерфейса, который содержит объявления констант, классов и т. п., и файл реализации, в котором содержится описание функций. Как уже говорилось, стандартное расширение программных файлов реализации — **.cpp**, стандартное расширение программных заголовочных файлов — **.h**.

Главная функция приложения размещается в отдельном программном модуле, состоящем только из реализационного файла. Создание модуля с текстом главной функции осуществляется автоматически системой программирования. Хороший стиль программирования предполагает, что этот файл может быть расширен только действиями, связанными с организацией работы приложения в целом. Например, в главный файл могут быть добавлены действия по обработке параметров, передаваемых в приложение при запуске программы. Наилучшим решением будет включение в главную функцию вызова расположенной в другом программном модуле функции обработки параметров приложения с пересылкой ей полученных главной функцией значений параметров.

Все объекты компонентов размещаются в объектах — формах. Для каждой формы, которую мы проектируем в приложении, инструментальная среда C++Builder создает отдельный модуль. Именно в модулях и осуществляется программирование задачи. В обработчиках событий объектов — форм и компонентов, мы размещаем все наши алгоритмы. В основном они сводятся к обработке информации, содержащейся в свойствах одних объектов, и задании по результатам обработки значений свойств других объектов. При этом постоянно происходят вызовы методов различных объектов.

Области видимости элементов программного проекта создаются в системе программных модулей. Элементы описания класса, объявленные в разделе **public** или **published** или **protected**, видимы и доступны для объектов любых классов и для других модулей, в которых директивой **#include** включен заголовочный файл данного модуля. При этом из экземпляров класса к ним можно обращаться непосредственно по имени, а из других объектов и процедур — только со ссылкой на объект данного класса.

Контрольные вопросы

1. Какое назначение конструкции **class** в программе?
2. Почему понадобилась новая конструкция **class**?
3. Чем конструкция **class** отличается от конструкции **struct**?
4. Опишите в общих чертах синтаксис конструкции **class**.
5. На какие части делится структура программного модуля?
6. Какие области видимости определяются при описании класса?
7. Какие имеются средства придания модульности программным файлам?
8. Как создаются области видимости в системе программных файлов?

8.3. Использование программных модулей

Каждое имя в C++ программе должно относиться к уникальной сущности (объекту, функции, типу, шаблону и т. д.). Это не значит, что использование имени для обозначения сущности должно встречаться только один раз в программе: его можно использовать для обозначения другой сущности, если будет обеспечен некоторый контекст, позволяющий различать разные значения одного и того же имени. Таким контекстом служит область видимости. Областью видимости или областью действия имени называется участок кода, внутри которого с именем можно работать бесконфликтно. Блочная структура програм-

мы вносит особенности в построении областей видимости имен. Для изучения областей видимости используются понятия определяющего и использующего вхождения имен.

Определяющее вхождение имени соответствует вхождению имени в его определение.

Использующее вхождение имени — это вхождение имени в оператор или в выражение.

Алгоритмом идентификации называется процедура сопоставления использующему вхождению имени соответствующего определяющего вхождения имени.

Область видимости имени начинается в точке объявления имени и распространяется на текущий блок, в котором множество использующих вхождений одного и того же имени соответствует единственному определяющему вхождению имени. При выполнении определенных условий область видимости имени может распространяться на текущий модуль, пространство имен, другие модули.

В C++ поддерживается три разновидности областей видимости имен: *локальная* область видимости, область видимости *пространства имен* и область видимости *класса*.

Локальная область видимости — это часть исходного текста программы, образующая блок. Объявляющие вхождения имени находятся внутри блока. Использующие вхождения имени находятся внутри блока, включая вложенные блоки.

Область видимости пространства имен образует некоторую область программы, в которой отсутствуют конфликты объявления имен: в пространстве имен каждое имя объявлено только один раз, в разных пространствах имен одинаковые имена представляют собой разные сущности. Считается, что имеется глобальное пространство имен, внутри которого программисту разрешено задавать пользовательские пространства имен. Каждое такое пространство является отдельной областью видимости. Пользовательское пространство имен, как и глобальное, может содержать объявления и определения объектов, типов, шаблонов, а также вложенные пользовательские пространства имен.

Область видимости класса образует каждое определение класса.

8.3.1. От исходного кода к исполняемому модулю

Создание исполняемого файла издавна производится в три этапа: 1) обработка каждого программного модуля препроцессором, 2) компиляция каждого программного модуля в объектный код и 3) компоновка объектных модулей, включая модули из объектных библиотек, в исполняемый файл. Это классическая схема для компилируемых языков.

Часто *компиляцией* программы называют весь процесс преобразования исходного кода в исполняемый модуль. В системе программирования RAD Studio этот процесс называется *построение* (build) проекта, и за ним скрываются рассмотренные три отдельных этапа создания исполняемого модуля. Эти этапы проявляют себя в тех случаях, когда на этапе препроцессинга, компиляции или

компоновки обнаруживаются ошибки. Поэтому кратко рассмотрим содержание каждого этапа.

Пусть у нас имеется знаменитая программа C++ «Hello, World!»:

```
#include <iostream>
int main()
{
   std::cout << «Hello, World!\n»;
}
```

Обработка программы препроцессором происходит перед ее компиляцией. Ориентируясь на этот этап предварительной обработки, мы можем выполнить следующие действия: включить в компилируемый файл другие файлы, определить *символические константы* и *макросы*, задать режим *условной компиляции* программного кода и *условного выполнения директив препроцессора*. Все директивы препроцессора начинаются с символа «#» и до начала директивы в строке могут находиться только символы пробела. Любая строка, начинающаяся с символа «#», воспринимается как директива препроцессора.

Сначала исходный код обрабатывается препроцессором. В нашем примере препроцессор находит директиву **#include <iostream>**, выполняет поиск заголовочного файла **iostream** и заменяет директиву текстом из этого файла, попутно обрабатывая все директивы препроцессора во включаемом тексте. Подчеркнем: файл, указанный в директиве **#include**, является *заголовочным файлом*.

После текстуального включения заголовочного файла в текст программы (или модуля) становится возможным использование в тексте программы всего того, что описано в этом заголовочном файле.

Затем результат работы препроцессора передается компилятору. Компилятор производит весь положенный комплекс работ: от синтаксического разбора и поиска ошибок до создания объектного файла (понятно, что если имеются синтаксические ошибки, то объектный файл не создается). Объектный файл создается для каждого программного модуля. В объектном файле обычно имеется *таблица внешних ссылок* — некая таблица, в которой, в частности, перечислены имена подпрограмм, которые используются в объектном модуле, но код которых находится в других объектных модулях. Эти подпрограммы являются *внешними* по отношению к модулю.

Чтобы получить нормальный исполняемый модуль, необходимо «разрешить» внешние ссылки, т. е. добавить в исполняемый модуль код отсутствующих подпрограмм и настроить соответствующим образом все ссылки на этот код. Этим занимается компоновщик. Он анализирует таблицу внешних ссылок объектного модуля, ищет в объектных библиотеках недостающие модули, копирует их в исполняемый модуль и настраивает ссылки. После этого исполняемый модуль готов к исполнению.

Библиотека (объектная библиотека) — это набор откомпилированных подпрограмм, собранных в единый файл определенной структуры. Подключение библиотеки происходит на этапе компоновки исполняемого файла из объ-

ектных файлов (т. е. из тех файлов, которые получаются в результате компиляции исходного текста программы).

Обширные объектные библиотеки входят в комплект поставки компилятора[29]. В комплект поставки библиотек входит набор заголовочных файлов, которые содержат объявления, необходимые компилятору.

Если исходный код программы разделен на несколько файлов, то процесс компиляции и сборки происходит аналогично. Сначала все единицы компиляции по отдельности компилируются, а затем компоновщик собирает полученные объектные модули (с подключением библиотек) в исполняемый файл.

Контрольные вопросы

1. Какие имеются этапы создания исполняемого модуля?

8.3.2. Разделение текста программы на модули

Разделение исходного текста программы на программные модули становится необходимым по многим причинам:

1. С большим текстом просто неудобно работать.
2. Необходимости декомпозиции сложной задачи на подзадачи. Конкретную подзадачу удобно разместить в отдельном программном модуле.
3. Разделить программу на отдельные модули с целью повторного использования этих модулей в других программах.
4. При коллективной разработке программ удобно частные результаты размещать в отдельных программных модулях с последующей автономной отладкой.
5. Разделение интерфейса и реализации с целью защиты модуля от несанкционированного использования данных и операций.

Как только мы решаем разделить исходный текст программы на несколько модулей, возникают три задачи:

1. Необходимо от простой компиляции программы перейти к построению проекта (препроцессор, компиляция, компоновка). Для этого надо внести соответствующие изменения в проект.
2. Необходимо решить, каким образом разбить текст программы на отдельные файлы.
3. Необходимо упорядочить содержание каждого программного модуля.

Первая задача чисто техническая. Она решается чтением руководств по использованию среды разработки (IDE). Поэтому на решении этой проблемы мы останавливаться не будем.

Вторая задача требует гораздо более творческого подхода. Здесь существуют определенные рекомендации, несоблюдение которых приводит либо к невозможности собрать проект, либо к трудностям в дальнейшем развитии проекта. Разбиение любого проекта происходит на подсистемы, пакеты, классы и т. д. Каждая такая часть оказывается менее сложной, чем решаемая программной системой задача, но в совокупности они составляют целостное решение. Первостепенное значение имеет качество такого разбиения.

[29] Рассмотрению библиотеки визуальных компонентов RAD Studio C++ Builder посвящена глава 12.

Основными качествами модульности являются *связность* и *сцепление*. Мы должны стремиться к тому, чтобы у модулей были: **сильная связность и слабое сцепление.**

Связность — это оценка того, в какой мере близкие функции, собранные вместе, и насколько хорошо части модуля действуют в совокупности. Связность — это то, что обеспечивает единство модуля. Слабая связность модулей — признак неудачной декомпозиции. Каждый модуль должен иметь четко определенную задачу, а не выступать как мешок, в который сваливают не связанные друг с другом классы.

Сцепление — это показатель взаимозависимости *между* модулями: количество входов и выходов из них. В простейших случаях модули мало связаны друг с другом и потому менее зависят один от другого. Очевидно, что нельзя полностью развязать модули, иначе они вообще не смогут взаимодействовать!

Модули соединяются между собой различными способами — прямыми или косвенными. Модуль может вызывать методы классов из других модулей или сам являться объектом подобного обращения из других модулей. Он может использовать типы данных, определенные в другом модуле, или выделять какие-то данные для общего доступа (например, файлы). В хорошем проекте каналы связи создаются только там, где без них совершенно невозможно обойтись. Эти каналы связи во многом определяют конструкцию программного кода.

После того как модуль определен, его можно разрабатывать и тестировать независимо от других. Это преимущество модульности; оно позволяет распределить задание между несколькими программистами. Однако следует проявлять осторожность. Так называемый закон Конвэя предупреждает, что структура программного продукта может отражать структуру команды разработчиков: «Если в разработке компилятора участвуют четыре команды, у нас получится четырехпроходный компилятор»[30]. Необходимо следить за тем, чтобы декомпозиция была разумной и соответствовала решаемой проблеме, а не штатной структуре.

Таким образом, мы приходим к правилу: *проектируем такие модули, которые внутренне связны и минимально соединены между собой. Декомпозиция должна отражать правильное разбиение задачи на части.*

Третья задача разбиения на модули связана с упорядочиванием структуры программного модуля. Мы снова возвращаемся к понятиям: интерфейс и реализация. Подробно эта тема будет рассмотрена в п. 8.3.3 «Интерфейс и реализация».

8.3.2.1. Включение программных модулей

Директива #include включает в текст программы содержимое указанного файла. Эта директива имеет две формы:

#include «имя_файла»
#include < имя_файла >.

[30] В четырехпроходном компиляторе исходный текст обрабатываться за четыре отдельных просмотра текста от начала до конца.

Имя файла должно соответствовать соглашениям операционной системы и может состоять либо только из имени файла, либо из имени файла с предшествующим ему маршрутом. Если имя файла указано в кавычках, то поиск файла осуществляется в соответствии с заданным маршрутом, а при его отсутствии — в текущем каталоге. Если имя файла задано в угловых скобках, то поиск файла производится в стандартных директориях операционной системы, задаваемых командой PATH.

Директива #include может быть вложенной, т. е. во включаемом файле тоже может содержаться директива #include, которая замещается после включения файла, содержащего эту директиву.

Директива #include широко используется для включения в программу так называемых заголовочных файлов, содержащих прототипы библиотечных функций, и поэтому большинство программ на C++ начинаются с этой директивы.

8.3.2.2. Использование пространств имен

Определение пространства имен имеет вид:

```
namespace nsName
{
   class TMyClass { };
    ...
}
```

где **namespace** — ключевое слово, **nsName** — вводимое имя пространства имен, блок {...} — тело пространства имен.

Внутри тела пространства имен обращение к идентификаторам происходит по имени идентификатора. Но вне тела пространства имен требуется указание имени пространства имен перед идентификатором. Например, обращение к имени класса **TMyClass** в рассмотренном пространстве имен **nsName** необходимо записать **nsName*::*TMyClass**. C++ содержит некоторые другие конструкции, делающие подобные требования необязательными. Так, при добавлении строки

```
using namespace nsName;
```

указание префикса **nsName::** в дальнейшем не требуется.

Операционные системы и многие современные языки программирования обеспечивают поддержку своей модели пространств имен: используют каталоги (или папки) как модель пространства имен. Это позволяет существовать двум файлам с одинаковыми именами (пока они находятся в разных каталогах). В языке программирования C++ идентификаторы имен пространств сами ассоциированы с соответствующими пространствами. Поэтому в этих языках пространства имен могут вкладываться друг в друга, формируя дерево пространств имен. Корень такого дерева называется *глобальным пространством имен*.

Рассмотрим пример:

```
namespace MySpace
{
   int x=0;
}
```

Теперь для доступа к переменной **x** в другом контексте необходимо сделать пространство имен видимым:

```
void fun1()
{
 using namespace MySpace;
 x++;
}
```

Имеется другой прием для обеспечения доступа к элементам пространства имен. Для нашего примера возможна прямая ссылка на переменную из существующего пространства имен:

```
void fun2()
{
 MySpace::x=1;
}
```

Разрешение пространств имен в C++ иерархично. Это означает, что мы можем неоднократно применять для доступа к элементу пространства имен операцию **::**. Например, **A::B::C** означает, что некоторый элемент (пространство имен) **C** находится в пространстве имен **B**, которое, в свою очередь, входит в пространство имен **A**. Разработчики библиотеки визуальных компонентов RAD Studio интенсивно используют пространства имен. Для иллюстрации рассмотрим несколько фрагментов кода из модулей библиотеки визуальных компонентов. Приведем только такие строки, которые являются существенными для нашего примера. Начнем с программного модуля **System.UITypes.hpp**:

```
namespace System
{
    namespace Uitypes
    {
       enum class DECLSPEC_DENUM TWindowState :
                          unsigned char { wsNormal, wsMinimized, wsMaximized };.
    }
}
```

Как видим из приведенного текста, в пространстве имен **System::UITypes**, среди прочего, объявлен перечислимый тип **TWindowState**.

Заглянем теперь в программный модуль **Vcl.Forms.hpp**:

```
#include <System.UITypes.hpp>
namespace Vcl
{
    namespace Forms
    {
        using System::Uitypes::TWindowState;
    }
}
namespace Vcl
{
```

```
    namespace Forms
    {
        using System::Uitypes::TBorderIcon;
        using System::Uitypes::TBorderIcons;
    }
}
```

Во-первых, в приведенном фрагменте с помощью операции **#include** устанавливается доступ к программному модулю **System.UITypes.hpp**. Во-вторых, в пространстве имен **Vcl::Forms** размещается объявление

using System::Uitypes::TWindowState,

посредством которого становится видимым тип данных **TWindowState**, введенный в пространстве имен **System::UITypes** программного *модуля* **System.UITypes.hpp**. Теперь тип данных **TWindowState** может свободно использоваться внутри пространства имен **Vcl::Forms** программного модуля **Vcl.Forms.hpp**. Аналогично устанавливается доступ к типам данных **TBorderIcon** и **TBorderIcons**, объявления которых в нашем примере не показаны, но они имеются в пространстве имен **System::Uitypes** программного модуля **System.UITypes.hpp**.

Для использования приведенного определения типа данных **TWindowState** в своем программном модуле можно записать

#include <System.UITypes.hpp>

либо

#include <Vcl.Forms.hpp>

Затем записываются действия по использованию типа данных **TWindowState**

```
namespace Vcl
{
    namespace Forms
    {
    TWindowState x=wsNormal;
    ...
    }
}
```

Как показал последний пример, мы можем создать условия для ограниченного доступа к содержимому программного модуля **System.UITypes.hpp**, что является одним из приемов инкапсуляции.

8.3.2.3. Взаимодействие программных модулей и пространств имен

Рассмотренный пример помогает нам понять методику образования физической (программные модули) и логической (пространства имен) структур программных модулей. Обе структуры библиотеки визуальных компонентов в основном строятся параллельно. Физическая структура модулей дополняется логической структурой пространств имен. Приведем некоторые, вытекающие из наших рассуждений, примеры. Показанный в нашем примере

программный модуль **Vcl.Forms.hpp** включает размещенные в пространство имен **Vcl::Forms** определения, связанные с функционированием экранной формы.

В программном модуле **System.UITypes.hpp** введено пространство имен **Uitypes**, которое является внутренним по отношению к пространству имен **System**, а определения пространства имен **System** находятся в модуле **sysclass.h**. Кроме того, имеется пространство имен **Vcl**, которое является объединяющим для многих других пространств имен.

Такое взаимодействие физической и логической структур программных модулей создает дополнительные возможности для инкапсуляции отдельных фрагментов программы. Проиллюстрируем эту мысль примером.

Пусть имеется программный модуль **Unit1**[31]:

```
int z;
 namespace nsX
{
        int x;
        namespace nsY
        {
         int y;
        }
}
```

В модуле **Unit1** в глобальном пространстве имен введена переменная **z**, в пространстве имен **nsX** объявлена переменная **x**, а во вложенном пространстве имен **nsY** находится описание переменной **y**.

Пусть имеется программный модуль **Unit2** следующего содержания:

```
#include «Unit1.h»
using nsX::x;
using nsX::nsY::y;
namespace nsC
 {
    class TClass1
   {
       public:
         int funx() {return x; };
         int funy() {return y; };
         int funz() {return z; };
   };
 }
```

Использование переменной **z** гарантируется директивой препроцессора **#include «Unit1.h»**. Однако для переменных **x** и **y** этого недостаточно, необхо-

[31] В параграфе 8.5 рассматривается правило: объектно-ориентированная программа должна содержать только классы и объекты. В рассматриваемых примерах мы нарушаем указанное правило. Основная причина нарушения данного правила заключается в стремлении компактно проиллюстрировать рассматриваемые приемы.

димы дополнительные описания доступа к пространствам имен с помощью операции **using**.

Наконец, рассмотрим некоторый клиентский, по отношению к уже приведенным, модуль, содержащий следующий код:

```
#include «Unit2.h»
        nsC::TClass1 * Obj1=new nsC::TClass1();
        x=5;
        y=6;
        z=7;
        Edit1–>Text=IntToStr(Obj1–>funx());
        Edit2–>Text=IntToStr(Obj1–>funy());
        Edit3–>Text=IntToStr(Obj1–>funz());.
```

В результате обработки препроцессором рассмотренных модулей происходят две подстановки[32]:

1. Вместо вхождения директивы **#include «Unit2.h»** в головной модуль подставляется модуль **Unit2.h**;

2. Вместо вхождения директивы **#include «Unit1.h»** в модуль **Unit2.h** подставляется модуль **Unit1.h**.

После выполненных подстановок исходный текст программы будет иметь вид:

```
int z;
 namespace nsX
 {
          int x;
          namespace nsY
          {
           int y;
          }
};
using nsX::x;
using nsX::nsY::y;
namespace nsC
 {
     class TClass1
    {
        public:
          int funx() {return x; };
          int funy() {return y; };
          int funz() {return z; };
    };
 };
        nsC::TClass1 * Obj1=new nsC::TClass1();
        x=5;
```

[32] В приведенных примерах мы рассматриваем результаты действий препроцессора. Реально результаты рассмотренных подстановок в исходном тексте программы не отображаются.

```
        y=6;
        z=7;
        Edit1–>Text=IntToStr(Obj1–>funx());
        Edit2–>Text=IntToStr(Obj1–>funy());
        Edit3–>Text=IntToStr(Obj1–>funz());.
```

Далее обрабатываются директивы **using**, с помощью которых в программе формируются составные имена. В результате получим окончательный текст:

```
int z;
 namespace nsX
 {
          int x;
          namespace nsY
          {
           int y;
          }
};
namespace nsC
 {
     class TClass1
    {
        public:
          int funx() {return nsX::x; };
          int funy() {return nsX::nsY::y; };
          int funz() {return z; };
    };
 };
        nsC::TClass1 * Obj1=new nsC::TClass1();
        nsX::x=5;
        nsX::nsY::y=6;
        z=7;
        Edit1–>Text=IntToStr(Obj1–>funx());
        Edit2–>Text=IntToStr(Obj1–>funy());
        Edit3–>Text=IntToStr(Obj1–>funz());.
```

Теперь для каждого использующего вхождения переменной и типа данных программы определен доступ к определяющему вхождению.

Замечание. Приведенный пример показывает логику формирования текста модульной программы. Физически отдельный программный модуль может использовать несколькими программными модулями программы. В этом случае текст каждого программного модуля появляется в окончательном тексте программы только один раз, установление реальных связей определяется отдельными моделями компиляции.

Проведем эксперимент с рассмотренным примером программы. Результаты эксперимента приведены в таблице 8.1.

Таблица 8.1

Результаты эксперимента использования программных модулей и пространств имен

Модуль Unit1.h	Модуль Unit2.h	Головной модуль	Сообщение транслятора	Комментарий
int z; **namespace nsX** **{** **int x;** **namespace nsY** **{** **int y;** **}** **}**	**#include «Unit1.h»** **using nsX::x;** **using sX::nsY::y;** **namespace nsC** **{** **class TClass1** **{** **public:** **int funx()** **{return x; };** **int funy()** **{return y; };** **int funz()** **{return z; };** **};** **}**	**#include «Unit2.h»** **x=5;** **y=6;** **z=7;** **nsC::TClass1 * Obj1;** **Obj1 =** **new nsC::TClass1();** **Edit1–>Text=** **IntToStr(Obj1–>funx());** **Edit2–>Text=** **IntToStr(Obj1–>funy());** **Edit3–>Text=** **IntToStr(Obj1–>funz());**	Текст верный	Доступ к переменным **x** и **y** осуществляется благодаря директивам: **#include «Unit2.h»** головного модуля, **#include «Unit1.h»** **using nsX::x;** **using sX::nsY::y;** модуля **Unit2.h**
	#include «Unit1.h» **//using nsX::x;** **//using sX::nsY::y;** **namespace nsC** **{** **class TClass1** **{** **public:** **int funx()** **{return x; };** **int funy()** **{return y; };** **int funz()** **{return z; };** **};**	**#include «Unit2.h»** **x=5;** **y=6;** **z=7;** **nsC::TClass1 * Obj1;** **Obj1 =** **new nsC::TClass1();** **Edit1–>Text=** **IntToStr(Obj1–>funx());** **Edit2–>Text=** **IntToStr(Obj1–>funy());** **Edit3–>Text=** **IntToStr(Obj1–>funz());**	Unit2:Undefined symbol x Unit2:Undefined symbol y Unit-Main:Undefined symbol x Unit-Main:Undefined symbol y	Доступ к переменным **x** и **y** перекрыт в модуле **Unit2.h** Переменная **z** доступна

Продолжение табл. 8.1

Модуль Unit1.h	Модуль Unit2.h	Головной модуль	Сообщение транслятора	Комментарий

	#include «Unit1.h»
	//using nsX::x;
	//using sX::nsY::y;
	namespace nsC
	{
	class TClass1
	{
	public:
	int funx()
	{return x; };
	int funy()
	{return y; };
	int funz()
	{return z; };
	};
		#include «Unit1.h»
		#include «Unit2.h»
		x=5;
		y=6;
		z=7;
		nsC::TClass1 * Obj1;
		Obj1 =
		new nsC::TClass1();
		Edit1–>Text=
		IntToStr(Obj1–>funx());
		Edit2–>Text=
		IntToStr(Obj1–>funy());
		Edit3–>Text=
		IntToStr(Obj1–>funz());
			Unit1.h: Multiply declaration for x
			Unit1.h: Multiply declaration for y
			Unit1.h: Multiply declaration for z
				Множественное определение переменных: модуль **Unit1.h** вставлен два раза в исходный текст по директиве **#include «Unit1.h»** модуля **Unit2.h** и директивам **#include «Unit2.h»** головного модуля, **#include «Unit1.h»** модуля **Unit2.h**

namespace nsGen
{
int z;
namespace nsX
{
int x;
namespace nsY
{
int y;
}
}
}
	namespace nsGen
	{
	namespace nsX
	{
	namespace nsY
	{
	namespace nsC
	{
	class TClass1
	{
	public:
	int funx() {return x;
	};
	int funy() {return y;
	};
	int funz() {return z;
	};
	};
	}
	}
	}
	}
		using namespace nsGen;
		using namespace nsGen::nsX;
		using namespace nsGen::nsX::nsY;
		using namespace nsGen::nsX::nsY::nsC;
		x=5;
		y=6;
		z=7;
		TClass1 * Obj1=new TClass1();
		Edit1–>Text=IntToStr(Obj1–>funx());
		Edit2–>Text=IntToStr(Obj1–>funy());
		Edit3–>Text=IntToStr(Obj1–>funz());
			Текст верный	В модулях **Unit1.h**, **Unit2.h** используется пространство имен **nsGen**. Хотя полученный код является верным, структура модуля **Unit2.h** получилась трудночитаемой

Контрольные вопросы

1. Какие имеются причины разделения исходного текста программы на модули?
2. Какие возникают задачи при разделении исходного текста программы на модули?
3. Расшифруйте понятие «связность модуля».
4. Расшифруйте понятие «сцепление модулей».
5. Какое существует правило достижения качества модульности?
6. Какие директивы используются для включения программного модуля в проект (программу)?
7. Какие имеются правила использования пространств имен?
8. Какая существует причина появления сообщения об ошибке «Множественное определение имени», являющаяся следствием неправильного использования директив взаимодействия программных модулей и пространств имен?

8.3.3. Интерфейс и реализация

Разделение описания интерфейса и реализации объекта обеспечивается структурой модуля C++Builder. Каждый программный модуль делится на две части.

Первая часть программного модуля имеет расширение **.h** (иногда **.hpp**) и называется заголовочным файлом (header file). В C++ заголовочный файл — это основной способ подключить к программе типы данных, структуры, прототипы функций, перечислимые типы, используемые в другом модуле. В заголовочном файле рекомендуется располагать интерфейсную часть описания класса.

Когда часть программы выделяется в модуль, остальной части программы надо каким-то образом объяснить, что имеется в этом модуле. Для этого и служат заголовочные файлы.

Таким образом, модуль состоит из двух файлов: заголовочного (интерфейс) и файла реализации.

Файл реализации имеет расширение **.cpp** для программ на C++. Объявления, которые не должны быть видны снаружи, помещаются в файл реализации.

Рассмотрим правила, которых следует придерживаться при формировании заголовочного и реализационного файлов.

Правило 1. Заголовочный файл может содержать только объявления полей, свойств и методов. Заголовочный файл не должен содержать описания реализации методов.

Как следствие, при обработке содержимого заголовочного файла компилятор не должен генерировать информацию для объектного модуля.

Правило 2. Заголовочный файл должен иметь механизм защиты от повторного включения.

Защита от повторного включения реализуется директивами препроцессора:

```
#ifndef <identifier>
#define < identifier >
```

//набор объявлений

#endif.

Для препроцессора при первом включении заголовочного файла это выглядит так: поскольку условие «identifier не определен» **(#ifndef** <identifier>) истинно, определить identifier **(#define** < identifier >) и обработать все строки до директивы **#endif**. При повторном включении так: поскольку условие «identifier не определен» **(#ifndef** <identifier>) ложно (символ был определен при первом включении), то пропустить все до директивы **#endif**.

В качестве <identifier> обычно применяют имя самого заголовочного файла с добавлением символа H. Например, для файла **ufrmStopWatch.h** запишем **#define ufrmStopWatchH**. В качестве альтернативного способа может применяться директива **#pragma once**. Однако преимущество первого способа в том, что он работает на любых компиляторах.

Заголовочный файл сам по себе не является единицей компиляции, в единицу компиляции входит одновременно и заголовочный, и соответствующий ему реализационный файл.

Правило 3. В файле реализации должна быть директива включения соответствующего заголовочного файла.

Понятно, что объявления, которые видны снаружи модуля, должны быть также доступны и внутри.

Правило также гарантирует соответствие между описанием и реализацией. При несовпадении, допустим, сигнатуры функции в объявлении и вызова функции компилятор выдаст ошибку.

Правило 4. В файле реализации не должно быть объявлений, дублирующих объявления в соответствующем заголовочном файле.

При выполнении правила 3 нарушение правила 4 приведет к ошибкам компиляции.

Рассмотрим пример. Допустим, у нас имеется следующая программа:

```
const int intCGlobal=10;      // глобальная константа
extern int intVGlobal;          // глобальная переменная
int intVFuncsGlobal=0;       // глобальная переменная для func1 и func2
int func1()
{
        ++intVGlobal;
        return ++intVFuncsGlobal;
}
int func2()
{
        ++intVGlobal;
        return --intVFuncsGlobal;
}
class TMyClass
{
      private:
          int intFPrivate;
```

```
        static int intFStaticPrivate;
    public:
        TMyClass();
        ~TMyClass();
        void ChangePrivate(int arg);
        int GetFPrivate()const;
        int GetFStaticPrivate() const;
};
int TMyClass::intFStaticPrivate=0;
TMyClass::TMyClass() : intFPrivate(intCGlobal)
{
        ++intFStaticPrivate;
}
TMyClass::~TMyClass()
{
        --intFStaticPrivate;
}
void TMyClass::ChangePrivate(int arg)
{
        intFPrivate += arg;
}
int TMyClass::GetFPrivate()const
{
        return intFPrivate;
}
int TMyClass::GetFStaticPrivate() const
{
        return intFStaticPrivate;
}
int MainFun()
{
        int intVBalance;
        intVBalance=func1();
        intVBalance=func2();
        ShowMessage(«intVBalance=«+ IntToStr(intVBalance)
                        +» intVGlobal=«+IntToStr(intVGlobal));
        TMyClass obj1, obj2;
        if (obj1.GetFPrivate() == intCGlobal) ShowMessage(«GetFPrivate=intCGlobal»);
        ShowMessage(«intFStaticPrivate=«+IntToStr(obj2.GetFStaticPrivate()));
        return 0;
}.
```

Эта программа не является образцом для подражания, поскольку некоторые моменты идеологически неправильны (выявить самостоятельно), но для демонстрации наших рассуждений эта программа подходит очень неплохо.

Итак, перечислим по порядку, что у нас имеется?

1. Глобальная константа **intCGlobal**, которая используется и в классе, и в функции **MainFun**.

2. Глобальная переменная **intVGlobal**, которая используется в функциях **func1**, **func2** и **MainFun**.

3. Глобальная переменная **intVFuncsGlobal**, которая используется только в функциях **func1** и **func2**.

4. Функции **func1** и **func2**.

5. Класс **TMyClass**.

6. Функция **MainFun**.

Вырисовываются три единицы компиляци.

1. Функция **main.**

2. Класс **TMyClass**.

3. Функции **func1** и **func2** с глобальной переменной **intVFuncsGlobal**, которая используется только в них.

ЛАНЬ®

Не совсем понятно, что делать с глобальной константой **intCGlobal** и глобальной переменной **intVGlobal**. Первая тяготеет к классу **TMyClass**, вторая — к функциям **func1** и **func2**. Однако предположим, что планируется и эту константу, и эту переменную использовать еще в каких-то, пока не написанных, модулях программы. Поэтому прибавится еще одна единица компиляции.

Теперь пробуем разделить программу на модули.

Сначала, как наиболее связанные сущности (используются во многих местах программы), выносим глобальную константу **intCGlobal** и глобальную переменную **intVGlobal** в отдельную единицу компиляции.

```
// модуль uGlobals.h
#ifndef uGlobalsH
#define uGlobalsH
const int intCGlobal=10;          // глобальная константа
extern int intVGlobal;            // глобальная переменная
#endif // uGlobalsH
// модуль uGlobals.cpp
#include «uGlobals.h»
int intVGlobal=0;                 // глобальная переменная.
```

Обратим внимание на то, что глобальная переменная в заголовочном файле имеет спецификатор **extern**. При этом получается *объявление* переменной, а не ее *определение*. Такое описание означает, что где-то существует переменная с таким именем и указанным типом. А определение этой переменной (с инициализацией) помещено в файл реализации. Константа описана в заголовочном файле.

С объявлением констант в заголовочном файле существует одна тонкость. Если константа тривиального типа, то ее можно объявить в заголовочном файле. В противном случае она должна быть определена в файле реализации, а в заголовочном файле должно быть ее объявление (аналогично, как для переменной).

ЛАНЬ®

Также обратим внимание, во-первых, на защиту от повторного включения заголовочного файла и, во-вторых, на включение заголовочного файла в файле реализации.

Затем выносим в отдельный модуль функции **func1** и **func2** с глобальной переменной **intVFuncsGlobal**. Получаем еще два файла:

```
// модуль uFuncs.h
#ifndef uFuncsH
#define uFuncsH
int func1();
int func2();
#endif   // uFuncsH
// модуль uFuncs.cpp
#include «uFuncs.h»
#include «uGlobals.h»
int intVFuncsGlobal=0;        // глобальная переменная для func1 и func2
int func1()
{
   ++intVGlobal;
          return ++intVFuncsGlobal;
}
int func2()
{
          ++intVGlobal;
   return --intVFuncsGlobal;
}.
```

Поскольку переменная **intVFuncsGlobal** используется только этими двумя функциями, ее объявление в заголовочном файле отсутствует. Из этого модуля «на экспорт» идут только две функции.

В функциях используется переменная из другого модуля, поэтому необходимо добавить **#include « uGlobals.h »**.

Наконец выносим в отдельный модуль класс **TMyClass**:

```
// модуль uClass.h

#ifndef uClassH
#define uClassH
class TMyClass
{
    private:
            int intFPrivate;
            static int intFStaticPrivate;
    public:
            TMyClass();
           ~TMyClass();
            void ChangePrivate(int arg);
            int GetFPrivate()const;
            int GetFStaticPrivate() const;
};
#endif  // uClassH

// модуль uClass.cpp

#include «uClass.h»
```

```
#include «uGlobals.h»
int TMyClass::intFStaticPrivate=0;
TMyClass::TMyClass() : intFPrivate(intCGlobal)
{
    ++intFStaticPrivate;
}

TMyClass::~TMyClass()
{
    --intFStaticPrivate;
}
void TMyClass::ChangePrivate(int arg)
{
    intFPrivate += arg;
}
int TMyClass::GetFPrivate() const
{
   return intFPrivate;
}
int TMyClass::GetFStaticPrivate() const
{
   return intFStaticPrivate;
}
```

Обратим внимание на следующие моменты:

1) из объявления класса убрали определения тел функций (методов). Это сделано по идеологическим причинам: интерфейс и реализация должны быть разделены для возможности последующих изменений реализации без изменения интерфейса;

2) класс имеет статическое поле, поэтому для всех экземпляров класса эта переменная будет общей. Ее инициализация выполняется не в конструкторе, а в глобальной области модуля;

3) в файл реализации добавлена директива **#include «uGlobals.h»** для доступа к константе **intCGlobal**.

Классы практически всегда выделяются в отдельные единицы компиляции.

В файл **uMain.h** включаем объявление функции **MainFun**:

```
#ifndef uMainH
#define uMainH
int MainFun();
#endif,
```

а в файле **uMain.cpp** оставляем определение функции **MainFun** и добавляем необходимые директивы включения заголовочных файлов.

```
// модуль uMain.cpp

#ifndef uMainH
#define uMainH
#include «uFuncs.h»
```

```
#include «uClass.h»
#include «uGlobals.h»
#include «vcl.h»
int MainFun ()
{
    int intVBalance;
    intVBalance=func1();
    intVBalance=func2();
    ShowMessage(«intVBalance=«+ IntToStr(intVBalance)+
                                « intVGlobal=«+IntToStr(intVGlobal));
    TMyClass obj1, obj2;
    if (obj1.GetFPrivate() == intCGlobal)
                                    ShowMessage(«GetFPrivate=intCGlobal»);
    ShowMessage(«intFStaticPrivate=«+IntToStr(obj2.GetFStaticPrivate()));
    return 0;
}
#endif    // uMain.cpp.
```

Наконец, при создании приложения с формой в среде RAD Studio у нас появится еще один модуль, из которого происходит вызов функции **MainFun**:

```
void __fastcall TForm2::Button1Click(TObject *Sender)
{
  MainFun();
}
```

Типичные ошибки построения файловой системы. Рассмотрим некоторые ошибки, которые могут негативно влиять на качество создаваемой программы.

Ошибка 1. Определение в заголовочном файле. Эта ошибка в некоторых случаях может себя не проявлять. Например, когда заголовочный файл с этой ошибкой включается только один раз. Но как только этот заголовочный файл будет включен более одного раза, получим либо ошибку компиляции «*многократное определение символа ...»*, либо ошибку компоновщика аналогичного содержания, если второе включение было сделано в другой единице компиляции.

Ошибка 2. Отсутствие защиты от повторного включения заголовочного файла. Тоже проявляет себя при определенных обстоятельствах. Может вызывать ошибку компиляции *«многократное определение символа ...»*.

Ошибка 3. Несовпадение объявления в заголовочном файле и определения в файле реализации. Обычно возникает в процессе редактирования исходного кода, когда в файл реализации вносятся изменения, а про заголовочный файл забывают.

Ошибка 4. Отсутствие необходимой директивы **#include**. Если необходимый заголовочный файл не включен, то все сущности, которые в нем объявлены, останутся неизвестными компилятору. Вызывает ошибку компиляции *«не определен символ ...»*.

Ошибка 5. Отсутствие необходимого модуля в проекте построения программы. Вызывает ошибку компоновки «*не определен символ ...»*. Обратите

внимание, что имя символа в сообщении компоновщика почти всегда отличается от того, которое определено в программе: оно дополнено другими буквами, цифрами или знаками.

Ошибка 6. Зависимость от порядка включения заголовочных файлов. Не совсем ошибка, но таких ситуаций следует избегать. Обычно сигнализирует либо об ошибках в проектировании программы, либо об ошибках при разделении исходного кода на модули.

В рамках ограниченного объема невозможно рассмотреть все случаи, возникающие при раздельной компиляции. Бывают ситуации, когда разделение программы или большого модуля на более мелкие модули кажется невозможным. Обычно это бывает, когда программа плохо спроектирована (части кода имеют сильные взаимные связи). Конечно, можно приложить дополнительные усилия и все-таки разделить код на модули (или оставить как есть), но лучше их (усилия) потратить более эффективно: на изменение структуры программы. Это принесет в дальнейшем гораздо больше пользы, чем просто силовое решение.

Контрольные вопросы

1. Какова структура программного модуля в C++?
2. Какие имеются правила разделения модуля на две части?
3. Какие типичные ошибки возникают при построении файловой системы приложения?

8.4. Использование полей и свойств с разными областями видимости

Важным инструментом реализации инкапсуляции в проектах являются свойства. Понятие свойства рассматривалось в главе 6. Здесь мы рассмотрим типичный прием создания и использования свойств. Пусть имеется описание класса **TAnyClass**:

```
class TAnyClass
{
    private:
      int intFValue;
      bool Correct(int intAValue);
      void SetValue(int intANewValue);
    public:
      __property int intPValue={read=intFValue, write=SetValue};
};
```

В классе **TAnyClass** в разделе **public** находится свойство **intPValue**, доступ к которому не ограничен. В опции задано поле **intFValue**. Само поле **intFValue** имеет статус «локальный». Поэтому доступ к значению поля возможно извне только через свойство **intPValue**.

В опции **write** указано имя метода **SetValue**. Пусть реализация метода имеет вид:

```
void TAnyClass::SetValue(int intANewValue)
{
   if ((intANewValue != intFValue) && Correct(intANewValue)) int-
FValue=intANewValue;
}
```

Как видим, запись нового значения свойства в поле **intFValue** возможна только в случае, если устанавливается действительно новое значение свойства и, кроме того, если это новое значение прошло проверку в методе **Correct**. Использование функции **SetValue** за пределами экземпляра класса невозможно, так как ее описание расположено в разделе **private**.

Таким образом, здесь реализуется определенное поведение экземпляра класса: получить значение свойства можно без ограничений, а запись нового значения свойства осуществляется только при удовлетворении условия **Correct**.

Модифицируем рассмотренный пример:

```
class TAnyClass
{
    private:
       int intFValue;
       bool Correct(int intAValue);
       void SetValue(int intANewValue);
       int GetValue();
    public:
       __property int intPValue={read=GetValue, write=SetValue};
};
```

ЛАНЬ®

Реализация функции **GetValue** в этом случае может ограничивать доступ к значению свойства, например, проверкой пароля (детали реализации функции специально рассматривать не будем). Реализация функции **SetValue** может быть такой же, как в предыдущем примере.

Таким образом, мы задаем новое поведение экземпляра класса: чтение значения свойства осуществляется с использованием пароля, а запись значения свойства сопровождается проверкой корректности нового значения. Заметим, что использование пароля можно включить и в метод **SetValue**.

ЛАНЬ

Контрольные вопросы

1. Какие приемы можно применить для задания поведения экземпляра класса по отношению к чтению и записи значения свойства?

8.5. Обеспечение инкапсуляции в проектах

Работу между клиентскими классами и функциями, взаимодействующими для достижения целей программы, всегда можно разделить несколькими способами. Есть также несколько способов проектирования серверных классов и функций, обрабатывающих данные и выполняющих операции по запросам функций-клиентов. Если предположить, что все версии эквивалентны с точки зрения корректности программы, то возникает вопрос: как выбрать лучшую версию?

Определяемые программистом методы обычно часто модифицируются с целью придания им таких свойств, которые соответствуют удобству их вызова. Когда возникает проблема, ее источником может быть вызывающая функция (функция-клиент) или любая вызываемая функция (серверная функция). Следовательно, программист, занимающийся функцией-клиентом (или сопровождающий ее), должен изучить исходный код связанных с нею функций — клиентов и серверов. Очень часто серверные функции принадлежат другим классам, поэтому дополнительно нужно изучать устройство классов. Желательно разрабатывать функции так, чтобы свести к минимуму подобные дополнительные сложности. Принцип инкапсуляции — один из принципов, помогающих программисту достичь данной цели. После успешного тестирования серверных функций они интерпретируются программистами, отвечающими за клиентские и серверные функции, аналогично библиотечным функциям — как «черный ящик» с известным интерфейсом.

Рассмотрим простой пример — часть графического пакета, работающего с геометрическими фигурами, например цилиндрами. Для простоты предположим, что каждый объект-цилиндр характеризуется только двумя свойствами: радиусом и высотой цилиндра. Данная программа запрашивает у пользователя размеры цилиндра.

Версия 1 программы. Не задумываясь об инкапсуляции, запишем решение задачи.

Пусть описание класса «цилиндр» имеет вид:

```
class TCylinder
{
    public:
        int intFCylinderNumber;
            double dubFRadius;
            double dubFHeight;
};
```

Описание класса формы обработки свойств цилиндра

```
class TfrmCylinder : public TForm
{
__published:
        TLabel * lblFCylinderNumber;
        TEdit * edtFCylinderNumber;
        TLabel * lblFRadius;
        TEdit * edtFRadius;
        TLabel * lblFHeight;
        TEdit * edtFHeight;
        TButton * btnFSave;
        TButton * btnFExit;
        TEdit * edtFMessage;
        void __fastcall btnFSaveClick(TObject * Sender);
        void __fastcall btnFExitClick(TObject * Sender);
public:
        TCylinder * Cylinder;
```

```
        __fastcall TfrmCylinder(TComponent * Owner);
};
```

содержит 3 поля редактирования. В первом поле **edtFCylinderNumber** располагается информация о номере цилиндра. Второе **edtFRadius** и третье **edtFHeight** поля предназначены для ввода значений, соответственно, радиуса и высоты цилиндра. Поясняющие надписи находятся в компонентах **TLabel**. Поле **TEdit*edtFMessage** предназначено для вывода на форму результатов действий нашей программы, это поле должно обладать свойством «только для чтения». В составе класса также находится указатель **TCylinder*Cylinder** на обрабатываемый цилиндр, с помощью этого указателя клиентская функция передает цилиндр для обработки. Экранная форма имеет вид, представленный на рисунке 8.2.

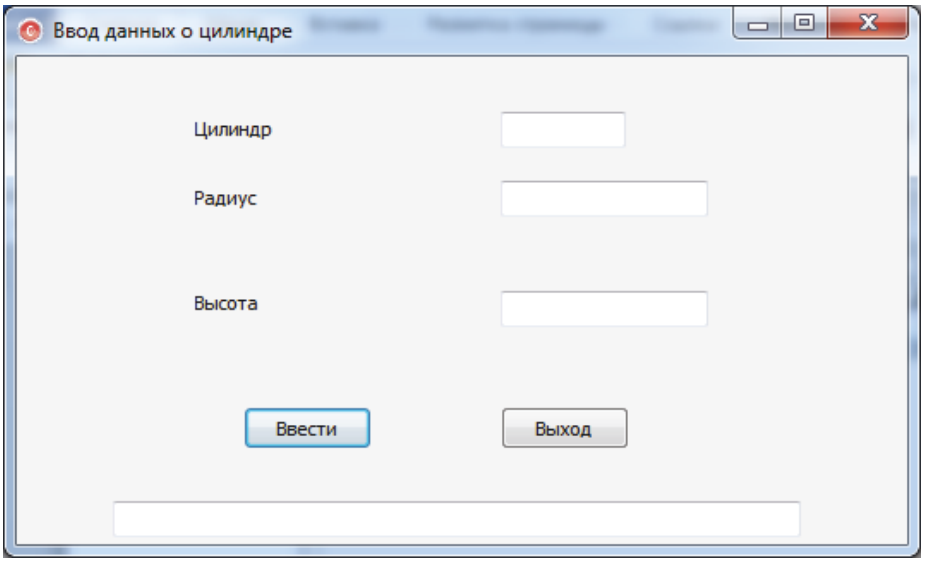


Рис. 8.2
Экранная форма ввода данных цилиндра

Методы обработки событий класса **TfrmCylinder** предназначены для запоминания введенных значений в структуре **TCylinder** обрабатываемого цилиндра

```
void __fastcall TfrmCylinder::btnFSaveClick(TObject *Sender)
{
        Cylinder–>intFCylinderNumber=StrToInt(edtFCylinderNumber–>Text);
        Cylinder–>dubFRadius=StrToFloat(edtFRadius–>Text);
        Cylinder–>dubFHeight=StrToFloat(edtFHeight–>Text);
        edtFMessage–>Text=«Сохранено»;
}
```

и обработки события завершения работы с формой

```
void __fastcall TfrmCylinder::btnFExitClick (TObject *Sender)
{
  Close();
}
```

Теперь перед нами стоит прикладная задача: имеются два цилиндра. Если объем первого цилиндра меньше объема второго, то первый цилиндр масштабируется увеличением его размеров на 20%. В реальной ситуации такой код может быть частью программы, которая использует объекты-цилиндры для описания процессов обмена, происходящих в химическом реакторе, изучения электрического тока в микропроцессоре или анализа стальной фермы.

Введем две объектные переменные, соответствующие двум цилиндрам:

TCylinder * Cylinder1, * Cylinder2;

Для решения задачи будем использовать главную форму. На главной форме имеется кнопка «Обработка цилиндров» и поле вывода сообщений, исходный текст описания главной формы и объектной переменной имеет вид:

```
class TfrmCylindersProcessing : public TForm
{
__published:
        TButton *btnFProcessing;
        TEdit *edtFMessage;
        void __fastcall FormCreate(TObject *Sender);
        void __fastcall btnFProcessingClick(TObject *Sender);
public:
        __fastcall TfrmCylindersProcessing(TComponent* Owner);
};
TfrmCylindersProcessing * frmCylindersProcessing;
```

Экранная форма представлена на рисунке 8.3.

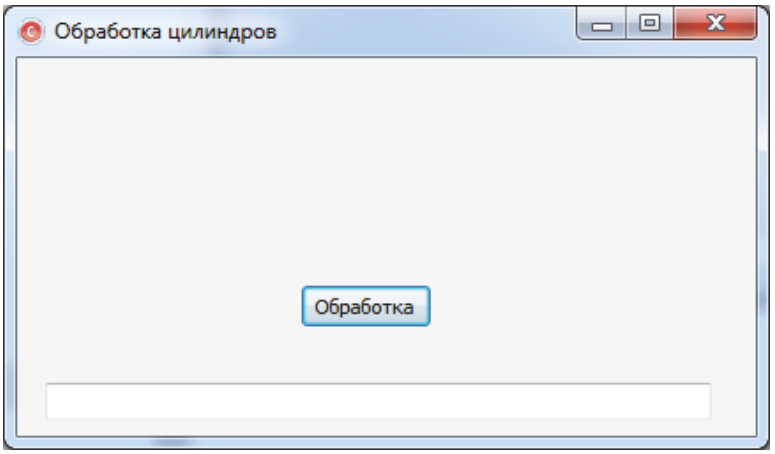


Рис. 8.3
Главная экранная форма прикладной задачи

В состав класса главной формы входят два метода. Первый метод **FormCreate** выполняется после создания главной формы и отвечает за инициализацию переменных. Реализация метода имеет вид:

```
void __fastcall TfrmCylindersProcessing::FormCreate(TObject *Sender)
{
```

```
        Cylinder1=new TCylinder();
        Cylinder2=new TCylinder();
}.
```

Второй метод класса главной формы

```
void __fastcall TfrmCylindersProcessing::btnFProcessingClick(TObject *Sender)
{
        Application–>CreateForm(__classid(TfrmCylinder), &frmCylinder);
frmCylinder–>Cylinder=Cylinder1;
        frmCylinder–>ShowModal();
        frmCylinder–>Cylinder=Cylinder2;
        frmCylinder–>ShowModal();
        if (Cylinder1–>dubFHeight * Cylinder1–>dubFRadius * Cylinder1–>dubFRadius *
                3.141593 < Cylinder2–>dubFHeight * Cylinder2–>dubFRadius *
                Cylinder2–>dubFRadius * 3.141593) // сравнить объемы
        {
                Cylinder1–>dubFRadius *= 1.2;
                Cylinder1–>dubFHeight *= 1.2; // масштабировать
                edtFMessage–>Text=«Измененный размер первого цилиндра: радиус « +
                        FloatToStr(Cylinder1–>dubFRadius)+» высота: « +
                        FloatToStr(Cylinder1–>dubFHeight); // вывести нов. размер
        }
        else // иначе ничего не делать
             edtFMessage–>Text=«Размер первого цилиндра не изменен»;
}
```

реализует основную функцию прикладной задачи.

При нажатии кнопки **btnFProcessingClick** (обработка цилиндров) поочередно данные из двух объектов «цилиндр» заносятся на экранную форму и с помощью стандартного метода **frmCylinder–>ShowModal()** форма выводится на экран.

Перед каждым вызовом формы **frmCylinder–>ShowModal();** желательно вставить действия по очистке полей экранной формы от устаревшей информации вида **frmCylinder–>edtFRadius–>Clear();** (в нашем примере отсутствуют).

В нашей реализации присутствует следующая система отношений. Экземпляр класса **TfrmCylindersProcessing**, являясь клиентом, обращается внутри функции **btnFProcessingClick()** к экземпляру класса-сервера **TfrmCylinder**. Далее при взаимодействии экземпляра класса **TfrmCylinder** с экземпляром класса **TCylinder** возникает отношение клиент-сервер. При этом, как можно было заметить, экземпляр класса **TfrmCylinder** является в одной ситуации сервером, а в другой ситуации — клиентом.

Представим себе, что у нас имеется законченная разработка классов **TfrmCylinder** и **TCylinder**. Поставим себя на место разработчика класса **TfrmCylindersProcessing** и функции **btnFProcessingClick()**. Функция **btnFProcessingClick()** обращается непосредственно к представлению данных **TCylinder**: совмещает доступ к данным (например, **Cylinder1–>dubFRadius**) с операциями над данными (такими, как вычисление объема, масштабирование

размера или вывод значений данных). В результате нам придется уяснить смысл операций не по именам функций-серверов, а по исходному коду (что сложнее). Конечно, разработчик программы может написать комментарии и пояснить смысл операторов, как это сделано в нашем примере, однако комментарии не всегда достаточно понятны читателю и не всегда точны. Иногда их просто опускают. Еще хуже, когда у разработчика нет времени на обновление комментариев при изменении исходного текста программы. Таким образом, использующий в своей программе класс **TCylinder** разработчик должен отложить решение своей основной задачи и затратить время и усилия на детальное ознакомление с реализацией класса. Из нашего простого примера может показаться, что это не так уж и сложно, но какой смысл тратить время на изучение того, что можно было бы не изучать?

Версия 2 программы. Выход из создавшейся ситуации заключается в том, чтобы создать набор серверных функций, обращающихся к полям структуры **TCylinder** от имени клиентского кода. Перекладывая обязанности выполнения вычислений на функции-серверы, можно освободить клиентский код от низкоуровневых операций. При этом имена функций несут дополнительную нагрузку: позволяют уяснить смысл вычислений без обращения к подробностям. В результате исходный текст клиента становится понятнее и говорит сам за себя. Читатель понимает по функции-клиенту, что происходит, даже если не вполне понятно, как именно это делает функция-сервер.

Пересмотрим наше первое решение с учетом правил инкапсуляции. В первую очередь пересмотрим описание класса TCylinder:

```
class TCylinder
{
        private:
          int intFCylinderNumber;
          double dubFRadius;
          double dubFHeight;
        public:
          void SetCylinder(int intACylinderNumber, double dubARadius, double du-
bAHeight);
          double GetRadius();
          double GetHeight();
          double GetVolume();      // вычислить объем
          void ScaleCylinder(double dubAFactor); // масштабировать размеры
};
```

Во-первых, доступные без ограничений в предыдущей версии программы поля класса переместились в область локальной доступности. Во-вторых, в классе появились методы, которые взяли на себя всю работу с полями. Обратим внимание на название методов и рассмотрим их реализацию.

Первый метод предназначен для записи параметров цилиндра в соответствующие локальные поля:

```
void TCylinder::SetCylinder(int intACylinderNumber, double dubARadius, double
dubAHeight)
```

```
{
  intFCylinderNumber=intACylinderNumber;
  dubFRadius=dubARadius;
  dubFHeight=dubAHeight;
}
```

Следующие два метода осуществляют чтение радиуса и высоты цилиндра:

```
double TCylinder::GetRadius()
{
   return dubFRadius;
}
double TCylinder::GetHeight()
{
   return dubFHeight;
}
```

Метод **GetVolume** возвращает вычисленное по параметрам цилиндра значение объема цилиндра:

```
double TCylinder::GetVolume() {return dubFHeight * dubFRadius * dubFRadius *
3.141593; }.
```

Наконец, метод **ScaleCylinder**:

```
void TCylinder::ScaleCylinder(double dubAFactor)
{
   dubFRadius=dubFRadius * dubAFactor;
   dubFHeight=dubFHeight * dubAFactor;
}
```

выполняет масштабирование цилиндра на величину **dubAFactor**.

Описание класса **TfrmCylindersProcessing** останется неизменным. Изменится реализация его методов:

```
void __fastcall TfrmCylindersProcessing::FormCreate(TObject *Sender)
{
        Cylinder1=new TCylinder();
        Cylinder1–>setCylinder(1,10,30);
        Cylinder2=new TCylinder();
        Cylinder2–>setCylinder(2,20,30);
}
void __fastcall TfrmCylindersProcessing::btnFProcessingClick(TObject *Sender)
{
    frmCylinder–>Cylinder=Cylinder1;
    frmCylinder–>ShowModal();
    frmCylinder–>Cylinder=Cylinder2;
    frmCylinder–>ShowModal();
    if (Cylinder1–>getVolume() < Cylinder2–>getVolume())   // сравнить объемы
          {
             Cylinder1–>scaleCylinder(1.2);    // масштабировать
             edtFMessage–>Text=«Измененный размер первого цилиндра: радиус « +
                              Cylinder1–>printCylinder();  // вывести новый размер
```

```
        }
  else    // иначе ничего не делать
     edtFMessage–>Text=«Размер первого цилиндра не изменен»;
}
```

Описание класса формы обработки свойств цилиндра также не изменится.

Чтобы понять смысл данной версии программы, на самом деле нет необходимости разбираться в том, как серверные функции **SetCylinder**, **GetRadius**, **GetHeight**, **GetVolume**, **ScaleCylinder** делают свою работу. Комментарии здесь те же, что и в предыдущей версии программы, где не используются функции доступа, но они здесь совсем не помогают, а просто повторяют то, что и так ясно из имен функций, вызываемых клиентом. Это одно из важных преимуществ «переноса обязанностей» с клиента на функции-серверы — принципа, о котором говорилось в начале параграфа.

При традиционном подходе к программированию строки комментариев очень важны. Если исходный код не содержит комментариев, то программисту следует вернуться назад и добавить их. При инкапсуляции, когда детали вычислений переносятся на серверные функции, клиентский код не нуждается в подробных комментариях. Смысл обработки ясен из имен вызываемых функций-серверов. Если без комментариев код клиента остается не вполне понятным, это означает, что функции-серверы спроектированы недостаточно хорошо. Программисту следует перепроектировать код. Хорошим приемом здесь является не использование комментариев: если вызов серверной функции не нуждается в комментарии, то мы получили успешное решение.

Версия 3 программы. Продолжим наш пример. Изменим содержание класса **TCylinder**:

```
class TCylinder
{
private:
  int intFCylinderNumber;
  double dubFRadius;
  double dubFHeight;
  void SetCylinderNumber(int intANew);
  void SetRadius(double dubANew);
  void SetHeight(double dubANew);
  double GetVolume();  // вычислить объем
public:
  __property int intPCylinderNumber={read=intFCylinderNumber,
                                     write=SetCylinderNumber};
  __property double dubPRadius={read=dubFRadius,write=SetRadius};
  __property double dubPHeight={read=dubFHeight,write=SetHeight};
  __property double dubPVolume={read=GetVolume};
  void ScaleCylinder(double dubAFactor); // масштабировать размеры
};
```

Вместо методов обработки значений параметров цилиндра теперь мы имеем дело со свойствами. Приведем реализацию задействованных в свойствах методов:

```
void TCylinder::SetCylinderNumber(int intANew)
{
  intFCylinderNumber=intANew;
}
void TCylinder::SetRadius(double dubANew)
{
  dubFRadius=dubANew;
}
void TCylinder::SetHeight(double dubANew)
{
  dubFHeight=dubANew;
}
double TCylinder::GetVolume() // вычислить объем
{
  return dubFHeight * dubFRadius * dubFRadius * 3.141593;
}
```

Незначительно изменится реализация метода **btnFProcessingClick**:

```
void __fastcall TfrmCylindersProcessing::btnFProcessingClick(TObject *Sender)
{
  frmCylinder–>Cylinder=Cylinder1;
  frmCylinder–>ShowModal();
  frmCylinder–>Cylinder=Cylinder2;
  frmCylinder–>ShowModal();
  if (Cylinder1–>dubPVolume < Cylinder2–>dubPVolume)   // сравнить объемы
        {    // масштабировать
             Cylinder1–>ScaleCylinder(1.2);
             edtFMessage–>Text=«Измененный размер первого цилиндра: радиус
«+FloatToStr(Cylinder1–>dubPRadius)+» высота «+FloatToStr(Cylinder1–
>dubPHeight);  // вывести новый размер
        }
  else    // иначе ничего не делать
     edtFMessage–>Text=«Размер первого цилиндра не изменен»;
}
```

Последняя версия программы наиболее точно отражает объектно-ориентированный стиль программирования. Обычно комбинирование доступа к данным с вычислениями их значений затрудняет и делает не очень понятной проверку данных. Часто ее просто опускают. В этом примере данные поступают от пользователя, и следует защитить программу от ошибок. В реальной ситуации данные могут считываться из внешнего файла или поступать по коммуникационной линии. Как и пользователь, эти источники нередко дают ошибочные данные. Между тем, даже простейшая защита от ошибок (например, присваивание полям Cylinder значений по умолчанию) усложняет код клиента. В последней версии программы проверка значений легко может быть локализована. Например, функцию **SetRadius** можно реализовать следующим образом:

```
void TCylinder::SetRadius(double dubANew)
{
```

```
    if (dubANew < 0) dubFRadius=10;
   else dubFRadius=dubANew;
}
```

Версия 4 программы. Предложим, наконец, экзотический вариант рассмотренной программы. Он связан с тем, что одни программисты-пользователи предпочитают использовать свойства, а другие — методы. Поэтому можно объединить возможности второй и третьей версий, т. е. ввести методы из второй версии программы и, одновременно, ввести свойства из третьей версии программы. Конечно, это приведет к удорожанию проекта, но при этом повысится удобство использования объектов. Текст программы предлагаем разработать самостоятельно.

Инкапсуляция данных — относительно новая концепция, и ее не всегда хорошо понимают. Многие программисты считают, что инкапсуляция данных относится к защите функций от ошибочных и несанкционированных изменений. Без такой инкапсуляции клиентская программа, обращаясь к полям прямо по имени, может произвольно и незаметно изменять данные. При инкапсуляции данных клиент вызывает функции доступа к данным, и эти функции изменяют данные.

Такой вопрос защиты данных аналогичен проблеме использования глобальных переменных. Если глобальные имена доступны во всей программе, то кто-то может некорректно присвоить им значения, что повлияет на другие части программы. Поэтому неслучайно возникло правило объектно-ориентированного программирования: не использовать глобальные переменные.

Таким образом, инкапсуляция данных — это, прежде всего, удобство чтения исходного кода и независимость компонентов программы.

В случае инкапсуляции при изменении имени или типа поля данных изменять приходится лишь набор функций доступа. На другие части программы это не влияет. Перекомпилировать также потребуется только те части программы, которые обращаются к данным функциям, но их исходный код при этом не изменяется.

Следовательно, сопровождающий приложение программист должен сосредоточить свое внимание на относительно узкой области, ограниченной кодом, имеющим дело с именами полей данных. Вот в чем истинное преимущество инкапсуляции. Если имена полей данных не используются непосредственно, то в клиенте можно обойти зависимость обработки от архитектуры данных.

Очень важно научиться продумывать архитектуру программы с точки зрения инкапсуляции данных. В этом случае создается две разные сферы полномочий: сегменты кода, использующие имена полей данных и не использующие их.

Правило. *Объектно-ориентированная программа должна содержать только классы и объекты. Управление состояниями объектов осуществляется только с помощью свойств и методов.*

Это сужает горизонт разработчиков клиентской части и сопровождающих приложение программистов, но приводит к повышению надежности программы.

Контрольные вопросы

1. Какими признаками обладает хорошо спроектированная программа?
2. В рассмотренных примерах не рассматривалось разбиение программы на модули. Какие ваши предложения по структуре модулей?

8.6. Абстрагирование и инкапсуляция

Тему «Абстрагирование» мы обсуждали в главе 3. Теперь рассмотрим связь абстрагирования и инкапсуляции.

Абстрагирование связано с тем, как данная проблема представлена в пространстве программы. Во-первых, абстрагирование заложено в самих языках программирования. Постараемся вспомнить: давно ли нам приходилось заботиться о стеке или регистрах процессора? Возможно, когда-то многие из нас изучали программирование на языке ассемблера, но сейчас детали реализации программы на низшем, машинно-зависимом, уровне занимают умы узкого круга программистов. Причина проста: большинство языков отстраняют нас (абстрагируют) от таких подробностей, позволяя сосредоточиться на решении прикладной задачи.

При объявлении классов в объектно-ориентированных языках мы должны использовать такие имена и интерфейсы, которые отражают смысл и назначение объектов предметной области. Скрытие элементов, не связанных напрямую с решением задачи, позволяет нам полностью сосредоточиться на самой задаче и решить ее более эффективно. Перефразируя высказывание из книги Брюса Эккеля (Bruce Eckel) «Философия C++», можно сказать: в большинстве случаев умение достичь решения проблемы сводится к качеству применяемого абстрагирования.

Однако язык — это один уровень абстрагирования. Если мы пойдем дальше, то, как разработчику класса, нам нужно придумать такую степень абстрагирования, чтобы клиенты нашего класса могли сразу сосредоточиться на своей задаче, не тратя время на изучение работы класса. На очевидный вопрос: какое отношение интерфейс класса имеет к абстрагированию? — можно ответить так: интерфейс класса и есть реализация абстрагирования.

Чтобы обсуждаемые здесь идеи были понятней, воспользуемся аналогией с работой внутренних устройств торговых автоматов. Описать подробно, что происходит внутри торгового автомата, довольно трудно. Чтобы выполнить свою задачу, автомат должен принять деньги, рассчитать, дать сдачу, а затем — требуемый товар. Однако покупателям — пользователям автомата, видно лишь несколько его функций. Элементы интерфейса автомата: щель для приема денег, кнопки выбора товара, рычаг для запроса сдачи, лоток, куда поступает сдача, и желоб подачи товара. Торговые автоматы остаются без изменений (более или менее) со времени их изобретения. Это связано с тем, что их внутренняя организация совершенствовалась по мере развития технологии, а основной интерфейс не нуждался в больших переменах.

Неотъемлемой частью проектирования интерфейса класса является достаточно глубокое понимание предметной области. Такое понимание поможет нам создать интерфейс, предоставляющий пользователям доступ к нужной им информации и методам, но изолирующий их от внутренней организации класса. При разработке интерфейса мы должны думать не только о решении текущей задачи, но и о том, чтобы обеспечить такое абстрагирование от внутреннего представления класса, которое позволит неограниченно модифицировать закрытые элементы класса, не затрагивая существующего кода.

При определении нужной степени абстрагирования класса важно помнить и о программисте клиентского кода. Представим, что мы пишем основное ядро базы данных. Возможно, мы прекрасно разбираемся в таких понятиях баз данных, как *курсоры* (cursors), *управление фиксацией* (commitment control) и *кортежи* (tuples). Однако многие разработчики, не столь искушенные в программировании баз данных, не собираются вникать в тонкости этих понятий. Используя терминологию, не понятную клиентам нашего класса, мы не достигнем основной цели абстрагирования — повысить эффективность работы программиста путем представления предметной области в понятных ему и естественных терминах.

Кроме того, решая, какие элементы класса сделать открытыми, надо опять вспомнить о клиенте. Это еще раз подтверждает необходимость иметь хотя бы начальное представление о предметной области и клиентах вашего класса. Так, в случае с базой данных наши клиенты, наверное, не должны иметь прямого доступа к компонентам, представляющим внутренние буферы данных. Ведь структура этих буферов может когда-нибудь измениться. Кроме того, от целостности этих буферов зависит вся работа ядра базы данных, и поэтому операции по их изменению следует выполнять только нашими методами. Только после этого можно сказать, что предприняты все меры предосторожности.

Может показаться, что применение объектно-ориентированных технологий, главным образом, исчерпывается более упрощенным созданием классов. При этом на самом деле достигается некоторый выигрыш в производительности, однако долговременные выгоды мы получим, поняв, что основное назначение объектно-ориентированного программирования в облегчении клиентам программирования классов. При разработке своих классов мы всегда должны ставить себя на место программиста, которому предстоит работать либо с экземплярами этих классов, либо с производными от них классами.

Наличие в классах абстрагирования, которое максимально удобно для программистов, работающих с этими классами, имеет первостепенное значение при разработке повторно используемого программного обеспечения. Если мы выстроим интерфейс, на который не влияют изменения в реализации, то нашему приложению долгое время не понадобятся никакие модификации. Вспомним пример с расчетом зарплаты. При работе с объектом Employee и функциями, обеспечивающими расчет зарплаты, клиенту нужны лишь несколько методов, таких как CalculatePay, GetAddress и GetEmployeeType. Если мы знакомы с предметной областью задачи, то без труда определим, какие методы понадобятся пользователям класса. Скажем так: если при проектировании класса нам удается сочетать хорошее знание предметной области с прогнозом относительно дальнейших перспектив использования класса, можно гарантировать, что большая часть интерфейса этого класса останется неизменной, даже в случае возможного совершенствования реализации класса. В данном примере для пользователя главным является только класс Employee, в котором, с его точки зрения, от версии к версии лучше бы ничего не менять.

В результате отстранения пользователя от деталей реализации система в целом становится понятнее, а значит, и удобнее в работе. Иначе обстоит дело с та-

кими процедурными языками как C, в которых нужно показать явно каждый модуль и предоставить доступ к элементам структуры. И при каждом ее изменении нужно редактировать строки кода, имеющие отношение к данной структуре.

Сказанное непосредственно относится и к взаимоотношению программист — пользователь программного продукта. Проиллюстрируем наши рассуждения примером. Автору пришлось разрабатывать программное обеспечение для отдела социальной защиты населения районной администрации. На экранной форме была размещена кнопка с надписью: «Переиндексация базы данных». От работников отдела посыпались вопросы: что такое «переиндексация»? После долгих и безуспешных попыток объяснить понятие «переиндексация» с совершенно неинтересным работникам отдела кратким экскурсом в теорию баз данных и некоторого анализа ситуации автор написал на «несчастной» кнопке «Ремонт файлов». Все вопросы были сняты!

Контрольные вопросы

1. Какую абстракцию содержит язык программирования?
2. Какое отношение интерфейс класса имеет к абстрагированию?
3. Какие элементы класса следует оставлять открытыми?
4. Для чего программисту нужно хорошее знание предметной области приложения?

Задачи

В приведенных задачах необходимо:

— продумать архитектуру программы с точки зрения инкапсуляции данных;

— каждый класс разместить в отдельном программном модуле;

— для каждого класса тщательно проработать содержание интерфейсной и реализационной части программного файла;

— в раздел public размещать только свойства и методы;

— для полей, свойств и методов использовать осмысленные идентификаторы, позволяющие понять поведение класса без использования комментариев;

— обеспечить проверку корректности исходных данных.

Выбор способа реализации ввода значений параметров задачи (свойства и/или методы) остается за читателем.

Список заданий.

1. Создать класс TMyFile, инкапсулирующий в себе такие функции работы с файлами, как Open, Close, Seek, Read, Write, GetPosition, GetLength. На базе этого класса создать производный класс TMyDataFile-файл, содержащий в себе данные некоторого определенного типа TMyData, а также заголовок, облегчающий доступ к этому файлу. Написать программу, демонстрирующую работу с этими классами. Программа должна содержать меню, позволяющее осуществить проверку всех методов классов.

2. Создать класс TMyPoint — точка. На его основе создать классы TColoredPoint — цветная точка и TLine — отрезок прямой. На основе класс TLine создать класс TColoredLine — цветная линия и класс TPolyLine — многоуголь-

ник. Все классы должны иметь средства для установки и получения всех координат, а также изменения цвета и получения текущего цвета. Написать демонстрационную программу, в которой будет использоваться список адресов объектов этих классов в динамической памяти.

3. Создать класс TVehicle (машина). На его основе реализовать классы TPlane-самолет, TCar-автомобиль, TShip-корабль. Классы должны иметь возможность задавать и получать координаты, параметры средств передвижения (цена, скорость, год выпуска). Для самолета должна быть определена высота полета, для самолета и корабля — количество пассажиров, для корабля — порт приписки. Написать демонстрационную программу, создающую список адресов объектов этих классов в динамической памяти. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

4. Описать базовый класс «Элемент».

Поля: имя элемента (указатель на строку символов); количество входов элемента; количество выходов элемента.

Методы: конструктор класса; деструктор класса; метод, задающий имя элемента.

А) На основе класса «Элемент» описать производный класс «Комбинационный», представляющий собой комбинационный элемент (двоичный вентиль), который может иметь несколько входов и один выход.

Поля: указатель, используемый для динамического размещения полей, содержащих значения входов.

Методы: конструктор; конструктор копирования; деструктор; метод, задающий значение на входах экземпляра класса; метод, позволяющий опрашивать состояние отдельного входа экземпляра класса; метод, вычисляющий значение выхода (по варианту задания).

Б) На основе класса «Элемент» описать производный класс «Память», представляющий собой триггер. Триггер имеет входы, соответствующие типу триггера (см. ниже вариант задания), и входы установки и сброса. Все триггеры считаются синхронными, сам синхровход в состав триггера не включается.

Поля: массив значений входов объекта класса (задается статически), в массиве учитываются все входы (управляющие и информационные); состояние на прямом выходе триггера; состояние на инверсном выходе триггера.

Методы: конструктор (по умолчанию сбрасывает состояние триггера); конструктор копирования; деструктор; метод, задающий значение на входах экземпляра класса; методы, позволяющие опрашивать состояния отдельного входа экземпляра класса; метод, вычисляющий состояние триггера (по варианту задания) в зависимости от текущего состояния и значений на входах.

В) Создать класс «Регистр», используя класс «Память» как включаемый класс.

Поля: состояние входа «Сброс» — один для экземпляра класса; состояние входа «Установка» — один для экземпляра класса; статический массив типа «Память» заданной в варианте размерности; статический(е) массив(ы), содержащий(е) значения на соответствующих входах элементов массива типа «Память».

Методы: метод, задающий значение на входах экземпляра класса (желательно в качестве параметров передавать методу указатели на массивы значений); метод, позволяющий опрашивать состояние отдельного выхода экземпляра класса; метод, вычисляющий значение нового состояния экземпляра класса.

В задании перечислены только обязательные поля и методы класса. Можно задавать дополнительные поля и методы, если они не отменяют обязательные и обеспечивают дополнительные удобства при работе с объектами.

Для проверки функционирования созданных классов написать программу. В программе должны быть продемонстрированы все свойства созданных классов.

Конкретный тип комбинационного элемента, тип триггера и разрядность регистра выбираются в соответствии с вариантом задания.

Вариант	Комбинационный элемент	Число входов	Триггер	Разрядность регистра
1	И-НЕ	4	RS	8
2	ИЛИ	5	RST	10
3	МОД2-НЕ	6	D	12
4	И	8	T	8
5	ИЛИ-НЕ	8	V	9
6	И	4	RS	10
7	ИЛИ-НЕ	5	JK	11
8	МОД2	5	D	8
9	И	4	T	10
10	ИЛИ	3	JK	8
11	И-НЕ	3	RS	12
12	ИЛИ-НЕ	4	RST	4
13	МОД2	5	D	10
14	МОД2-НЕ	6	T	10
15	ИЛИ-НЕ	8	V	10
16	И	8	JK	6
17	И-НЕ	8	RS	10
18	ИЛИ	8	T	10
19	МОД2	6	JK	8
20	МоД2-НЕ	5	V	10

5. Описать базовый класс СТРОКА.

Обязательные поля класса: указатель на char — хранит адрес динамически выделенной памяти для размещения символов строки; значение типа int — хранит длину строки в байтах.

Методы: конструктор без параметров; конструктор, принимающий в качестве параметра С-строку (заканчивается нулевым байтом); конструктор, принимающий в качестве параметра символ; конструктор копирования; получение длины строки; очистка строки (сделать строку пустой); деструктор.

Создать производный от СТРОКА класс СТРОКА_ИДЕНТИФИКАТОР. Строки данного класса строятся по правилам записи идентификаторов. Если исходные данные противоречат правилам записи идентификатора, то создается пустая строка.

Методы: конструктор без параметров; конструктор, принимающий в качестве параметра С-строку (заканчивается нулевым байтом); конструктор, при-

нимающий в качестве параметра символ; конструктор копирования; перевод всех символов строки в верхний регистр; перевод всех символов строки в нижний регистр; поиск первого вхождения символа в строку; деструктор.

Также предусмотреть операции: сложения строк, вычитания строк (из первого операнда удаляются все символы, входящие во вторую строку), отношения на строках «меньше», «больше».

Написать тестовую программу, которая:

— динамически выделяет массив указателей на базовый класс (4–6 элементов);

— в режиме диалога заполняет этот массив указателями на производные классы, при этом экземпляры производных классов создаются динамически с заданием начальных значений;

— для созданных экземпляров производных классов выполняет проверку всех разработанных методов с выводом исходных данных и результатов.

Для конструкторов копирования каждого класса предусмотреть диагностический вывод количества его вызовов.

6. Описать базовый класс СТРОКА.

Обязательные поля класса: указатель на char — хранит адрес динамически выделенной памяти для размещения символов строки; значение типа int — хранит длину строки в байтах.

Методы: конструктор без параметров; конструктор, принимающий в качестве параметра С-строку (заканчивается нулевым байтом); конструктор, принимающий в качестве параметра символ; конструктор копирования; получение длины строки; очистка строки (сделать строку пустой); деструктор.

Создать производный от СТРОКА класс БИТОВАЯ_СТРОКА. Строки данного класса могут содержать только символы «0» или «1». Если в составе инициализирующей строки будут встречены любые символы, отличные от допустимых, БИТОВАЯ_СТРОКА принимает нулевое значение. Содержимое данных строк рассматривается как двоичное число. Отрицательные числа хранятся в дополнительном коде.

Методы: конструктор без параметров; конструктор, принимающий в качестве параметра С-строку (заканчивается нулевым байтом); конструктор копирования; деструктор; изменение знака на противоположный (перевод числа в дополнительный код); операция арифметического сложения; операция проверки на равенство.

Написать тестовую программу, которая:

— динамически выделяет массив указателей на базовый класс (4–6 элементов);

— в режиме диалога заполняет этот массив указателями на производные классы, при этом экземпляры производных классов создаются динамически с заданием начальных значений;

— для созданных экземпляров производных классов выполняет проверку всех разработанных методов с выводом исходных данных и результатов.

Для конструкторов копирования каждого класса предусмотреть диагностический вывод количества его вызовов.

7. Описать базовый класс СТРОКА.

Обязательные поля класса: указатель на char — хранит адрес динамически выделенной памяти для размещения символов строки; значение типа int — хранит длину строки в байтах.

Методы: конструктор без параметров; конструктор, принимающий в качестве параметра С-строку (заканчивается нулевым байтом); конструктор, принимающий в качестве параметра символ; конструктор копирования; получение длины строки; очистка строки (сделать строку пустой); деструктор.

Описать производный от СТРОКА класс ДЕСЯТИЧНАЯ_СТРОКА.

Строки данного класса могут содержать только символы десятичных цифр и символы «–» и «+», задающие знак числа. Символы «–» или «+» могут находиться только в первой позиции числа, причем символ «+» может отсутствовать, в этом случае число считается положительным. Если в составе инициализирующей строки будут встречены символы, отличные от допустимых, ДЕСЯТИЧНАЯ_СТРОКА принимает нулевое значение. Содержимое данных строк рассматривается как десятичное число.

Методы: конструктор без параметров; конструктор, принимающий в качестве параметра С-строку (заканчивается нулевым байтом); конструктор копирования; деструктор; метод, определяющий, можно ли представить данное число в формате int; операция вычитания (арифметического); отношения равно, меньше, больше (по значению).

Написать тестовую программу, которая:

— динамически выделяет массив указателей на базовый класс (4–6 элементов);

— в режиме диалога заполняет этот массив указателями на производные классы, при этом экземпляры производных классов создаются динамически с заданием начальных значений;

— для созданных экземпляров производных классов выполняет проверку всех разработанных методов с выводом исходных данных и результатов.

Для конструкторов копирования каждого класса предусмотреть диагностический вывод количества его вызовов.

8. Описать базовый класс СТРОКА.

Обязательные поля класса: указатель на char — хранит адрес динамически выделенной памяти для размещения символов строки; значение типа int — хранит длину строки в байтах.

Методы: конструктор без параметров; конструктор, принимающий в качестве параметра С-строку (заканчивается нулевым байтом); конструктор, принимающий в качестве параметра символ; конструктор копирования; очистка строки (сделать строку пустой); деструктор.

Производный от СТРОКА класс КОМПЛЕКСНОЕ_ЧИСЛО.

Строки данного класса состоят из двух полей, разделенных символом «i». Первое поле задает значение реальной части числа, а второе — мнимой. Каждое из полей может содержать только символы десятичных цифр и символы «–» и «+», задающие знак числа. Символы «–» и «+» могут находиться только в первой позиции числа, причем символ «+» может отсутствовать, в этом случае число

считается положительным. Если в составе инициализирующей строки будут встречены символы, отличные от допустимых, КОМПЛЕКСНОЕ_ЧИСЛО принимает нулевое значение. Примеры строк: 53i15, –5i206, +9i–62.

Методы: конструктор без параметров; конструктор, принимающий в качестве параметра С-строку (заканчивается нулевым байтом); конструктор копирования; деструктор; операция вычитания (арифметического); отношение равно (по значению).

Написать тестовую программу, которая:

— динамически выделяет массив указателей на базовый класс (4–6 элементов);

— в режиме диалога заполняет этот массив указателями на производные классы, при этом экземпляры производных классов создаются динамически с заданием начальных значений;

— для созданных экземпляров производных классов выполняет проверку всех разработанных методов с выводом исходных данных и результатов.

Для конструкторов копирования каждого класса предусмотреть диагностический вывод количества его вызовов.

ЛАНЬ®

9. ПОЛИМОРФИЗМ

Размышляя о полиморфизме, приходишь к очередному выводу: существуют идеи, и только после — их реализация.

Идея представляет что, но не обязательно как.

Джефф Дантеман

Полиморфизм, наряду с наследованием и инкапсуляцией, относится к основным концептуальным основам объектно-ориентированного программирования.

Понятие полиморфных операций рассмотрено в главе 4. Существенным свойством полиморфной функции (процедуры) является наличие только одного тела функции (процедуры), независимо от сочетаний типов параметров. В настоящей главе мы рассмотрим назначение полиморфизма, средства языка программирования для обеспечения полиморфизма, принципы реализации полиморфизма в системе программирования, а также приемы использования полиморфизма в разработке программ с целью придать этим программам свойства модифицируемости.

9.1. Назначение и виды полиморфизма

Слово «полиморфизм» греческого происхождения (poly — много, morphos — форма), имеет отношение к греческому богу сна Морфею, который мог являться спящим людям в любой форме, следовательно, был воистину полиморфным. Понятие «полиморфизм» используется довольно часто в различных областях знаний. Например, в биологии полиморфизм обозначает способность некоторых организмов существовать в состояниях с различной внутренней структурой (матка и рабочая пчела — это две разновидности пчелиных самок); полиморфизм кристаллов — это способность вещества существовать в различных кристаллических структурах (углерод имеет две кристаллические формы — графита и алмаза); в языках программирования и теории типов полиморфизмом называется способность функции обрабатывать данные разных типов. В этом параграфе мы попытаемся раскрыть смысл и использование понятия «полиморфизм» в языке программирования (C++Builder), а также виды полиморфизма.

9.1.1. Метафора полиморфизма

Концепция полиморфизма используется для придания программным изделиям свойства модифицируемости. *Модифицируемость* программных продуктов означает приспособленность к внесению изменений, например, расширение функциональных возможностей программы, переход на другую техническую базу и т. п.

Основными причинами возникновения потребности в модификации программного обеспечения являются следующие:

— устаревание программного обеспечения;

— присутствие ряда недостатков, снижающих эксплуатационное качество программного обеспечения;

— необходимость усовершенствовать программу под текущие требования;

— утрата контроля над содержащимися в программе данными.

Процесс устаревания программного обеспечения провоцирует снижение эффективности программ, модернизацию которых обычно выполняют авторы программы. Например, частой причиной устаревания программы является недолговечность (моральная) носителей информации. Появление новых, более объемных и быстродействующих устройств хранения информации вынуждает разработчиков программ проводить их модернизацию. Более того, почти каждые 10 лет происходит смена архитектур компьютеров, языков программирования и операционных сред, возникают проблемы сопровождения готовых программных средств и их компонентов в новой среде или архитектуре, решение которых приводит к изменению либо обновлению отдельных элементов системы или системы полностью. Иногда устаревание программы может быть запланированным из сугубо коммерческих соображений.

Недостатки, влияющие на качество программного обеспечения, являются следствием недоработок или ошибок разработчиков и могут быть внесены на различных этапах жизненного цикла: проектирования, кодирования, конфигурирования и эксплуатации.

Желание расширить функциональность программы, улучшить пользовательский интерфейс создает потребность в усовершенствовании программы. Как правило, потребители программного изделия, получив опыт его использования для решения своих задач, начинают более точно и профессионально формулировать требования к функциональности и эргономичности программного инструмента. Важную роль играют и научные исследования в области автоматизации методов решения задач. Эти и другие факторы вызывают потребность в модификации программного изделия.

Утрата контроля над содержащимися в программе данными может возникнуть, например, в связи с усилившимися требованиями к защищенности персональных данных.

С учетом сказанного можно сделать вывод: процесс модернизации программных изделий неизбежен, поэтому разработчик программного обеспечения должен заранее позаботиться о том, чтобы создать условия для модернизации своих изделий, т. е. придать программам свойство модифицируемости.

Можно однозначно утверждать, что программа, в которой даже небольшое изменение структур данных или алгоритмов влечет переделку всего остального текста, не обладает свойством модифицируемости. Поэтому необходимо стремиться разрабатывать программы так, чтобы любые переделки выполнялись локально в ограниченных участках программы, не оказывая существенного влияния на другие фрагменты текста программы. Существует много технологических, организационных и других приемов обеспечения модифицируемости программ.

Наиболее известные технологии обновления программного компонента и программы в целом включают в себя:

— реинженерию, целью которой является повторная реализация программы (системы) в целях повышения удобства ее эксплуатации, сопровождения или изменения ее функций;

— рефакторинг — это совокупность моделей, методов, процессов, применяемых к определенным классам объектов и компонентов для получения новых или изменения существующих объектов-компонентов с целью повышения качественных характеристик программной системы в целом или добавления новых возможностей к существующей компонентной системе;

— реверсная инженерия ориентирована на индустриальные системы в миллионы строк кода с использованием метрических оценок характеристик системы. Метод реверсной инженерии разрешает генерацию тестов для проверки кодов, а также проведение метрического анализа системы для получения фактических значений внутренних и внешних характеристик системы.

Среди множества технологий и приемов обновления программных компонентов важное место занимает полиморфизм. Программы должны обладать свойством полиморфизма, т. е. допускать существование многих форм. В этом случае введение новых и переделка существующих форм программы будут осуществляться с наименьшими затратами.

Полиморфизм — это техника, позволяющая одному и тому же фрагменту кода оперировать разными типами входных данных. В случае полиморфизма свойство модифицируемости программы достигается за счет «нечувствительности» к изменениям типов обрабатываемых данных.

В общем случае основной механизм полиморфизма строится на использовании следующих приемов:

— обобщенный указатель совместно с приведением типов;

— обобщенное имя функций с различными наборами аргументов;

— типовый параметр с последующей подстановкой конкретного типа.

Рассмотрим в общих чертах использование приемов полиморфизма в программах.

1. Использование обобщенного указателя совместно с приведением типов. Предполагается, что параметром функции является переменная некоторого общего (универсального) типа. Во время вызова функции этой переменной присваивается значение конкретного типа. В языках группы С в качестве переменной общего типа обычно используется переменная типа **void*** или **void****. Например, пусть имеется описание функции **Boolean MoreOrEqual (void * x1, x2)**. Предположим, вызов этой функции может осуществляться с конкретными значениями параметров целого или вещественного, или строкового типа, или другого типа. Рассматриваемый прием возможен только в том случае, когда в реализации функции **MoreOrEqual** будет возможность выполнять нужные действия для каждого типа значений фактических параметров.

Другая возможность использования обобщенного указателя появляется при наличии нескольких родственных классов с общим предком. Например, это могут быть классы различных потоков **TFileStream**, **TMemoryStream**, **TStringStream** с общим предком — **TStream**. Пусть также имеется метод:

```
void SaveToStream (TStream * Stream)
{
   int Count;
  Stream–>WriteBuffer(Count, SizeOf(Count));
  for (int i=0; i < Count; i++)
     Stream–>WriteBuffer(Data[i], SizeOf(Data[i]));
 }
```

Данный метод можно считать полиморфным, так как все его вызовы с экземплярами классов **TFileStream** или **TMemoryStream** или **TStringStream** будут допустимыми. Если мы подставим в вызов функции **SaveToStream** экземпляр **TMemoryStream**, данные будут записаны в память, если **TFileStream** — в файл. Допустимыми будут также вызовы метода с экземпляром любого потомка класса **TStream**. Наш код может вести себя по-разному в зависимости от того, с каким классом реально имеет дело. В то же время он нигде не проверяет классы, а просто использует общий для всех классов данной иерархии интерфейс. Поэтому появление со временем новых потомков класса **TStream** позволит использовать уже имеющийся код функции **SaveToStream** без переделок.

Из этого примера должно быть видно назначение полиморфизма — обеспечить возможность писать обобщенный клиентский код, а всю конкретику переложить на сами классы. Благодаря этому мы избавлены от необходимости писать построенные на условных операторах сложные длинные проверки, код становится более коротким, понятным и, что не менее важно, — расширяемым, т. е. наш код работает даже с классами, которых еще нет, но которые обязуется поддерживать заложенный в базовом классе интерфейс.

Примечание: под наследованием в объектно-ориентированном программировании обычно подразумевают две вещи: наследование интерфейса и наследование реализации. Первое означает, что потомок получает интерфейс своего предка и может дополнить его своей реализацией. Второе — что реализация методов от предка также переходит потомку, если ничего не предпринять для того, чтобы это изменить. Для осуществления полиморфного поведения достаточно только, чтобы потомок унаследовал интерфейс, а реализация в базовом классе иногда не нужна, главное — это возможность ее подменить своей. Для того чтобы не писать реализацию в базовом классе, можно использовать абстрактные методы. Синтаксическим признаком абстрактного метода является приравнивание объявления метода нулю. Например, стандартный класс **TStrings** содержит абстрактный метод

```
virtual int __fastcall GetCount(void)=0;
```

а его непосредственный потомок **TStringList** переопределяет данный метод.

2. Использование обобщенного имени функций с различными наборами аргументов поддерживается во многих языках программирования посредством перегрузки функций. Этот подход применяется в статически типизированных языках, которые проверяют типы аргументов при вызове функции. Перегруженная функция фактически представляет собой несколько разных функций и выбор подходящей происходит на этапе трансляции.

Для демонстрации перегруженных функций рассмотрим пример. Пусть определены три функции:

```
int Volume (int s)              // Объем куба, заданного стороной  s
{
   return s * s * s;
}
double Volume (double r, int h)  // Объем цилиндра, заданного радиусом r и высотой h
{
   return 3.14 * r*r*h;
}
long int Volume (long int a, b, c) // Объем параллелепипеда, заданного сторонами a, b, c
{
   return a * b * c;
}.
```

Теперь вызов **Volume (50);** возвращает объем куба; вызов **Volume (25.0, 10);** возвращает объем цилиндра; вызов **Volume (150, 320, 240);** возвращает объем параллелепипеда. Заметим, что вызов **Volume (1, 2, 3, 4);** будет некорректным.

Таким образом, выбор подходящей функции осуществляется автоматически по типу данных и количеству параметров в вызове.

3. Использование типового параметра с последующей подстановкой конкретного типа. При таком подходе некоторый параметр может быть описан только частично, например, «список из Т», где тип данных Т остается неопределенным. Это позволяет, например, вводить функции, оперирующие со списками. Такие функции могут вызываться для списков с элементами произвольного типа.

Попытаемся развить указанную мысль, используя понятия абстракции высокого уровня и абстракции низкого уровня. Абстракция высокого уровня отражает планирование более общего характера (например, алгоритм сортировки или изменения размера окна), когда определяется общий подход, без конкретизации деталей. Абстракция низкого уровня — это базовая операция (например, над структурой данных), которая построена на механизмах реализации.

Алгоритмы обычно описываются абстракциями высокого уровня. Их фактическая реализация также должна быть абстракцией высокого уровня, основанной, однако, на структурах низкого уровня. Приведем пример. На высоком уровне рекурсивный алгоритм, вычисляющий длину **length** списка **list**, записывается следующим образом:

```
int length (TList  list)
{
if (list.link == NULL)return 1;
else return 1+length(list.link);
},
```

ЛАНЬ®

где **link** — ссылочное поле, соединяющее отдельные звенья в список.

Фактическая реализация алгоритма на традиционном языке программирования потребует конкретной спецификации не только тех данных, которые используются в алгоритме (поле **link**), но и всех свойств, которые входят

в структуру данных, но не задействованы в алгоритме. Тем самым алгоритм должен быть переписан следующим образом:

```
struct TIntList
{
  int value;
  TIntList *link;
};
int length (TIntList *x)
{
 if (x–>link == NULL) return 1;
 else return 1+length(x–>link);
}
```

Теперь функция **length** может быть использована для вычисления длины связного списка, составленного из целых чисел, но она непригодна для определения длины связного списка вещественных чисел. Вычисление длины любого нецелочисленного списка потребует полного переопределения структур данных.

Для использования абстракций высокого уровня в C++ предусмотрен инструмент шаблонов (template). Шаблон позволяет ввести типовую переменную, которая в процессе трансляции получит конкретное значение. Для рассмотренного примера вычисления длины списка шаблон и описание функции могут выглядеть следующим образом:

```
template <typename T>
int length (T  list)
{
if (list.link == NULL)return 1;
else return 1+length(list.link);
}
```

К функции **length** можно будет обращаться с указанием в качестве параметра переменной, обозначающей список из элементов любого типа. Например, с переменной **TIntList lst** можно записать вызов:

```
length(lst);
```

Большинство программ состоят из абстракций как высокого, так и низкого уровней. Разработчик, знакомый с понятиями структурного программирования и абстракциями данных, сразу же определит абстракции низкого уровня и воплотит их так, чтобы они не зависели от конкретного применения. То есть абстракции низкого уровня — это инструменты, которые могут переноситься из одного проекта в другой. С другой стороны, в традиционных языках программирования абстракция высокого уровня должна базироваться на конкретной структуре данных. Поэтому трудно переносить абстракции высокого уровня из одного проекта в другой. Переносится только общая идея. Как следствие, даже простейшие абстракции высокого уровня (вычисление длины списка или поиск в таблице определенного значения) зачастую переписываются заново для каждого нового приложения с учетом конкретных структур данных.

Рассмотрим несколько определений, отражающих различные нюансы программного полиморфизма.

Полиморфизм — это возможность одного и того же программного кода вести себя по-разному в зависимости от того, с экземпляром какого класса он имеет дело. В этом случае программный код оперирует только лишь интерфейсом базового класса, а его потомки, по-разному реализуя методы этого интерфейса, обеспечивают и различное поведение.

В языках программирования полиморфный объект — это сущность (переменная, аргумент функции), хранящая во время выполнения программы значения различных типов.

Полиморфные функции — это такие функции, которые имеют полиморфные аргументы.

Полиморфизм — возможность объектов или функций с одинаковой спецификацией иметь различную реализацию. Язык программирования поддерживает полиморфизм, если классы или функции с одинаковой спецификацией могут иметь различную реализацию, например, реализация класса может быть изменена в процессе наследования.

Последний вопрос, на который стоит ответить, — как найти то место в программном коде, в котором целесообразно использовать полиморфизм? Выше уже частично был дан ответ на этот вопрос: в тех местах, где нам хочется применить условную логику, заключающуюся в проверке типа класса, и в зависимости от этого реализовать по-разному одно и то же действие. Встретив такое место в коде, необходимо подумать: а не лучше ли в данном случае перейти от условной логики к полиморфизму?

Сила полиморфизма состоит в том, что он позволяет записывать алгоритмы высокого уровня лишь однажды и затем повторно их использовать с различными абстракциями низкого уровня. Даже относительно сложные алгоритмы могут быть записаны в виде схемы или шаблона и применяться во многих приложениях.

Контрольные вопросы

1. Сформулируйте назначение полиморфизма.
2. Как используется обобщенный указатель совместно с приведением типов?
3. Как используется обобщенное имя функций с различными наборами аргументов?
4. Как используется типовый параметр с последующей подстановкой конкретного типа?
5. Как абстракции высокого и низкого уровней должны сочетаться в реализации алгоритмов?

9.1.2. Виды полиморфизма

Различают два вида полиморфизма: полиморфизм статический (полиморфизм компиляции) и полиморфизм динамический (полиморфизм времени исполнения программы). Как обычно, если связь вызова функции с ее реализацией или подстановка конкретного типа данных на место обобщенного типа

данных могут быть выполнены на этапе трансляции, то такой полиморфизм считается статическим. В противоположность статическому полиморфизму динамический полиморфизм допускает выполнение тех же действий только во время выполнения программы, т. е. в динамике. Часто можно встретить утверждение о том, что динамический полиморфизм является истинным полиморфизмом, а статический полиморфизм таким не является. Однако если назначением полиморфизма является придание программе свойства модифицируемости, то можно утверждать, что оба вида полиморфизма служат одной цели.

Существует несколько разновидностей полиморфизма. Впервые классификацию разновидностей полиморфизма осуществил Кристофер Стречи (Christopher Strachey) в 1967 году: две наиболее различные из них он назвал *параметрический полиморфизм* и ad hoc *полиморфизм*[33].

Параметрический полиморфизм подразумевает исполнение одного и того же кода для всех допустимых типов аргументов, тогда как ad hoc полиморфизм подразумевает исполнение потенциально разного кода для каждого типа или подтипа аргумента.

Бьерн Страуструп (Bjarne Stroustrup) определил полиморфизм как «*один интерфейс — много реализаций*», это относится к ad hoc полиморфизму, но, как уже было замечено, ad hoc полиморфизм не является истинным полиморфизмом (см. также рассуждения в п. 4.10).

Если параметру функции может быть сопоставлен ровно один тип, то такая функция называется *мономорфной*. Многие языки программирования предоставляют синтаксический механизм для назначения нескольким мономорфным функциям единого имени (идентификатора функции). В этом случае в исходном коде становится возможным осуществлять вызов функции с фактическими параметрами разных типов, но в скомпилированном коде фактически происходит вызов *различных* функций. Такая возможность получила название *перегрузка функций*. Стречи отнес такую возможность перегрузки функций к ad hoc *полиморфизмому*.

Если параметру функции можно сопоставить данные более чем одного типа, то такая функция называется *полиморфной*. Разумеется, с каждым фактическим значением может быть связан лишь один тип, но полиморфная функция рассматривает параметры на основе внешних свойств, а не их собственной организации и содержания. Стречи отнес возможность использования полиморфных функций к *параметрическому полиморфизму*.

В дальнейшем классификацию полиморфизма уточнил Лука Карделли (Luca Cardelli), выделив четыре разновидности:

- универсальный полиморфизм включает:
  - параметрический полиморфизм;
  - полиморфизм включения (или полиморфизм подтипов);
- ad hoc полиморфизм включает:

[33] Латинский фразеологизм «ad hoc» (буквально «*к случаю*») имеет двойственное значение: 1) спонтанный, непродуманный, сделанный «на коленке»; 2) специальный, устроенный конкретно для данной цели или данного случая.

- перегрузку;
- приведение типов.

В общем случае полиморфизм в объектно-ориентированных языках отражает принцип подстановки.

Далее будут рассмотрены все четыре разновидности полиморфизма.

Контрольные вопросы

1. Чем отличается статический полиморфизм от динамического полиморфизма?
2. Какие виды полиморфизма используются в программировании?

9.2. Универсальный полиморфизм

9.2.1. Параметрический полиморфизм

Как уже говорилось ранее, параметрический полиморфизм считается чистым полиморфизмом: один и тот же объект или функция могут единообразно использоваться в разных контекстах типизации без изменений, приведений типов или любых других проверок времени исполнения или преобразований. Однако для этого требуется некое единообразное представление данных (например, посредством указателей).

Параметрический полиморфизм позволяет определять функцию или тип данных обобщенно, так, что значения обрабатываются идентично вне зависимости от их типа. Параметрически полиморфная функция использует аргументы на основе поведения, а не значения, апеллируя лишь к необходимым ей свойствам аргументов, что делает ее применимой в любом контексте, где тип объекта удовлетворяет заданным требованиям поведения. Таким образом, реализуется концепция полиморфизма.

Параметрический полиморфизм доступен в объектно-ориентированных языках программирования, где для его обозначения часто используется термин «обобщенное программирование». В частности, в языках C и C++ нетипизированный параметрический полиморфизм традиционно обеспечивается за счет использования нетипизированного указателя void*.

Проиллюстрируем использование параметрического полиморфизма с нетипизированным указателем void* на примере задачи о сортировке массива. По условию полиморфизма задача формулируется так: разработать программу сортировки элементов массива произвольного типа[34].

Начнем с полиморфной функции сортировки:

```
void Sort (void **mas, int mas_size, int elem_size, bool (*more)(void*,void*), int
(*swap)(void*,void*))
{
   bool boolVContinueCycle = true;
   while (boolVContinueCycle)
      {
```

[34] По условию задачи сортировки тип элементов массива должен отвечать требованию упорядоченности.

```
        boolVContinueCycle=false;
        for (int i=0; i < mas_size/elem_size – 1; i++)
    {
            void * e1=mas+(elem_size * i)/4;                    // вычисляются адреса
            void * e2=mas+(elem_size* i+elem_size)/4;      // элементов в массиве
            if (more(e1,e2))
              {
                 swap(&e1,&e2);
                 boolVContinueCycle = true;
              }
          }
      }
 }
```

В функции обобщенно реализуется алгоритм сортировки массива методом перестановок. Параметрами функции являются: подлежащий сортировке массив **mas**, размер **mas_size** массива, размер **elem_size** элементов массива, функция **more** для проверки упорядоченности двух рядом стоящих элементов массива и, наконец, функция **swap**, предназначенная для обмена местами двух элементов массива. Параметры **more** и **swap** введены в функцию **Sort** для использования соответствующих функций обратного вызова[35].

Предположим, что нам нужно отсортировать элементы массива, которые представляют собой отрезки числовой оси, заданные двумя числами-координатами. Далее по тексту тип отрезок **TSegment** задается соответствующей структурой. Кроме того, определены: функция **SegmentsMore** для сравнения двух отрезков и функция **SegmentsSwap**, предназначенная для обмена местами в массиве двух отрезков. Параметрами обеих функций являются первый и второй отрезки.

```
struct TSegment { int intFStart; int intFEnd; };
bool SegmentsMore( TSegment* segAFirst, TSegment * segASecond )
{ return abs( segAFirst–>intFEnd – segAFirst–>intFStart ) >
          abs( segASecond–>intFEnd – segASecond–>intFStart ); }
void SegmentsSwap( struct TSegment **segAFirst, struct TSegment **segASecond )
{
  TSegment r; r.intFStart=(*segAFirst)–>intFStart; r.intFEnd=(*segAFirst)–>intFEnd;
  (*segAFirst)–>intFStart=(*segASecond)–>intFStart;
  (*segAFirst)–>intFEnd=(*segASecond)–>intFEnd;
  (*segASecond)–>intFStart=r.intFStart;
  (*segASecond)–>intFEnd=r.intFEnd;
}.
```

Элементы другого типа данных, подлежащих сортировке, задаются перечислимым типом данных. Также определены функция **EnumMore** сравнения и функция **EnumSwap** перестановки элементов.

```
bool EnumMore( TNumbers* enumAFirst, TNumbers* enumASecond )
{
```

[35] Применение функций обратного вызова в другом контексте задачи сортировки массива рассматривалось в п. 5.2.1.

```
   return *enumAFirst > *enumASecond ;
}
void EnumSwap( TNumbers** enumAFirst, TNumbers** enumASecond )
{
   TNumbers r; r=**enumAFirst;
   **enumAFirst=**enumASecond;
   **enumASecond=r;
}.
```

Теперь рассмотрим решение обеих задач сортировки:

```
TSegment arSegments[]={ {2,5}, {4,3}, {9,3}, {6,8} };
 void ** ptrVSegs=(void **) arSegments;
 int intVarSegmentsSize=sizeof(arSegments);
 int intVSegmentSize=sizeof(struct TSegment);
 Sort (ptrVSegs,intVarSegmentsSize,intVSegmentSize,
            (bool (*)(void*,void*))SegmentsMore,(int (*)(void*,void*))SegmentsSwap);
enum TNumbers { one, two , three, four, five};
TNumbers nums[]={four, two, five, three, one};
void ** ptrNums=(void **) nums;
int intVNumsSize=sizeof(nums);
int intVEnumSize=sizeof(TNumbers);
Sort( ptrNums, intVNumsSize, intVEnumSize,
            (bool (*)(void*,void*))EnumMore,(int (*)(void*,void*))EnumSwap);
```

Содержание последних сегментов программы достаточно очевидно, поэтому оставим его без дополнительных комментариев.

Для иллюстрации на примере задачи о сортировке нами были выбраны несравнимые типы данных. Аналогичным образом можно сортировать массивы других типов данных. Единственным ограничением на типы данных элементов массива является регулярность элементов массива. Это связано со способом получения адресов элементов массива в функции сортировки. Например, отсортировать массив строк не удастся.

Обратим внимание на то, что при использовании других типов данных нам необходимо задать эти типы данных, разработать функцию **...More**() сравнения элементов и функцию **...Swap**() перестановки элементов. В вызов функции **Sort** подставим необходимые значения фактических параметров, и на этом работа с новым типом данных заканчивается. Тело полиморфной функции **Sort** не пересматривается. Таким образом, полиморфная функция **Sort** обрабатывает данные разных типов, не меняя своего текста.

В полиморфной функции **Sort** связь вызова функций обратного вызова с соответствующими реализациями функции выполняется в процессе исполнения программы, поэтому данный вид полиморфизма является динамическим полиморфизмом. Считается, что в данном случае речь идет об истинном полиморфизме.

Контрольные вопросы

1. Объясните принцип параметрического полиморфизма.
2. Почему параметрический полиморфизм можно назвать истинным полиморфизмом?

3. Как обеспечивается модифицируемость программ с использованием параметрического полиморфизма?

9.2.2. Полиморфизм подтипов

Полиморфизм подтипов, как будет обосновано далее, является отличительной чертой объектно-ориентированного программирования, так как базируется на концепции наследования.

9.2.2.1. Метафора полиморфизма подтипов

Подтипизация (subtyping), или полиморфизм подтипов (subtype polymorphism), означает, что поведение параметрически полиморфной функции ограничивается множеством типов, связанных в иерархию наследования. Например, если имеются типы Number, Rational и Integer, связанные отношением «является предком»: R={(Number, Rational), (Number, Integer)}, то функция, определенная на типе Number, также сможет принять на вход аргументы типов Integer или Rational, и ее поведение будет идентичным. Действительный тип объекта может быть скрыт и предоставляться лишь по запросу идентификации объекта. На самом деле тип Number может быть абстрактным, т. е. конкретный объект этого типа не обязан существовать.

Отношения подтипизации возможны на самых разных категориях типов: примитивных типов, объектных типов и др. Идея подтипов мотивируется увеличением множества типов, которые могут обрабатываться уже написанными функциями, и за счет этого повышением коэффициента повторного использования кода. Это обеспечивается правилом включения: «*если выражение* е *относится к типу* t_1 *и выполняется* $t_2<t_1$ (t_2 *является предком* t_1), *то* е *принадлежит также и типу* t_2». На самом деле речь здесь идет об автоматическом приведении объектной переменной типа t_1 к объектной переменной типа t_2.

В объектно-ориентированном программировании правило включения реализуется с помощью механизма наследования. Понятие наследования является краеугольным камнем объектно-ориентированного программирования, поэтому в отличие от других видов полиморфизма полиморфизм подтипов можно считать отличительной чертой объектно-ориентированного программирования.

Полиморфизм подтипов в объектно-ориентированном программировании заключается в том, что вызывающий код использует объект, опираясь только на его интерфейс (контракт), не зная при этом фактического типа. Такой подход позволяет подтипам реализовывать свое поведение и таким образом изменять поведение программы без перекомпиляции кода-клиента. Рассмотрим пример:

```
class TAnimal
{
  virtual void String Who()=0;
}
class TCat : public TAnimal
{
  void String Who();
}
class TDog : public TAnimal
```

```
{
  void String Who();
}
void TCat::String Who()
{
 ShowMessage(«Это кот!»);
}
void TDog::String Who()
{
 ShowMessage(«Это собака!»);
}
void WhoIsAnimal(TAnimal * Animal)
{
 Animal–>Who();
}
void Main()
{
 TCat * objCat=new TCat();
 WhoIsAnimal(objCat);
 TDog * objDog=new TDog();
 WhoIsAnimal(objDog);
}
```

Здесь класс **TAnimal** (животное) — это базовый тип, объявляющий интерфейс (контракт). Интерфейс состоит всего из одного метода **Who()** (кто?). Далее у нас есть два дочерних класса: **TCat** (кот) и **TDog** (собака), каждый из которых переопределяет метод **Who()** своим собственным поведением, соответствующим этому классу. Обратим внимание на то, что в классе **TAnimal** метод **Who()** объявлен как абстрактный (**virtual void String Who()=0**), поэтому реализация этого метода в классе **TAnimal** отсутствует.

Метод **WhoIsAnimal(TAnimal*Animal)** является полиморфным: он принимает на вход объект типа **TAnimal** и вызывает метод **Who()**. При этом он не знает ничего о фактическом типе объекта, а пользуется только объявленным интерфейсом. Теперь скажем несколько слов о функции **Main()**. Отправляя в функции **WhoIsAnimal()** экземпляры разных типов(**TCat** или **TDog**), мы получаем разное поведение: выводится разный текст. В первом вызове мы получим сообщение «Это кот!», а во втором вызове, соответственно, «Это собака!».

Внутри полиморфной функции связь вызова функции с телом функции выполняется в процессе исполнения программы, поэтому полиморфизм подтипов считается динамическим полиморфизмом. Полиморфизм подтипов, как и параметрический полиморфизм, считается истинным полиморфизмом.

Свойство модифицируемости рассмотренного кода заключается в следующем. Если мы захотим включить в обработку других животных, то для каждого нового животного нам нужно будет добавить соответствующий класс с переопределением функции **Who()**. Добавив в функцию **Main()** необходимый вызов функции **WhoIsAnimal(),** займемся отладкой измененной программы. Тело полиморфной функции **WhoIsAnimal** не пересматривается. Таким обра-

зом, полиморфная функция **WhoIsAnimal** обрабатывает данные разных типов, не меняя своего текста.

Контрольные вопросы

1. Объясните принцип полиморфизма подтипов.
2. Почему полиморфизм подтипов можно назвать истинным полиморфизмом?
3. Почему полиморфизм подтипов можно считать отличительной чертой объектно-ориентированного программирования?
4. Как обеспечивается модифицируемость программ с использованием полиморфизма подтипов?

9.2.2.2. Реализация полиморфизма подтипов

Пусть у нас имеется некое обобщенное поле для хранения данных — класс **TField**, и три его потомка — для хранения строк (класс **TStringField**), хранения целых (класс **TIntegerField**) и вещественных чисел (класс **TSingleField**):

```
class TField
   {
     protected:
      String virtual GetData()=0;
   };
class TStringField : public TField
   {
      String GetData();
   };
class TIntegerField : public TField
   {
      String GetData();
   };
class TDoubleField : public TField
   {
      String GetData();
   };
```

Классы содержат разнотипные данные и «умеют» сообщать о значении этих данных текстовой строкой при помощи метода **GetData**.

Рассмотрим реализацию методов:

```
String TStringField::GetData()
   {
    return «A»;
   }
String TIntegerField::GetData()
   {
    return IntToStr(5);
   }
String DoubleField::GetData()
   {
```

```
  return FloatToStr(7.3);
}
```

Пусть также имеется метод потомка класса **TForm**, который получает объект в виде параметра и показывает полученную строку:

```
void TForm1::ShowData(TField * Field)
 {
  Label1–>Caption=Field–>GetData();
 }
```

Введем объектные переменные

TStringField * StringField; TIntegerField * IntegerField; TDoubleField * DoubleField;

и, предполагая, что созданы необходимые объекты путем вызова соответствующих конструкторов, рассмотрим вызовы:

```
Form1–>ShowData(StringField);
Form1–>ShowData(IntegerField);
Form1–>ShowData(DoubleField);
```

Нас в первую очередь интересует способ установления связи вызова метода и реализации метода, поэтому рассмотрим соответствующие структуры. Информация об объекте **StringField** представлена на рисунке 9.1.

В методе **ShowData** имеется вызов метода **Field–>GetData();**. Рассмотрим, как будет устанавливаться связь вызова метода **GetData()** с реализацией метода **GetData()**. Напомним, что в динамических методах указанная связь устанавливается динамически, т. е. во время исполнения программы. Для вызова **ShowData (StringField);** последовательность действий следующая (рис. 9.1).

Шаг 1. Формальный параметр **Field** метода **ShowData** получил в качестве значения объект **StringField**. С помощью имени **StringField** получаем доступ к экземпляру класса **TStringField**.

Шаг 2. В таблице динамических методов экземпляра класса **TStringField** выполняется поиск метода **GetData**. Поиск завершился успешно.

Шаг 3. По ссылке осуществляется переход на реализацию метода **GetData()**. Метод исполняется. На форме **Form1** появится значение «*A*».

Структуры объектов, идентифицируемых именами **IntegerField** и **DoubleField**, предлагаем построить самостоятельно. Также самостоятельно необходимо промоделировать установление связи вызова метода **Field–>GetData()** с реализацией. В результате исполнения метода на форме **Form1** в первом случае должно появиться значение «**5**», во втором — «7.3».

Возникает законный вопрос: с какой целью был введен класс **TField**? Попытаемся ответить на этот вопрос.

1. Если уберем класс **TField** из нашего примера, то возникают затруднения в описании параметра метода **ShowData**. Предположим, что заголовок метода будет таким:

void ShowData(TStringField * Field);

тогда вызов **ShowData(StringField)** будет обработан без каких-либо затруднений, а вызовы **ShowData(IntegerField)** и **ShowData(SingleField)** станут недопустимыми из-за несоответствия типов формального параметра в описании процедуры и фактического параметра в вызове процедуры.

2. В нашем случае классы **TStringField** и **TIntegerField** являются потомками класса **TField**, поэтому при описании заголовка как **void ShowData(TField * Field)** формальный параметр **Field** совместим по присваиванию с экземплярами всех классов-потомков **TStringField**, **TIntegerField**, **TSingleField**.

3. Возможно и такое решение, вводим три функции, заголовки которых имеют вид:

```
void ShowData(TStringField * Field);
void ShowData(TIntegerField * Field);
void ShowData(TSingle * Field);
```

однако в этом случае ни о каком динамическом полиморфизме речи быть не может[36].

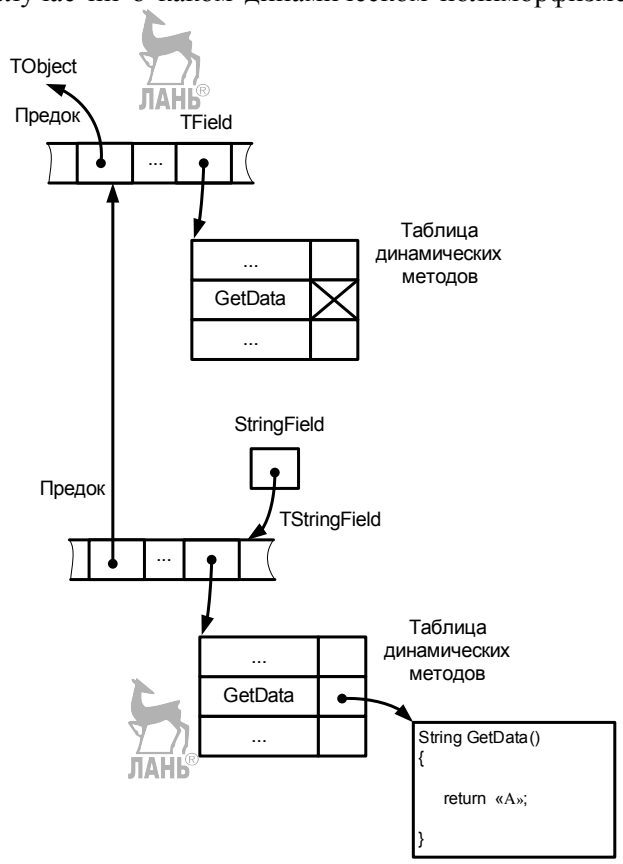


Рис. 9.1
Структура объекта StringField

[36] В этом случае мы имеем дело с полиморфизмом перегрузки.

Таким образом, присутствие класса **TField** является обязательным условием организации полиморфизма подтипов.

Теперь зададимся вопросом: можно ли построить полиморфную операцию подтипов с помощью статических методов? Учитывая, что связь вызова статического метода с реализацией метода устанавливается на этапе трансляции, ответ будет однозначный — нет.

Контрольные вопросы

1. Объясните назначение обобщенного поля **TField** в примерах этого параграфа.
2. Можно ли построить полиморфную операцию с помощью статических методов?

9.2.2.3. Общая схема реализации полиморфизма подтипов

Обратимся к рисунку 9.2. Мы хотим, чтобы класс **TClassParent** и метод **Poly** были неизменными, а классы **TClassA**, …, **TClassZ** могли бы исчезать, появляться, изменяться. И эти изменения никак не должны влиять на класс **TClassParent** и метод **Poly**. Наш замысел реализуется следующим образом (рис. 9.2):

1) в класс **TClassParent** помещаем виртуальные методы (**Method1**, …, **MethodN**).

```
class TClassParent
   {
      void virtual Method1();
            …
      void virtual MethodN();
   };.
```

2) в классы-потомки класса **TClassParent**: **TClassA**, …, **TClassZ** помещаем те же методы, что означает переопределение каждого одноименного метода предка в каждом классе в соответствии с индивидуальным поведением.

```
class TClassA : public TClassParent
   {
      void Method1();
         …
      void MethodN();
   };
      …
class TClassZ : public TClassParent
   {
      void Method1();
         …
      void MethodN();
   };
```

3) в полиморфный метод **Poly** в качестве формального параметра помещаем переменную класса **TClassParent**, а в тело метода — вызовы методов **Method1()**, …, **MethodN()** с использованием параметра в качестве владельца методов.

```
void Poly (TClassParent * Parameter);
  {
   Parameter–>Method1();
      …
   Parameter–>MethodN();
  }
```

4) в текст программы помещаем вызовы полиморфного метода с различными значениями параметра.

```
Poly(ObjectA);
  …
Poly(ObjectZ);
```

где **ObjectA**, …, **ObjectZ** экземпляры, соответственно, классов **TClassA**, …, **TClassZ**.

Каждый вызов полиморфного метода с одним из параметров **ObjectA**, …, **ObjectZ** позволяет использовать в качестве исполняемой реализации методов **Method1()**, **…, MethodN()** тот метод, который предусмотрен в соответствующих объектам **ObjectA, …, ObjectA** классах **TClassA**, **…**, **TClassZ**.

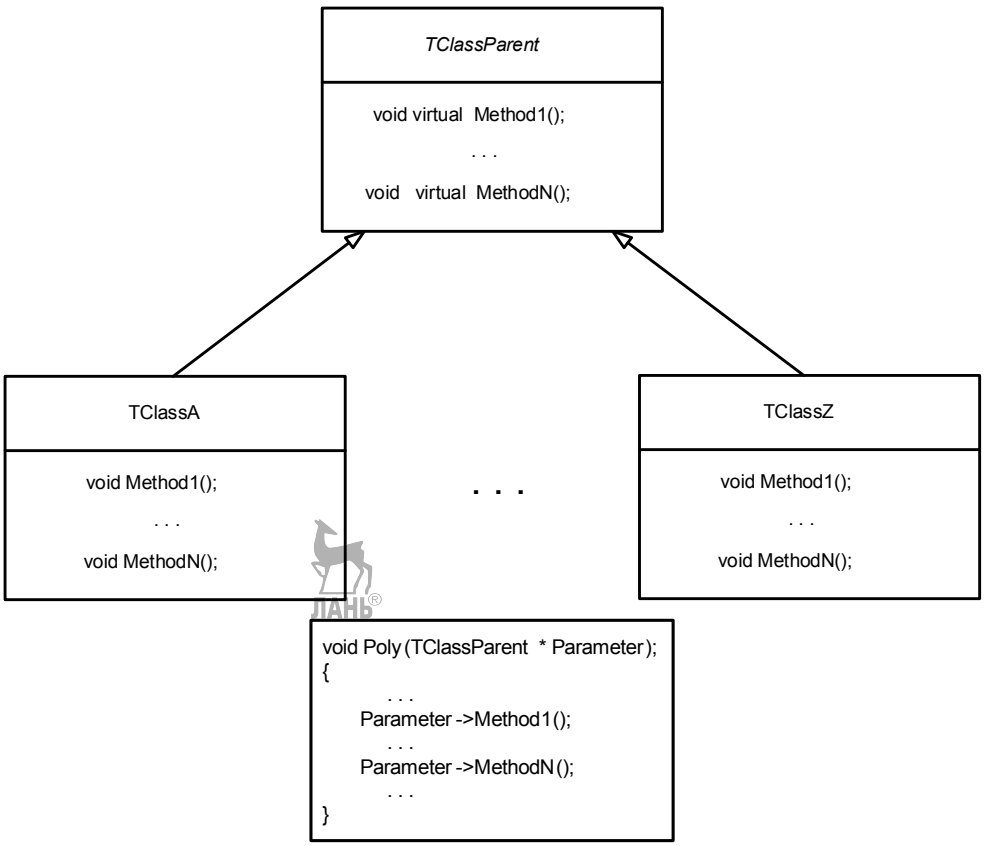


Рис. 9.2
Общая схема реализации полиморфизма

Контрольные вопросы

1. Воспроизведите общую схему реализации полиморфизма.
2. Постройте структуры объектов **ObjectA, …, ObjectZ**.
3. На построенных структурах объектов покажите процесс установления связи вызова методов **Parameter–>Method1() …, Parameter–>MethodN()** с их реализацией.

9.2.2.4. Пример использования полиморфизма подтипов

Рассмотрим задачу.

Задача. На форме расположены две фигуры: круг и квадрат. Необходимо организовать перемещение фигур в пределах клиентской области экранной формы. Перемещение каждой фигуры осуществляется наведением курсора на область фигуры, нажатием левой клавиши мыши, перетаскиванием фигуры на новое место. Заканчивается операция отпусканием нажатой клавиши мыши. В дальнейшем предполагается включение в приложение новых фигур, исключение существующих фигур, изменение конфигурации фигур.

Проектирование. Анализ условия задачи позволяет сделать вывод: в формулировке задачи предусмотрена возможность изменения типов обрабатываемых фигур при неизменности операций над каждой фигурой, поэтому в программу необходимо заложить свойство модифицируемости. Модифицируемость программы будем понимать как возможность локализации будущих изменений текста программы.

Для придания программе свойства модифицируемости введем следующие возможности:

1. Будем использовать метод базовой точки с целью придания единообразия процессу рисования фигур. Суть метода базовой точки заключается в том, что каждая фигура будет рисоваться относительно некоторой точки, называемой базовой точкой. В нашем случае фигура круг будет задана положением центра круга и размером радиуса, а фигура квадрат — положением центра квадрата и размером стороны. Перечисленные параметры будут у нас исходными данными для построения алгоритма прорисовки каждой фигуры. Теперь процесс перемещения фигуры будет осуществляться многократным применением трех действий: затирание существующей фигуры, пересчет положения базовой точки и, наконец, рисование фигуры относительно нового положения базовой точки.

2. Будем использовать полиморфизм подтипов с целью обобщения процесса рисования фигур, обобщение процесса рисования фигур будем понимать как использование единой операции для прорисовки известных и пока еще неизвестных фигур. В терминах языка программирования прием выглядит так: для рисования различных фигур необходимо ввести полиморфный метод.

Результат разработки структуры классов приведен на рисунке 9.3.

Поясним изображенные на рисунке 9.3 основные структурные решения.

Вводим пять классов:

— **TFormFigures** — моделирует экранную форму, в клиентской области которой будут происходить все события нашей задачи;

— **TBasePoint** — моделирует базовую точку;

— **TFigure** — моделирует обобщенное представление фигуры;
— **TCircle** — моделирует круг;
— **TSquare** — моделирует квадрат.

Для рисования круга и квадрата мы будем использовать стандартные методы **Ellipse** и **Rectangle**, принадлежащие компоненту **Canvas** класса **TForm**, потомком которого является наш класс **TFormFigures**. Отношения двух последних классов на рисунке 9.3 не показаны ввиду очевидности.

Класс **TBasePoint** содержит единственное свойство **Location**, предназначенное для фиксации положения базовой точки в клиентской области формы **TFormFigures**. Положение базовой точки будет устанавливаться из класса **TFormFigures** с помощью средств слежения за передвижением курсора, поэтому квантор видимости свойства **Location** имеет значение «+» — общедоступный.

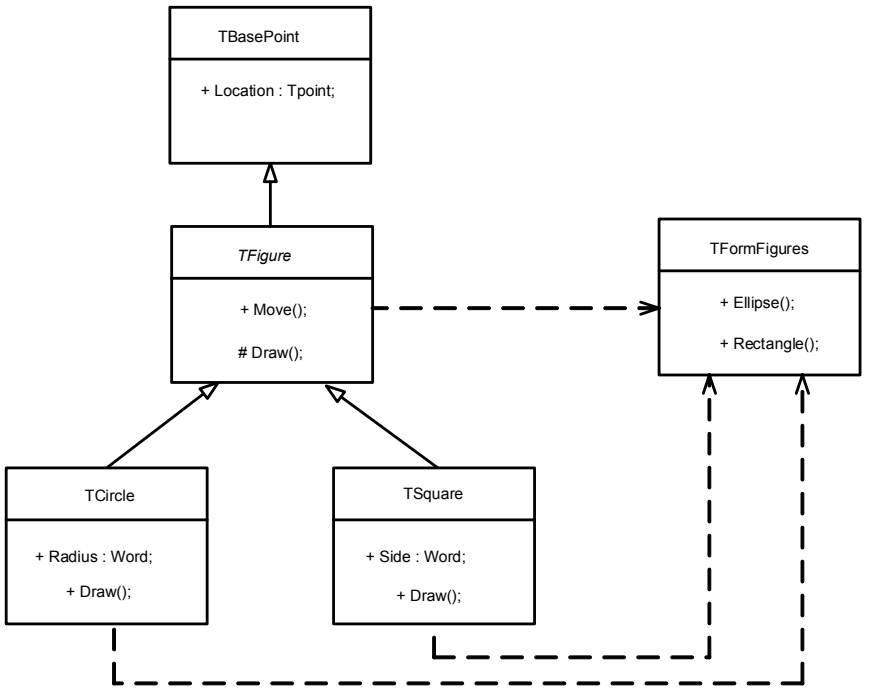


Рис. 9.3
Диаграмма классов задачи о перемещении фигур по форме

Класс **TFigure** включает два базовых для нашей задачи метода: **Move** и **Draw**, первый предназначен для перемещения фигуры в плоскости моделирования, а второй — для рисования фигуры. Метод **Move** имеет квантор видимости «общедоступный», так как цикл перемещения (затирание существующей фигуры, пересчет положения базовой точки и, наконец, рисование фигуры относительно нового положения базовой точки) целесообразно выполнять в классе **TFormFigures**. Метод **Draw** имеет квантор видимости «защищенный», так как этот метод будет всегда переопределяться в потомках.

Классы **TCircle** и **TSquare** содержат индивидуальные для каждой фигуры свойства и поведение. Обратим внимание на то, что одноименный метод **Draw**

содержится в классе **TFigure**. Установка значений свойств и вызов методов классов **TCircle** и **TSquare** будут осуществляться в классе **TFormFigures**, этим объясняется применение квантора видимости «общедоступный».

Рассмотрим теперь отношения между классами.

Отношения (**TBasePoint**, **TFigure**), (**TFigure**, **TCircle**), (**TFigure**, **TSquare**) являются обобщением. В соответствии с правилами языка UML стрелка на рисунке 9.3 во всех случаях направлена к предку. Причины выбора этого вида отношений достаточно очевидны.

Сложнее выбрать отношения (**TFormFigures**, **TFigure**), (**TFormFigures**, **TCircle**), (**TFormFigures**, **TSquare**), которые на рисунке 9.3 изображены как отношение использования (клиент-сервер).

Приступим теперь к написанию кода программы. Сначала запишем интерфейсную часть программы:

```
class TBasePoint
  {
          private:
            TPoint pntFLocation;
            void SetLocation(TPoint pntANew);
          public:
           __property TPoint pntPLocation={read= pntFLocation, write=SetLocation};
        };
   class TFigure : public TBasePoint
  {
     public:
           void virtual Move(TFigure * Sender, TPoint pntAIncrement);
     protected:
           void virtual Draw()=0;
  };
   class TCircle : public TFigure
   {
      private:
           Word wrdFRadius;
           void SetRadius(Word wrdANew);
      public:
          __property Word wrdPRadius={read= wrdFRadius, write=SetRadius};
           void Draw();

   };
   class TSquare : public TFigure
   {
      private:
           Word wrdFSide;
           void SetSide(Word wrdANew);
     public:
          __property Word wrdPSide={read= wrdFSide, write=SetSide};
           void Draw();
   };
```

```
class TfrmFigures : public TForm
{
    __published:
        void __fastcall FormCreate(TObject * Sender);
        void __fastcall FormPaint(TObject * Sender:);
        void __fastcall FormMouseDown(TObject * Sender, TMouseButton But-
ton, TShiftState Shift, int X, int Y);
        void FormMouseMove(TObject * Sender, TShiftState Shift, int X, int Y);
        void FormMouseUp(TObject * Sender, TMouseButton Button, TShift-
State Shift, int X, int Y);
    private:
        TFigure * objFigure;
        TPoint pntMousePosition;
      };
```

Теперь построим реализацию методов:

```
void TFigure::Move(TFigure * Sender, TPoint * pntAIncrement);
{
   TPoint pntVNewCoordinate;
   pntVNewCoordinate.X=Sender–>pntPLocation.X+pntAIncrement.X;
   pntVNewCoordinate.Y=Sender–>pntPLocation.Y+pntAIncrement.Y;
   Sender–>pntPLocation=pntVNewCoordinate;
   frmFigures–>Invalidate();
};

void TCircle::Draw()
{
  Word x0, y0, x1, y1;
  x0=pntPLocation.X – wrdRadius;
  y0=pntLocation.Y – wrdRadius;
  x1=pntLocation.X+wrdRadius;
  y1=pntLocation.Y+wrdRadius;
 frmPolymorphism–>Canvas.Ellipse(x0, y0, x1, y1);
};

void TSquare::Draw()
{
 Word x0, y0, x1, y1;
 x0=pntLocation.X – wrdSide/2;
 y0=pntLocation.Y – wrdSide/2;
 x1=pntLocation.X+wrdSide/2;
 y1=pntLocation.Y+wrdSide/2;
 frmPolymorphism–>Canvas–>Rectangle(x0, y0, x1, y1);
};

void TfrmPolymorphism::OnMouseDown(TObject * Sender, TMouseButton Button,
TShiftState Shift,, int X, int Y)
{
  if (Shift == [ssLeft])
```

```
    if (round(sqrt(sqr(objVCircle–>pntLocation.X – X)+sqr(objVCircle–
>pntLocation.Y – Y))) <= objVCircle–>wrdRadius)
          {
         objFigure=objVCircle;
        Cursor=crHSplit;
       }
    else if ((abs(objVSquare–>pntLocation.X – X) <= objVSquare–>wrdSide/2) &&
         (abs(objVSquare–>pntLocation.Y – Y) <= objVSquare–>wrdSide/2))
         {
          objFigure=objVSquare;
          Cursor=crHSplit;
         };
   pntMousePosition.X := X;
   pntMousePosition.Y := Y;
  };
}

void TfrmPolymorphism::OnMouseMove(TObject * Sender, TShiftState Shift, int X,
int Y)
{
  TPoint P;
 if (Cursor == crHSplit)
   {
    P.X=X – pntMousePosition.X;
    P.Y=Y – pntMousePosition.Y;
    objFigure–>Move(objFigure, P);
    pntMousePosition.X=X;
    pntMousePosition.Y=Y;
   }
}

void TfrmPolymorphism::OnMouseUp(TObject * Sender, TMouseButton Button,
                                 TShiftState Shift, int X, int Y)
{
 Cursor=crDefault;
}
```

Контрольные вопросы

1. Назовите особенности задачи, которые требуют применения в программе полиморфизма подтипов.

9.3. Специальный полиморфизм

9.3.1. Перегрузка функций

Перегрузкой функций в языках программирования называется возможность использования одноименных функций различной сигнатуры[37]. Перегруз-

[37] Сигнатуры и полиморфизм операций обсуждались в главе 4.

ка представляет собой синтаксический механизм, позволяющий по единому идентификатору вызывать разные функции.

При трансляции происходит контроль одноименных функций на отличие сигнатур, так как в этом случае транслятор может однозначно определить вызов нужной подпрограммы.

В следующем примере функции **Add** выглядят как реализующие одну и ту же функциональность над разными типами, но компилятор определяет их как две совершенно разные функции.

```
int Add (int x, y) { return x+y; };
String Add (String x, y) { return x+y; };
ShowMessage(IntToStr(Add(2,3));
ShowMessage(Add("пятьдесят ","шесть"));
```

Перегружаемая функция должна отличаться либо арностью, либо типами параметров, либо типом возвращаемых данных. То же самое имя функции используется для различных случаев вызова функции. Это классификация статического полиморфизма, в котором вызов функции будет решен в зависимости от списка аргументов.

Конструкторы, используемые для создания экземпляров объектов, также могут быть перегружены. В языке C++ название конструктора совпадает с именем класса, поэтому введение нескольких одноименных конструкторов приводит к перегрузке. Виды и использование конструкторов рассмотрены в параграфе 6.5.

Контрольные вопросы

1. Объясните принцип полиморфизма перегрузки функций.
2. Почему перегрузку функций нельзя назвать истинным полиморфизмом?
3. Какие еще операции (за исключением арифметических) обычно являются перегруженными в традиционных языках программирования (Pascal и C)?

9.3.2. Приведение типов

Приведение типов представляет собой семантический механизм, осуществляемый для преобразования фактического типа аргумента к ожидаемому функцией типу. Отсутствие указанного преобразования приводит к ошибке типизации. Как следствие, при вызове функции с приведением типа происходит исполнение различного кода для различных типов (предваряющего вызов самой функции).

Приведение типов не является истинным полиморфизмом: кажется, будто оператор принимает значения множества типов, но значения должны быть *преобразованы* к некоторому представлению до того, как он сможет их использовать, так что на самом деле оператор работает лишь над одним типом (т. е. имеет один тип). Более того, тип возвращаемого значения здесь не зависит от типа входного параметра, как в случае параметрического полиморфизма.

Здесь полезно вспомнить, чему вообще может служить приведение типа. Выделим следующие случаи:

— изменить действительное представление данных либо поведение объекта, на который ссылается некоторый указатель. Простейшее приведение такого рода — преобразование целого типа в вещественный тип;

— изменить лишь *интерпретацию* компилятором некоторых данных, не меняя их действительного (физического) представления. Таково, например, приведение типа int к типу unsigned, и наоборот;

— снять ограничения на возможные манипуляции с объектом, накладываемые модификатором const;

— безопасно приводить типы в различных полиморфных иерархиях классов, в том числе с виртуальными базовыми классами.

Давайте разберемся с тем, как типы данных взаимодействуют друг с другом. Существует так называемая иерархия типов, где все типы размещены по старшинству. Для того, чтобы разбираться в преобразовании типов, необходимо всегда помнить порядок типов этой иерархии. Для некоторых стандартных типов данных порядок выглядит следующим образом:

bool, char, unsigned short int, unsigned long int, int, long int и т. д.

Преобразования типов данных могут быть сужающими: больший тип данных в иерархии преобразуется к меньшему типу, безусловно, в этом случае может произойти потеря данных, поэтому с сужающим преобразованием следует быть осторожными. Например, int a=23.5. Расширяющие преобразования ведут к так называемому расширению типа данных от меньшего диапазона значений к большему диапазону. Например, unsigned int a=3 000 000 000; обратим внимание на то, что в обычный int такое число не поместится.

Преобразования могут быть неявными (автоматическими) и явными.

Неявное приведение типов происходит в следующих случаях:

— после вычисления операндов бинарных арифметических, логических, битовых операций, операций сравнения;

— перед выполнением присваивания;

— перед передачей аргумента функции;

— перед возвратом функцией возвращаемого значения;

— после вычисления выражения конструкции **switch** значение приводится к целочисленному типу;

— после вычисления выражений конструкций **if**, **for**, **while**, **do-while** значение приводится к типу **bool**.

Неявное приведение выполняется транслятором по правилам, описанным в стандарте языка. Стандарты многих языков запрещают неявные преобразования. При неявных преобразованиях возможны побочные эффекты. Например, при приведении числа вещественного типа к целому типу дробная часть отсекается (округление не выполняется).

Явное приведение задается программистом в тексте программы с помощью:

— конструкций языка;

— функции, принимающей значение одного типа и возвращающей значение другого типа.

С-стиль приведения типов данных доступен и языке С++, но считается несамодостаточным по сравнению с приведением типов в С++. С-стиль приведения типов данных может быть использован для преобразования любого типа

в любой другой тип, при этом неважно, насколько это небезопасное преобразование, например, преобразование целого числа в указатель типа **int**. Казалось бы, такое преобразование невозможно, однако компилятор выполнит это приведение. И каков получится результат, ему совершенно не важно. Рассмотрим синтаксис явного приведения типов данных согласно С-стилю:

(тип) выражение,

где **тип** — тип данных, к которому преобразуется результат вычисления **выражение**.

По мнению Бьёрна Страуструпа, операция приведения типа в языке С слишком нестрогая. В качестве примера приводится фрагмент кода:

```
struct TData
{
   double data[200];
};
struct TJunk
{
   int junk[100];
};
TData d={2.5e33, 3.5e-19, 20.2e32};
char * pch=(char *) (&d);              // преобразование в строку
char ch=char (&d);                     // преобразование в символ
TJunk * pj=(TJunk *) (&d);             // приведение в указатель на TJunk.
```

Во-первых, какие из этих преобразований имеют смысл? Пока мы не захотим невероятного, ни одно из этих приведений не имеет смысла. Во-вторых, какие из этих приведений допустимы? В языке С допустимы все эти приведения.

В С++ различают явное и неявное преобразование типов данных. Неявное преобразование типов данных выполняет компилятор С++, а явное преобразование данных выполняет сам программист. Во время преобразования действует правило: результат любого вычисления будет преобразовываться к наиболее точному типу данных, из тех типов данных, которые участвуют в вычислении. В качестве примера рассмотрим таблицу 9.1 с преобразованиями типов данных. В таблице представлена операция деления. В качестве целочисленного типа данных возьмем **int**, а вещественный тип данных у нас будет **double**.

Из таблицы 9.1 видно, что при изменении мест переменных различных типов данных результат остается тот же (в нашем случае это делимое и делитель).

Таблица 9.1

Явное и неявное преобразование типов данных в С++

Делимое	Делитель	Результат деления	Пример
int	**int**	**int**	15/2=7
int	**double**	**double**	15/2.0=7.5
double	**int**	**double**	15.0/2=7.5
double	**double**	**double**	15.0/2.0=7.5

Рассмотрим теперь явное преобразование типов данных.

Явное преобразование типов данных необходимо для того, чтобы выполнять некоторые манипуляции, тем самым меняя результат вычисления. Самый простой способ явного преобразования типов данных: допустим, нам необходимо разделить такие числа 15 и 2. Результат тот же, что и в таблице 9.1. Но, если сделать незначительные преобразования, например 15.0/2=7.5, при таком делении число 15 является вещественным, значит и результат будет вещественный. Этот же прием можно было применить к двойке, результат был бы тем же, а можно было сразу к двум числам. Затруднения могут возникнуть, если операнды в выражении будут представлены переменными или другими выражениями.

В C++ предусмотрены четыре операции явного приведения типа.

Первая из них имеет синтаксис:

static_cast <тип_данных**> (**выражение**),**

где **тип_данных** указывает тип данных, к которому необходимо привести результат вычисления выражения, заданного параметром **выражение**. Оператор **static_cast** аналогичен оператору «круглые скобки» с одним исключением: оператор не выполняет приведение указателей на базовые типы к указателям на производные типы (для этого применяется операторы dynamic_cast или reinterpreted_cast, которые рассмотрены далее).

Например, в результате вычисления

```
int i=15;
double j=static_cast <double> (i)/2;
```

получим значение j=7.5.

В последнем примере переменная **i** не преобразуется в тип данных **double**, а создается временная копия переменной **i** с типом данных **double**.

Операция **static_cast** может быть использована для преобразования одного типа в другой, но она не должна быть использована для выполнения недопустимого преобразования, например, преобразование значения в указатель или наоборот. Рекомендуется пользоваться операцией **static_cast**, а не С-стилем приведения, потому что **static_cast** ограничивает недопустимое приведение типов и, следовательно, безопаснее.

Рассмотрим пример.

```
class TA
  {
        public:
          int iA;
  };
  class TB : public TA
  {
 public:
  int iB;
  void f (TA* pA, TB* pB)
   {
        TA* pA1=static_cast<TA*> (pB);   // Безопасное приведение, TB всегда
```

```
                                        // содержит все TA
    TB* pB1=static_cast<TB*> (pA);   // Не безопасно, TB может иметь поля
                                        // и методы, которые не находятся в TA
    pA1–>iA=4;
    pB1–>iA=5;
    pB1–>iB=6;
 }
};
TA * a=new TA();  a–>iA=1;
TB * b=new TB();  b–>iA=2; b–>iB=3;
b–>f(a, b);
```

В результате исполнения этого фрагмента кода переменная **iA** из класса **TA** получит значение 5, а переменная **iB** из класса **TB** будет равна 3. Первое приведение верно, потому что восходящее преобразование может быть выполнено явно. Второе преобразование, из указателя базового класса в указатель производного класса, не может быть выполнено без явного приведения типа. Поскольку восходящее по иерархии типов преобразование может быть выполнено без приведения типа, операцию **static_cast** можно использовать для нисходящего по иерархии приведения. Однако здесь нужно быть очень осторожным.

Операция **static_cast** безопасна в том случае, если **тип_данных** может быть неявно преобразован в тип, который имеет **выражение** (**тип_данных** и **выражение** — это обозначения из определения синтаксиса операции **static_cast**).

В случае использования явного преобразования появляются возможности повышения защищенности программы. Например, допустимое традиционное преобразование

```
int * x=(int *) 176;
```

становится невозможным уже на этапе трансляции при использовании операции **static_cast**:

```
int * y=static_cast<int *>(176);.
```

Преобразование **static_cast** рекомендуется использовать в следующих случаях:

— приведение целых типов к вещественным, и наоборот;

— преобразование значений типа с меньшей разрядностью к значению типа с большей разрядностью, и наоборот (увеличение и уменьшение разрядности);

— отключение предупреждений компилятора вида «Возможная потеря точности» при преобразовании с уменьшением разрядности;

— приведение указателей к типу **void ***, и наоборот;

— выбор одной из нескольких перегруженных функций;

— приведение указателей на производные типы к указателям на базовые типы, и наоборот;

— явный вызов конструктора с одним аргументом или перегруженной операции приведения типа;

— приведение типа в шаблонах (компилятор уже при специализации шаблона решает, какие операции использовать).

Ошибки возможны в следующих случаях:

— приведение к неправильному типу;

— отсутствие необходимого приведения.

Вторая операция имеет имя **dynamic_cast**. Синтаксис операции имеет вид:

dynamic_cast <тип_данных**> (**выражение**),**

где **тип_данных** — указывает тип данных, к которому необходимо привести результат вычисления выражения, заданного параметром **выражение**.

Операция **dynamic_cast** имеет смысл только применительно к членам класса иерархии наследования типов. Динамическое приведение типов данных может быть использовано для безопасного приведения указателя (или ссылки) на суперкласс, в указатель (или ссылку) на подкласс в иерархии классов. Если динамическое приведение типов недопустимо, так как реальный тип объекта указывает не на тот тип подкласса, приведение типов не выполнится.

При приведении указателя, в случае неудачи, **dynamic_cast** возвращает нулевой указатель **NULL**. Такое поведение обеспечивает быстрый способ определения, является ли данный объект потомком динамического типа.

Важной особенностью динамического приведения типов **dynamic_cast** является то, что если невозможно выполнить преобразование указателей, то возвращается значение **NULL;** а если невозможно выполнить преобразование ссылок, то возникает исключительная ситуация **std::bad_cast** (для этого необходимо включить заголовочный файл **typeinfo**[38]). В результате может быть выполнена проверка успешности такого преобразования.

Динамическое приведение типов **dynamic_cast** наиболее часто используется для динамического приведения типов **TObject *** или **TComponent * Owner**, чтобы гарантировать принадлежность объекта к нужному классу[39]. Например, при размещении компонента в форме может потребоваться узнать способ его размещения (размещен ли он непосредственно на форме или на принадлежащем форме компоненте, например **TPanel**). Для такой проверки необходимо применить следующий код:

```
TForm * OwnerForm=dynamic_cast<TFotm *>(Owner);
if (OwnerForm)
{
     // Обработать, поскольку OwnerForm != NULL
}.
```

Сначала объявляется указатель на нужный тип, а затем ему присваивается результат динамического приведения **dynamic_cast**. Если приведение типов оказалось удачным, то указатель укажет на нужный тип и может быть использован для доступа к нему. Если же приведение завершилось неудачно, то будет получен NULL-указатель, который впоследствии может быть использован в ло-

[38] Обработка исключительных ситуаций (ошибок) рассмотрена в главе 11.

[39] Использование стандартных компонентов, в частности класса TComponent, см. главу 12.

гическом выражении. Аналогично применяется объект **Sender**. Помимо этой ситуации есть много других случаев, когда может потребоваться такое приведение типов. В каждом из них важно четко представить себе цель, а затем определить средства ее достижения.

Оператор **dynamic_cast** будет успешно выполнен только при условии, что разрешено полиморфное приведение типов (т. е. если новый тип можно законно применять к типу объекта, который подвергается этой операции).

Чтобы безопасно пользоваться динамическим приведением, все вызовы **dynamic_cast** должны быть обрамлены в блок **try-catch** (см. главу 11 «Обработка ошибок»).

Операция константного приведения типов данных **const_cast** доступна только в C++. Синтаксис операции имеет вид:

const_cast <тип_данных>(константная_переменная).

Константное приведение используется, чтобы константную переменную преобразовать в не константную переменную. При этом не константным становится возвращаемое значение операции **const_cast**, а не сама переменная.

Следующий пример преобразует константный указатель на символьную строку в не константный указатель на эту же строку.

```
void f (char *);          // прототип функции с не константным параметром
    ...
const char * str=«Строка»;      // константная строка
f(const_cast<char *>(str));
```

Операция приведения типов данных **reinterpret_cast**. Синтаксис имеет традиционную для операций приведения структуру:

reinterpret_cast< тип_данных >(выражение);

Операция **reinterpret_cast** доступна только в C++ и является наименее безопасной формой приведения типов данных в C++, она позволяет интерпретировать значение в другой тип данных с сохранением битового представления. Операция **reinterpret_cast** не должна быть использована для приведения иерархии классов или преобразования константных переменных.

Рассмотрим пример использования этой операции приведения: чтобы преобразовать целое значение в указатель, нужно написать следующее:

int * x=reinterpret_cast< int *>(555);

Возможные ошибки. Объект, возвращаемый выражением **выражение**, может не иметь типа **тип_данных**. Нет никакой возможности проверить это, всю ответственность за корректность преобразования программист берет на себя.

Операцию **reinterpret_cast** можно применить для того, чтобы изменить интерпретацию объекта без действительного преобразования данных.

Целевой тип **тип_данных** может быть типом ссылки, указателя, целым, перечислимым или вещественным типом.

Если целевой тип **тип_данных** является типом указателя или ссылки, то **выражение** может быть указателем или ссылкой, а также числовой (вещест-

венной, целой, перечислимой) переменной; когда целевым типом является числовой тип, то операнд может быть указателем или ссылкой.

Все рассмотренные операции явного преобразования типа можно проделать и с помощью обычных операций приведения, однако последние малонадежны. Тут при опечатках могут происходить совершенно дикие преобразования, и компилятор не выдает даже предупреждающих сообщений. Специальные же операции имеют более корректный вид и явно показывают, что вы делаете.

Контрольные вопросы

1. Сформулируйте назначение приведения типов.
2. Какие имеются виды приведения типов?
3. Чем С стиль приведения типов отличается от С++ стиля приведения типов?
4. Какое назначение операции **static_cast**?
5. Какое назначение операции **dynamic_cast**?
6. Какое назначение операции **const_cast**?
7. Какое назначение операции **reinterpret_cast**?

9.4. Шаблоны функций и классов

Шаблоны, которые называют иногда *родовыми* или *параметризованными типами*, позволяют создавать (конструировать) семейства родственных функций и классов. В начале этого параграфа рассмотрены канонические приемы использования шаблонов. В последнем подпараграфе рассмотрен пример, который учитывает особенности реализации инструмента шаблонов в среде C++Builder.

9.4.1. Шаблоны функций

Целью введения шаблонов функций является автоматизация создания функций, которые могут обрабатывать разнотипные данные. В отличие от механизма перегрузки, когда для каждого набора формальных параметров определяется своя функция, шаблон семейства функций определяется один раз, но это определение параметризуется. Параметризовать в шаблоне функций можно все элементы сигнатуры, т. е. тип возвращаемого функцией значения и типы любых параметров, количество и порядок размещения которых должны быть фиксированы. Для параметризации используется список параметров шаблона.

В определении шаблона семейства функций используется служебное слово **template**. Для параметризации используется список формальных параметров шаблона, который заключается в угловые скобки **<>**. Каждый формальный параметр шаблона обозначается служебным словом **class**, за которым следует имя параметра (идентификатор). Пример определения шаблона функций, вычисляющих абсолютные значения числовых величин разных типов:

```
template <class types> types abs (types x) { if(x>0) return(x); else return(-x);}
```

Шаблон семейства функций состоит из двух частей — заголовка шаблона:

```
template <СписокПараметровШаблона>
```

и тела определения функции. В списке параметров шаблона слово **class** может относиться как к базовому, так и к абстрактному типу данных. При использовании базовых типов данных вместо служебного слова **class** можно использовать также служебное слово **typename**. Таким образом, список параметров шаблона **<class T>** просто означает, что **T** представляет собой тип, который будет задан позднее. Так как **T** является параметром, обозначающим тип, шаблоны иногда называют параметризованными типами.

Шаблон семейства функций служит для автоматического формирования конкретных определений функций по тем вызовам, которые транслятор обнаруживает в теле программы. Например, если программист употребляет обращение

```
cout << abs(-10.3);
```

то на основе приведенного ранее шаблона компилятор сформирует следующее определение функции:

```
double abs (double x) { if  (x > 0) return(x); else return(-x);}
```

Выполнение именно этой функции вернет в точку вызова в качестве результата вещественное значение **10.3**.

В качестве другого примера рассмотрим шаблон семейства функций для обмена значений двух передаваемых им параметров:

```
template <class T>
        void swap (T* x, T* y)
{
   T z=*x;
   *x=*y;
   *y=z;
}
```

Здесь параметр **T** шаблона функций используется не только в заголовке для спецификации формальных параметров, но и в теле определения функции, где он задает тип вспомогательной переменной **z**.

Если в программе присутствует приведенный ранее шаблон семейства функций **swap()** и имеет место, скажем, следующий вызов этой функции:

```
long int a=4, b=8;
swap (&a, &b);
```

то компилятор сформирует определение функции вида:

```
void swap (long int *x, long int *y)
{
    long int z=*x;
    *x=*y;
    *y=z;
}
```

Затем будет выполнено обращение именно к этой функции и значения переменных ***a***, ***b*** поменяются местами. Если в той же программе присутствуют операторы:

```
double a=2.44, b=66.3;
swap (&a, &b);
```

то сформируется и выполнится функция

```
void swap (double *x, double *y)
{
    double z=*x;
    *x=*y;
    *y=z;
}
```

Рассмотрим теперь программу, используя некоторые возможности функций, возвращающих значение типа «ссылка». Но тип ссылки будет определяться параметром шаблона:

```
#include <iostream>
using namespace std;
// Функция возвращает ссылку на элемент с максимальным значением
template <class type>
type & MaxElement (int n, type *mas)
{
    int im=0;
    for (int i=1; i < n; i++)
       if(mas[im] < mas[i])    im=i;
    return mas[im];
}
// Главная программа
int main()
          {
     // Обработка массива целых чисел
     int n=4;
     int intArray[]={ 10, 20, 30, 15};
     cout << "MaxElement (n, x)=" << MaxElement (n, intArray) << endl;
     for (int i=0; i < n; i++)
     cout << "  intArray [" << i << "]=" << intArray [i];
     MaxElement (n, intArray)=0;          // обнулить максимальный элемент
 cout << endl;
 for (int i=0; i < n; i++)
 cout << "  intArray [" << i << "]=" << intArray [i];
// Обработка массива вещественных чисел
 int m=3;
 double dubArray[]={ 10.3, 20.4, 10.5};
 cout << endl <<"MaxElement (m, dubArray)=" << MaxElement (m, dubArray) << endl;
 for (int i=0; i < m; i++)
 cout << "  dubArray [" << i << "]=" << dubArray [i];
 MaxElement (m, dubArray)=0;        // обнулить максимальный элемент
 cout << endl;
 for (int i=0; i < m; i++)
 cout << "  dubArray [" << i << "]=" << dubArray [i];
```

```
 cin.get();
 return 0;
}.
```

Результатом вызова функции **main()** будет следующий вывод:

```
MaxElement (n, x)=30
   intArray[0]=10  intArray[1]=20  intArray[2]=30  intArray[3]=15
   intArray[0]=10  intArray[1]=20  intArray[2]=0  intArray[3]=15
MaxElement (n, x)=20.4
   dubArray [0]=10.3  dubArray [1]=20.4  dubArray [2]=10.5
   dubArray [0]=10.3  dubArray [1]=0  dubArray [2]=10.5 .
```

В примере используются два разных обращения к функции **MaxElement()**. В одном случае параметр — целочисленный массив и возвращаемое значение — ссылка типа **int**. Во втором случае фактический параметр — имя массива типа **double** и возвращаемое значение имеет тип ссылки на **double**. Обратим внимание на то, что шаблон функции **MaxElement** введен в программу один раз.

По существу, механизм шаблонов функций позволяет автоматизировать подготовку переопределений перегруженных функций. При использовании шаблонов нет необходимости готовить заранее все варианты функций с перегруженным именем. Компилятор автоматически, анализируя вызовы функций в тексте программы, формирует необходимые определения именно для таких типов параметров, которые использованы в обращениях. Дальнейшая обработка выполняется так же, как и для перегруженных функций.

Параметры шаблонов. Параметры шаблона являются его формальными аргументами, а типы тех аргументов, которые используются в конкретных обращениях к функции, служат фактическими аргументами шаблона. По ним выполняется параметрическая настройка и с учетом этих типов генерируется конкретный текст определения функции. Говоря о шаблоне семейства функций, обычно имеют в виду формальные аргументы (параметры) шаблона.

Перечислим основные свойства параметров шаблона:

— имена параметров шаблона должны быть уникальными во всем определении шаблона;

— список параметров шаблона функции не может быть пустым, так как при этом теряется возможность параметризации и шаблон функций становится обычным определением конкретной функции;

— в списке параметров шаблона функций может быть несколько параметров. Каждый из них должен начинаться со служебного слова **class**. Например, допустим такой заголовок шаблона:

template <class type1, class type2>,

соответственно, неверен заголовок:

template <class type1, type2>.

— недопустимо использовать в заголовке шаблона параметры с одинаковыми именами, т. е. ошибочен такой заголовок:

template <class type, class type>.

— имя параметра шаблона (в примерах — type1, type2) имеет в определяемой шаблоном функции все права имени типа, т. е. с его помощью могут специализироваться формальные параметры, определяться тип возвращаемого функцией значения и типы любых объектов, локализованных в теле функции. Имя параметра шаблона видно во всем определении и скрывает другие использования того же идентификатора в области, глобальной по отношению к данному шаблону функций. Если внутри тела определяемой функции необходим доступ к внешним объектам с тем же именем, нужно применять операцию изменения области видимости (оператор разрешения контекста).

Рассмотрим пример.

```
#include <iostream>
using namespace std;
int N=0;      // глобальная переменная
template <class N>
N maxim (N x, N y)
{
     N a=x;
      cout << endl <<"N=" << ++::N;
      if (a < y) a=y;
      return a;
}
int main()
{
    int intA=12, intB=42;
    cout << endl << maxim (intA, intB);
    double dubC=66.3, dubD=222.4;
    cout << endl << maxim (dubC, dubD);
   cin.get();
    return 0;
}.
```

ЛАНЬ®

Результат выполнения функции main() имеет вид:

```
N=1
42
N=2
222.4
```

Одно имя нельзя использовать для обозначения нескольких параметров одного шаблона, но в разных шаблонах функций могут быть одинаковые имена у параметров шаблонов (аналогично формальным параметрам при определении обычных функций). Поскольку видимость параметра шаблона заканчивается в конце определения шаблона, соответствующий идентификатор свободен для последующего использования, в том числе и в качестве имени параметра другого шаблона.

Все параметры шаблона функций должны быть обязательно использованы в спецификациях параметров определения функции. Таким образом, будет ошибочным такой шаблон:

template <class A, class B, class C>
B func (A n, C m) {B value; ...}.

В данном случае остался неиспользованным параметр шаблона с именем B. Его применение в качестве типа возвращаемого функцией значения и для определения объекта **value** в теле функции недостаточно. Определяемая с помощью шаблона функция может иметь любое количество не параметризованных формальных параметров. Может быть не параметризовано и возвращаемое функцией значение.

Пример. Шаблон определяет семейство функций, каждая из которых подсчитывает количество нулевых элементов одномерного массива.

```
#include <iostream>
using namespace std;
template <class D>
long int count0 (int, D*); //Прототип шаблона
int main()
{
   int A[]={1,0,6,0,4,10};
   int n=sizeof(A)/sizeof(A[0]);
   cout << "count0(n, A)=" << count0(n, A) << endl;
   double X[]={10.0, 0.0, 3.3, 0.0, 2.1, 0.0};
   n=sizeof(X)/sizeof X[0];
   cout << "count0(n, X)=" << count0(n, X) << endl;
   cin.get();
   return 0;
}
template <class T>
long int count0 (int size, T* array)
{
   long int k=0;
   for (int i=0; i < size; i++)
      if (int(array[i]) == 0) k++;
   return k;
}
```

Результат выполнения функции **main()**:

count0(n,A)=2
count0(n,X)=3

В шаблоне функций **count0** параметр T используется только в спецификации одного формального параметра **array**. Параметр **size** и возвращаемое функцией значение имеют явно заданные не параметризованные типы.

Как и при работе с обычными функциями, для шаблонов функций существуют определения и описания. В качестве описания шаблона функций используется прототип шаблона:

template <СписокПараметровШаблона> ЗаголовокФункции;.

В списке параметров прототипа шаблона имена параметров не обязаны совпадать с именами тех же параметров в определении шаблона. Это и продемонстрировано в программе.

При конкретизации шаблонного определения функции необходимо, чтобы при вызове функции типы фактических аргументов, соответствующие одинаково параметризованным формальным аргументам, были одинаковыми. Для определенного выше шаблона функций с прототипом:

```
template < class E >
void swap (E, E);
```

недопустимо использовать такое обращение к функции:

```
int n=4; double d=4.3;
swap (n, d);    // Ошибка в типах параметров
```

Для правильного обращения к такой функции требуется явное приведение типа одного из параметров. Например, вызов

```
swap (double (n) , d); // Правильные типы параметров
```

приведет к конкретизации шаблонного определения функций с параметром типа **double**.

При использовании шаблонов функций возможна перегрузка, как шаблонов, так и функций. Могут быть шаблоны с одинаковыми именами, но разными параметрами. Или с помощью шаблона может создаваться функция с таким же именем, что и явно определенная функция. В обоих случаях распознавание конкретного вызова выполняется по сигнатуре, т. е. по типам, порядку и количеству фактических параметров.

Контрольные вопросы

1. Какова цель введения шаблонов функций?
2. Опишите синтаксис шаблона функций.
3. В чем проявляется полиморфизм при использовании шаблонов функций?

9.4.2. Шаблоны классов

Определим шаблон класса как предписание для создания класса, в котором один или несколько типов либо значений параметризованы.

Аналогично шаблонам функций определяется шаблон семейства классов:

template <СписокПараметровШаблона> ОпределениеКласса.

Шаблон семейства классов определяет способ построения отдельных классов подобно тому, как класс определяет правила построения и формат отдельных объектов. В определении класса, входящего в шаблон, особую роль играет имя класса. Оно является не именем отдельного класса, а параметризованным именем семейства классов. Как и для шаблонов функций, определение шаблона класса может быть только глобальным.

Когда шаблон введен, у программиста появляется возможность определять конкретные объекты конкретных классов, каждый из которых параметри-

чески порожден из шаблона. Формат определения объекта одного из классов, порождаемых шаблоном классов:

ИмяПараметризованногоКласса <ФактическиеПараметрыШаблона>
ИмяОбъекта
(ПараметрыКонструктора);

Например, определить вектор **z**, имеющий восемь вещественных координат типа **double**, можно следующим образом:

vector <double> z (8);

Рассмотрим теперь содержательный пример. Создадим векторный класс (в число данных входит одномерный массив). Какой бы тип ни имели элементы массива (целый, вещественный, с двойной точностью и т. д.), в этом классе должны быть определены одни и те же базовые операции, например, доступ к элементу по индексу и т. д. Если тип элементов вектора задавать как параметр шаблона класса, то система будет формировать вектор нужного типа (и соответствующий класс) при каждом определении конкретного объекта.

Решение задачи имеет вид:

```
#include <iostream>
using namespace std;
template <class T>                  // T-параметр шаблона
class TVector
{
    T *tFData;           // Указатель на одномерный массив
     int intFSize;       // Количество элементов в массиве
public:
    TVector (int);                          // Конструктор
   ~TVector () { delete [] tFData; }     // Деструктор
   T& operator [] (int i) { return tFData[i];} // Расширение действия (перегрузка)
операции []
};
// Определение конструктора класса:
template <class T>
TVector <T>:: TVector (int n=1)
{
   tFData=new T [n];
   intFSize=n;
}
int main()
{
   TVector <int>  X(5);     //Создаем вектор типа int
   TVector <char> C(5);   //Создаем вектор типа char
   for (int i=0; i < 5; i++)
        { X [i]=i; C[i]='A'+i;}
   for (int i=0; i < 5; i++)
         cout << " " << X[i];
   cout << endl;
   for (int i=0; i < 5; i++)
```

```
        cout << " " << C[i];
   cin.get();
   return 0;
}.
```

Результат вызова функции **main()** имеет вид:

```
0  1  2  3  4
A  B  C  D  E.
```

В приведенной программе шаблон семейства классов с общим именем **TVector** используется для формирования двух классов с массивами целого и символьного типов. В соответствии с требованием синтаксиса имя параметризованного класса, определенное в шаблоне (в примере **TVector**), используется в программе только с последующим конкретным фактическим параметром (аргументом), заключенным в угловые скобки. Параметром может быть имя базового или определенного пользователем типа (класса). В данном примере использованы стандартные типы **int** и **char**. Использовать имя **TVector** без указания фактического параметра шаблона нельзя — никакое умалчиваемое значение при этом не предусматривается.

В списке параметров шаблона могут присутствовать формальные параметры, не определяющие тип, точнее — это параметры, для которых тип фиксирован. Для иллюстрации изменим незначительно рассмотренный пример:

```
#include <iostream>
using namespace std;
template <class T, int n=1>     // n-параметр фиксированного типа
class TVector
{
    T * tFData;
     int intFSize;
  public:
    TVector (void)
    {
        tFData=new T [n];
        intFSize=n;
     }
    ~Vector () { delete [] tFData; }
     T& operator [] (int i) { return tFData [i];}
};
int main()
{
   TVector <int, 5> X;
   TVector <char, 5> C;
    for (int i=0; i < 5; i++)
        { X [i]=i; C[i]='A'+i;}
    for (int i=0; i < 5; i++)
        cout << " " << X[i];
    cout << endl;
    for (int i=0; i < 5; i++)
```

```
        cout << " " << C[i];
    cin.get();
    return 0;
}.
```

В качестве аргумента, заменяющего при обращении к шаблону параметр **n**, взята константа. В общем случае может быть использовано константное выражение, однако выражения, содержащие переменные, использовать в качестве фактических параметров шаблонов нельзя.

Использовать имя шаблона класса без указания фактического параметра шаблона нельзя, но если программист считает, что такая конструкция класса затрудняет читабельность кода, то можно воспользоваться оператором **typedef**, например, следующим образом:

```
typedef TVector <int> iTVector;
```

и в дальнейшем работать с типом **iTVector**.

При написании программ описания отдельных функций и классов может быть разбито на несколько файлов, образующих проект. Как обычно, описание классов помещается в заголовочные файлы, а реализация функций — в соответствующий реализационный файл.

Контрольные вопросы

1. Какова цель введения шаблонов класса?
2. Опишите синтаксис шаблона класса.
3. В чем проявляется полиморфизм при использовании шаблонов класса?

9.4.3. Пример использования шаблона

В п. 9.2.2.4 рассмотрен пример полиморфизма подтипов, в котором рассматривается задача перемещения различных фигур в пределах клиентской области экранной формы. Расширим этот пример дополнительными возможностями: по каждой фигуре движется точка. Пользователь имеет возможность регулировать размер фигуры и скорость движения точки. Точку будем моделировать с помощью небольшого круга (эллипса). Уточним нашу терминологию: понятие «фигура» заменим понятием «траектория», а понятию «точка» сопоставим понятие «фигура». Изменение терминологии введено с целью дальнейшего расширения задачи: по некоторой траектории движется некоторая фигура.

Сначала введем класс «абстрактная траектория»:

```
#define intCPointCount   300     // количество точек для рисования траектории
class TAbstractTrajectory
{
        private:
          TPoint pntFCenter;
          void SetCenter(TPoint New){pntFCenter=New;};
          double doubleFScale;
          void SetScale(double New){doubleFScale=New;};
        public:
          __property TPoint pntPCenter={read=pntFCenter,write=SetCenter};
```

```
        __property double doublePScale={read=doubleFScale,write=SetScale};
        void virtual Draw(TCanvas * Canvas)=0;
};
```

Приведенный класс содержит свойства: **pntPCenter** — координаты базовой точки траектории, **doublePScale** — масштаб (размер) траектории. Абстрактный метод **Draw** предназначен для рисования траектории на передаваемой параметром канве.

Теперь рассмотрим один из возможных вариантов траектории — круг:

```
class TCircle : public TAbstractTrajectory
{
      private:
        double doubFInitialAngle;
        int intFRadius;
        TSemiMode enumSemiMode;
        void SetRadius(int New){intFRadius=New;};
      public:
        __property int intPRadius={read=intFRadius,write=SetRadius};
        __fastcall TCircle(int intARadius, TPoint pntACenter, double doubAInitia-
lAngle);
        void Draw(TCanvas * Canvas);
        TPoint AngleToPosition(double doubAAngle);
};
```

Свойство **intPRadius** моделирует радиус круга, единица измерения — пиксел. Конструктор класса содержит параметр **doubAInitialAngle** — начальное значение угла для расчета массива точек при рисовании траектории и вычисления текущего положения точки на траектории. Параметр введен с целью обеспечения возможности дальнейшего развития приложения, например, нам когда-нибудь понадобится траектория как сектор круга (в частном случае — полукруг). Назначение остальных параметров очевидно. Метод **Draw** переопределяет одноименный метод предка. Функция **AngleToPosition** сопоставляет углу положение точки на траектории круга. Отсчет угла ведется от **doubAInitialAngle** против часовой стрелки.

Реализация методов класса **TCircle** имеет вид:

```
__fastcall TCircle::TCircle(int intARadius, TPoint pntACenter, double doubAInitia-
lAngle)
{
intFRadius=intARadius;
pntPCenter=pntACenter;
doubFInitialAngle=doubAInitialAngle;
}
void TCircle::Draw(TCanvas * Canvas)
{
      double doubVAngle=doubFInitialAngle;
      double doubVDelta=2 * M_PI/intCPointCount;
      TPoint p[intCPointCount];
      for (int i=0; i < intCPointCount; i++)
```

```
            {
              p[i]=AngleToPosition(doubVAngle);
              doubVAngle=doubVAngle+doubVDelta;
            }
        Canvas–>Polygon(p,intCPointCount – 1);
}
TPoint TCircle::AngleToPosition(double doubAAngle)
{
        TPoint p;
        p.X=pntPCenter.X+(int)(intFRadius * doublePScale * cos(doubAAngle));
p.Y=pntPCenter.Y – (int)(intFRadius * doublePScale * sin(doubAAngle));
        return p;
}
```

Реализация методов класса **TCircle** очевидна, поэтому оставляем без комментариев.

Класс **TPointFigure** моделирует в нашем примере точку, которая «бежит» по траектории:

```
class TPointFigure
{
     TPoint pntFCenter;
     void SetCenter(TPoint New){pntFCenter=New;};
   public:
     __property TPoint pntPCenter={read=pntFCenter,write=SetCenter};
     void virtual Draw(TCanvas * Canvas);

};
void TPointFigure::Draw(TCanvas * Canvas)
{
  Canvas–>Ellipse(pntFCenter.X – 10, pntFCenter.Y – 10, pntFCenter.X+10,
pntFCenter.Y+10);
}
```

Реализацию класса **TPointFigure** оставляем без комментариев.

Переходим к рассмотрению класса «траектория с бегущей точкой», представляющего в рассматриваемой теме наибольший интерес:

```
template <class TTrajectory> class TTrajectoryWithRunningPoint
{
   private:
       TPointFigure * objFigure;
       TPoint pntFFigureCenter;
       double doubleFInitialFigureAngle;
       double doubFCurrentFigureAngle;
       double doubleFFigureSpeed;
       void SetFigureSpeed(double New) {doubleFFigureSpeed=New;};
       TTrajectory * objTrajectory;
  public:
```

```
    __property double doublePFigure-
Speed={read=doubleFFigureSpeed,write=SetFigureSpeed};
    __fastcall TTrajectoryWithRunningPoint(TCanvas * Canvas,
int intARadius, TPoint pntACenter, double doubAInitialAngle):objTrajectory()
 {
    doubleFInitialFigureAngle=doubAInitialAngle;
    doubFCurrentFigureAngle=doubAInitialAngle;
    objTrajectory=new TTrajectory(Canvas, pntACenter, doubAAngle);
            objTrajectory–> intPRadius=intARadius;
           objTrajectory–> pntPCenter=pntACenter;
    objFigure=new TPointFigure();
    objFigure–>pntPCenter=objTrajectory–
>AngleToPosition(doubFCurrentFigureAngle);
  }
  void TrajectoryDraw(TCanvas * Canvas) {objTrajectory–>Draw(Canvas);};
  void FigureDraw(TCanvas * Canvas) {objFigure–>Draw(Canvas);};
  void SetTrajectoryScale(double New){objTrajectory–>doublePScale=New;};
  double GetTrajectoryScale(){return objTrajectory –>doublePScale};
  void NextFigurePosition()
  {
    doubFCurrentFigureAngle=doubFCurrentFigureAngle+doubleFFigureSpeed;
    if (doubFCurrentFigureAngle > doubleFInitialFigureAngle+2 * M_PI)
          doubFCurrentFigureAngle=doubleFInitialFigureAngle;
    objFigure–>pntPCenter=objTrajectory–
>AngleToPosition(doubFCurrentFigureAngle);
  }
};
```

Класс **TTrajectoryWithRunningPoint** представляет собой, как видно из заголовка, шаблон класса, параметризованный классом **TTrajectory**. Особенностью реализации шаблонов классов в C++Builder является невозможность переносить реализацию методов класса в другие модули, в том числе и в модуль реализации **.cpp**. Перечислим следствия указанной особенности.

Во-первых, реализация всех методов находится в теле описания класса.

Во-вторых, построение конкретного специализированного объекта должно находиться также в одном с шаблоном класса модуле. По этой причине объявление моделируемого шаблоном экземпляра класса **TTrajectory*objTrajectory;** включено в области видимости **private** класса **TTrajectoryWithRunningPoint**. Создание конкретного экземпляра моделируемого абстрактного класса **objTrajectory** происходит в конструкторе класса **TTrajectoryWithRunningPoint**.

В-третьих, возникает необходимость пересылки элементов экземпляра класса **TTrajectoryWithRunningPoint** в экземпляр класса **TTrajectory**. С этой целью введены специальные методы, например, метод **TrajectoryDraw()**.

Рассмотрим, наконец, реализацию методов экранной формы **frmMain**, с помощью которых осуществляется управление приложением и специализация шаблона класса **TTrajectoryWithRunningPoint**. Содержание методов класса приведено в сокращении: выделены действия, которые являются существенны-

ми для рассматриваемой нами задачи. На экранной форме размещены: компонент **TPaintBox * pboxFWorkSpace**, на канве которого выполняется прорисовка траектории и бегущей точки; компонент **TTrackBar * tbarFFigureSpeed** («движок»), с помощью которого пользователь управляет скоростью движения точки по траектории; невидимый компонент **TTimer * Timer**, управляющий динамикой происходящих в приложении процессов.

```
TfrmMain *frmMain;

TTrajectoryWithRunningPoint <TCircle> * objTrajectoryWithRunningPoint;
//TShapeWithRunningPoint <TEllipse> * objTrajectoryWithRunningPoint;
//TShapeWithRunningPoint <TEpicicloid> * objTrajectoryWithRunningPoint;

void __fastcall TfrmMain::FormCreate(TObject *Sender)
{
  TPoint pntVCenter;
   pntVCenter.X=pboxFWorkSpace –>ClientWidth/2;
   pntVCenter.Y=pboxFWorkSpace –>ClientHeight/2;
   int intVRadius=pboxFWorkSpace –>ClientWidth/3;
   objTrajectoryWithRunningPoint=new TTrajectoryWithRunningPoint <TCircle>
                                 (pboxFWorkSpace–>Canvas, intVRadius, pntVCenter, 0);
//  objTrajectoryWithRunningPoint=new TTrajectoryWithRunningPoint <TEllipse>
//                                (pboxFWorkSpace–>Canvas, intVRadius, pntVCenter, 0);
//  objTrajectoryWithRunningPoint=new TTrajectoryWithRunningPoint <TEpicicloid>
//                                (pboxFWorkSpace–>Canvas, intVRadius, pntVCenter, 0);

   objTrajectoryWithRunningPoint–>SetTrajectoryScale((double)(tbarFTrajectorySize–
>Position)/   (double)tbarFTrajectorySize–>Max);
   objTrajectoryWithRunningPoint–>doublePFigureSpeed= (dou-
ble)tbarFFigureSpeed–>Position /    (double)tbarFFigureSpeed–>Max;
}
void __fastcall TfrmMain::FormPaint(TObject *Sender)
{
   objTrajectoryWithRunningPoint–>TrajectoryDraw(pboxFWorkSpace–>Canvas);
   objTrajectoryWithRunningPoint–>FigureDraw(pboxFWorkSpace–>Canvas);
 }
void __fastcall TfrmMain::tbarFTrajectorySizeChange(TObject *Sender)
{
  objTrajectoryWithRunningPoint–>SetTrajectoryScale((double)(tbarFTrajectorySize–>
Position)/   (double) tbarFTrajectorySize–>Max);
  Invalidate();
}
void __fastcall TfrmMain::TimerTimer(TObject *Sender)
{
  objTrajectoryWithRunningPoint–>NextFigurePosition();
  Invalidate();
}
void __fastcall TfrmMain::tbarFFigureSpeedChange(TObject *Sender)
```

```
{
  objTrajectoryWithRunningPoint–>doublePFigureSpeed= (double)tbarFFigureSpeed–>
Position/(double)tbarFFigureSpeed–>Max;
}
```

В начале текста реализации экранной формы **frmMain** определяются конкретные объекты конкретных классов, каждый из которых параметрически порожден из шаблона. В нашем случае это экземпляры классов **TCircle**, **TEllipse**, **TEpicicloid**, описание двух последних заключено в комментарий, так как в нашем примере их реализация не рассматривалась (реализовать самостоятельно).

Обсудим наиболее существенные моменты реализации метода **FormCreate**. После вычисления параметров конструктора **TTrajectoryWithRunningPoint** создается конкретный объект **objTrajectoryWithRunningPoint**, специализированный классом **TCircle**. Аналогичным образом могут быть созданы экземпляры специализированных классов **TEllipse**, **TEpicicloid**.

Реализация других методов достаточно очевидна и не нуждается в пояснениях.

Контрольные вопросы

1. Какие особенности имеет реализация шаблонов в среде C++Builder?

Задачи

В следующих заданиях требуется реализовать абстрактный базовый класс, определив в нем виртуальные функции. Эти функции переопределяются в производных классах. В базовых классах должны быть также объявлены виртуальные функции ввода/вывода, которые реализуются в производных классах.

Вызывающая программа должна продемонстрировать все варианты вызова виртуальных функций с помощью указателей на базовый класс. Написать функцию вывода, получающую параметры базового класса по ссылке и демонстрирующую виртуальный вызов.

1. Создать абстрактный базовый класс **TFigure** с виртуальными методами вычисления площади и периметра. Создать производные классы: **TRectangle** (прямоугольник), **TCircle** (круг), **TTrapezium** (трапеция) со своими функциями площади и периметра. Самостоятельно определить, какие поля необходимы, какие из них можно задать в базовом классе, а какие — в производных. Площадь трапеции:

$$S=(a+b)*h/2.$$

2. Создать абстрактный базовый класс **TNumber** с виртуальными методами — арифметическими операциями. Создать производные классы **TInteger** (целое) и **TReal** (действительное).

3. Создать абстрактный базовый класс **TBody** (тело) с виртуальными функциями вычисления площади поверхности и объема. Создать производные классы: **TParallelepiped** (параллелепипед) и **TBall** (шар) со своими функциями площади поверхности и объема.

4. Создать абстрактный класс **TCurrency** (валюта) для работы с денежными суммами. Определить виртуальные функции перевода в рубли и вывода

на экран. Реализовать производные классы **TRouble** (рубль), **TDollar** (доллар) и **TEuro** (евро) со своими функциями перевода и вывода на экран.

5. Создать абстрактный базовый класс **TTriangle** для представления треугольника с виртуальными функциями вычисления площади и периметра. Поля данных должны включать две стороны и угол между ними. Определить классы-наследники: прямоугольный треугольник, равнобедренный треугольник, равносторонний треугольник со своими функциями вычисления площади и периметра.

6. Создать абстрактный базовый класс **TPair** (пара) с виртуальными арифметическими операциями. Создать производные классы **TFraction** (дробь) и **TComplex** (комплексное число). Дробное число должно быть представлено двумя полями: целая часть — длинное целое со знаком, дробная часть — беззнаковое короткое целое.

7. Создать абстрактный базовый класс **TRoot** (корень) с виртуальными методами вычисления корней и вывода результата на экран. Определить производные классы **TLinear** (линейное уравнение) и **TSquare** (квадратное уравнение) с собственными методами вычисления корней и вывода на экран.

8. Создать абстрактный базовый класс **TFunction** (функция) с виртуальными методами вычисления значения функции y=f(x) в заданной точке x и вывода результата на экран. Определить производные классы **TEllipse** (эллипс), **THyperbola** (гипербола) с собственными функциями вычисления y в зависимости от входного параметра x.

Уравнение эллипса:

$$x^2/a^2+y^2/b^2=1,$$

гиперболы:

$$x^2/a^2-y^2/b^2=1.$$

9. Создать абстрактный базовый класс **TPair** (пара) с виртуальными арифметическими операциями. Реализовать производные классы **TComplex** (комплексное число) и **TRational** (рациональное число). Рациональная (несократимая) дробь представляется парой чисел: числитель и знаменатель.

10. Создать абстрактный базовый класс **TTriad** (тройка) с виртуальными методами увеличения каждой составляющей на единицу. Создать производные классы **TDate** (год, месяц, день) и **TTime** (час, минута, секунда).

11. Создать абстрактный базовый класс **TPair** (пара) с виртуальными арифметическими операциями. Создать производные классы **TMoney** (деньги) и **TFraction** (дробное число). Деньги должны быть представлены двумя полями: беззнаковое длинное целое — для рублей, и беззнаковое короткое целое — для копеек. Дробное число должно быть представлено двумя полями: целая часть — длинное целое со знаком, дробная часть — беззнаковое короткое целое.

12. Создать абстрактный базовый класс **TPair** (пара) с виртуальными арифметическими операциями. Реализовать производные классы **TMoney** (деньги) и **TBills** (купюры). Деньги должны быть представлены двумя полями: беззнаковое длинное целое — для рублей, и беззнаковое короткое целое — для копеек. Купюры должны быть представлены количеством номиналов (1, 2, 5, 10, 50, 100, 500, 1000, 5000) для рублей и количеством номиналов (0.01, 0.05, 0.1, 0.5) для копеек.

13. Создать абстрактный базовый класс **TInteger** (целое) с виртуальными арифметическими операциями и функцией вывода на экран. Определить производные классы **TDecimal** (десятичное) и **TBinary** (двоичное), реализующие собственные арифметические операции и функцию вывода на экран. Число представляется массивом, каждый элемент которого — цифра.

14. Создать абстрактный базовый класс **TPair** (пара) с виртуальными арифметическими операциями. Создать производные классы **TMoney** (деньги) и **TComplex** (комплексное число). Деньги должны быть представлены двумя полями: беззнаковое длинное целое — для рублей и беззнаковое короткое целое — для копеек. Комплексное число должно быть представлено парой вещественных чисел.

15. Создать абстрактный базовый класс **TSeries** (прогрессия) с виртуальными функциями вычисления i-го элемента прогрессии и суммы прогрессии. Определить производные классы: **TLinear** (арифметическая прогрессия) и **TExponential** (геометрическая прогрессия). (Арифметическая прогрессия $a_i=a_0+id$, $i=0, 1, 2, ...$ Сумма арифметической прогрессии: $S_n=(n+1)(a_0+a_n)/2$. Геометрическая прогрессия: $a_i=a_o r^i$, $i=0, 1, 2, ...$ Сумма геометрической прогрессии:

$$S_n=(a_o - a_n r)/(1 - r).$$

16. Создать абстрактный класс **TNorm** с виртуальной функцией вычисления нормы и модуля. Определить производные классы **TComplex**, **TVector3D** с собственными функциями вычисления нормы и модуля. Модуль для комплексного числа вычисляется как корень из суммы квадратов действительной и мнимой частей; норма для комплексных чисел вычисляется как модуль в квадрате. Модуль вектора вычисляется как корень квадратный из суммы квадратов координат; норма вектора вычисляется как максимальное из абсолютных значений координат.

17. Создать абстрактный базовый класс **TPair** (пара) с виртуальными арифметическими операциями. Создать производные классы **TFuzzyNumber** (нечеткое число) и **TFraction** (задание 6). Нечеткое число представляется тройкой чисел $(x - e_1, x, x+e_2)$. Для чисел $A=(A - a_1, A, A+a_2)$ *и* $B=(B - b_1, B, B+b_2)$ арифметические операции выполняются по следующим формулам:

— сложение

$$A+B=(A+B - a_1 - b_1, A+B, A+B+a_2+b_2);$$

— вычитание

$$A - B=(A - B - a_1 - b_1, A - B, A - B+a_2+b_2);$$

— умножение

$$A\times B=(A\times B - B\times a_1 - A\times b_1+a_1\times b_1, A\times B, A\times B+B\times a_2+A\times b_2+a_2\times b_2);$$

— обратное число

$$A=(1/(A+a_2), 1/A, 1/(A - a_1)), A>0;$$

— деление

$$A/B=((A - a_1)/(B+b_2), A/B, (A+a_2)/(B+b_1), B>0.$$

18. Создать абстрактный базовый класс **TContainer** с виртуальными методами **sort()** и поэлементной обработки контейнера **foreach()**. Разработать производные классы **TBubble** (пузырек) и **TChoice** (выбор). В первом классе сортировка реализуется методом пузырька, а поэлементная обработка состоит

в извлечении квадратного корня. Во втором классе сортировка реализуется методом выбора, а поэлементная обработка — вычисление логарифма.

19. Создать абстрактный базовый класс **TArray** с виртуальными методами сложения и поэлементной обработки массива **foreach()**. Разработать производные классы **TSortArray** и **TXorArray**. В первом классе операция сложения реализуется как объединение множеств, а поэлементная обработка — сортировка. Во втором классе операция сложения реализуется как исключающее ИЛИ, а поэлементная обработка — вычисление корня.

20. Создать абстрактный базовый класс **TArray** с виртуальными методами сложения и поэлементной обработки массива **foreach()**. Разработать производные классы **TAndArray** и **TOrArray**. В первом классе операция сложения реализуется как пересечение множеств, а поэлементная обработка представляет собой извлечение квадратного корня. Во втором классе операция сложения реализуется как объединение, а поэлементная обработка — вычисление логарифма.

21. Следует ли рассматривать значение **nil** в языке Pascal или аналогичную величину **NULL** в языке C как полиморфный объект? Обоснуйте свой ответ.

В следующих задачах необходимо разработать шаблоны классов.

1. Создать шаблон класса «Односвязный список» и применить его для хранения списка товаров магазина и списка заявок от клиентов на покупку этих товаров.

2. Создать шаблон класса «Очередь» и применить его для хранения списка заявок на товары от клиентов (для каждого товара своя очередь) при решении предыдущей задачи.

3. Создать шаблон класса «Ассоциативный массив», индексом которого может быть элемент любого типа данных, включая объект класса, значение также может быть любого типа данных. Применить этот шаблон для хранения телефонной книжки (индекс — имя абонента, значение — массив телефонов). Применить этот шаблон для хранения каталога товаров (индекс — объект класса «Товар», значение — цена товара).

ЛАНЬ®

10. ОБРАБОТКА СОБЫТИЙ И СООБЩЕНИЙ

Если при объяснении вашей программы слушатель начинает кивать головой, его пора будить.

Алан. Дж. Перлис

10.1. Метафора обработки событий и сообщений

Многие черты Windows обязаны своей спецификой тому факту, что операционная система управляется событиями. В общем виде концепция событий рассмотрена в параграфе 5.2. Событие — это причина перехода объекта из одного состояния в другое. Взаимодействие объектов также осуществляется при помощи событий: изменение состояния одного объекта приводит к изменению состояния другого объекта, а сообщение оказывается средством связи между объектами. Запущенная программа обычно неактивна — она не выполняет никаких действий и пассивно ожидает сообщений о происходящих событиях от системы Windows. Таким образом, реакция и поведение программы определяется внешними и внутренними событиями, под воздействием которых в вычислительной системе начинается процесс передачи сообщений. В итоге получивший сообщение конечный адресат выполняет реакцию на произошедшее событие. Если реакция на событие не предусмотрена, оно обрабатывается стандартным образом операционной системой. На рисунке 10.1 представлена схема взаимодействия приложений, из которой видно, что приложение имеет возможность отправлять сообщения другим приложениям. Каждое приложение для временного хранения полученных сообщений имеет специальную очередь, содержимое которой обрабатывается в приоритетном порядке.

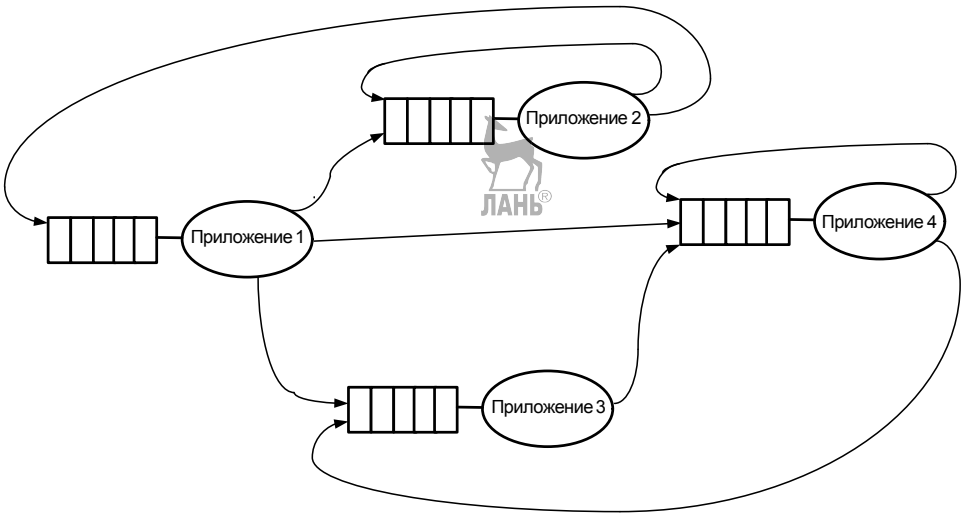


Рис. 10.1
Взаимодействие приложений

Система Windows предоставляет набор средств, с которыми приходится взаимодействовать как при создании, так и при использовании программы. Рассмотрим важнейшие для обработки событий и сообщений средства.

Самым важным средством является окно. Окно первым появляется на экране при запуске операционной системы или программы-приложения и последним исчезает при завершении работы. Windows сообщает окну приложения о некотором произошедшем в окружающей приложение среде событии при помощи сообщения. На рисунке 10.2 представлена роль окна при обработке событий. Когда мы перемещаем указатель мыши по окну, то Windows об этом узнает и отправит сообщение **WM_MOUSEMOVE** в очередь приложения, являющегося хозяином активного окна, по территории которого пробегает мышь. Приложение, не имеющее окна, сообщение получить не может. Кроме того, механизм сообщений является единственным для организации общения приложения со своими окнами.

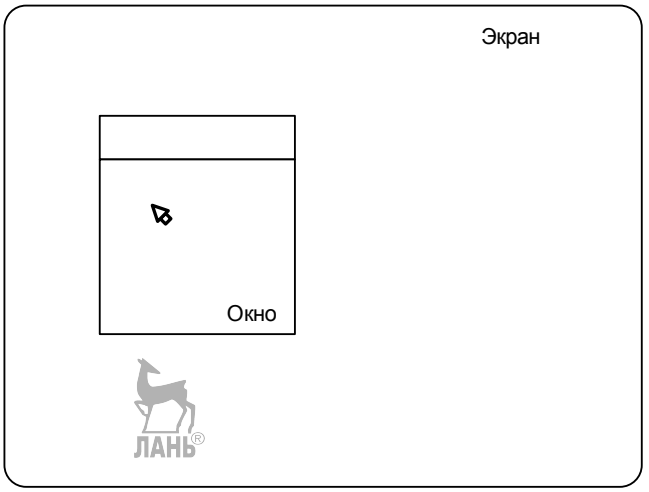


Рис. 10.2
Роль окна

Нашей задачей является изучение правил и освоение приемов обработки цикла:

событие → сообщение → обработка.

В этом цикле Windows обнаруживает в вычислительной системе некоторое *событие*, идентифицирует «заинтересованные» в событии приложения и отправляет каждому из них *сообщение* о произошедшем событии. Наконец, если приложение обладает *обработчиком события* и имеет возможность отреагировать на сообщение, то выполняется *обработка сообщения*. Таким образом, реакция приложения на происходящие в вычислительной системе события осуществляется с помощью механизма сообщений. Как правильно говорить: «обработка события» или «обработка сообщения»? В литературе можно встретить оба словосочетания, выражающих один и тот же смысл. Важно при этом понимать, что речь идет о реакции приложения на события с использованием механизма сообщений.

Для иллюстрации всего сказанного рассмотрим пример. Пусть в нашем распоряжении имеется экранная форма, на которой расположен компонент «панель». На панели размещена кнопка (рис. 10.3).

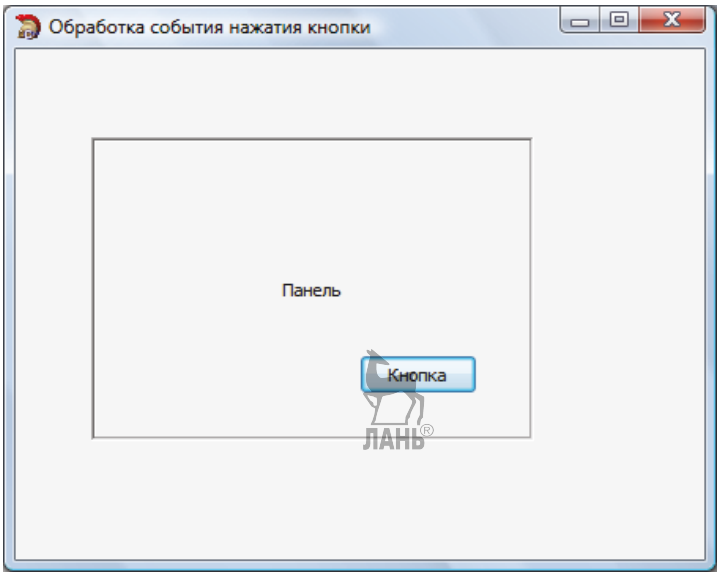


Рис. 10.3
Внешний вид экранной формы примера обработки события

Диаграмма классов приведена на рисунке 10.4.

Рис. 10.4
Диаграмма классов примера обработки события

Пусть над кнопкой расположился курсор. Рассмотрим по шагам процесс обработки события нажатия кнопки мыши. На рисунке 10.5 номера шагов приведены на стрелках.

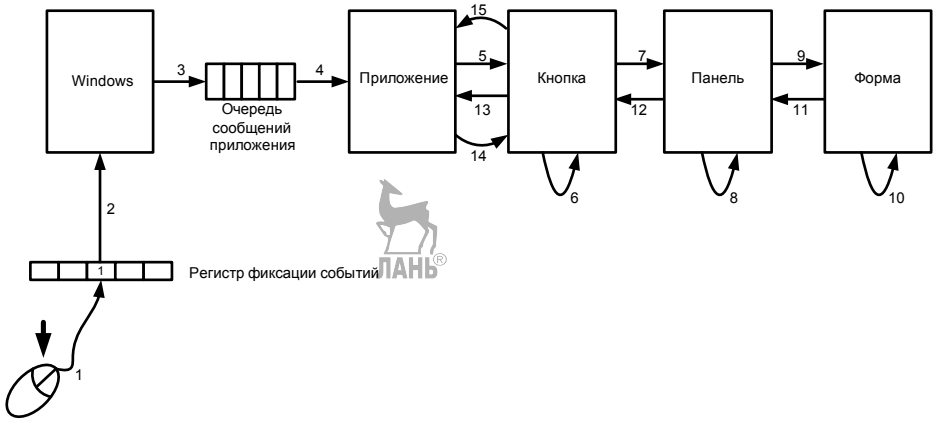


Рис. 10.5
Процесс обработки нажатия кнопки мыши

Шаг 1. Нажали на левую кнопку мыши, в регистр фиксации аппаратных событий поступил сигнал. Соответствующий левой кнопке мыши разряд регистра установлен в единицу.

Шаг 2. Windows периодически осуществляет опрос регистра фиксации событий. Во время очередного опроса, обнаружив единицу, он узнает, что произошло событие нажатия левой кнопки мыши. Сигнал принят, в соответствующий разряд регистра записан нолик. Состояние регистра восстановлено, и он готов к принятию очередного события: нажатия кнопки мыши.

Шаг 3. Операционная система установила, что во время нажатия кнопки мыши курсор находился в активном окне нашего приложения. Напомним, что если у приложения нет окна, то рассматриваемый процесс невозможен. В очередь приложения помещается параметризованное сообщение **WM_LBUTTONDOWN**, соответствующее нажатию левой клавиши мыши. Параметры сообщения содержат уточняющую информацию о сопутствующей событию обстановке. В нашем случае указанная информация включает позицию курсора на экране в момент нажатия на клавишу мыши и др.

Шаг 4. Выборка сообщения из очереди сообщений осуществляется с помощью метода:

```
void  TApplication::Run()
{
    do
        HandleMessages();
    while Terminated;
}.
```

Метод **Run** вызывается после инициализации приложения и создания экранных форм (см. в системе программирования файл, содержащий главную

функцию **_tWinMain**), поэтому цикл в теле **Run** исполняется до тех пор, пока переменная **Terminated** не получит значение «истина». Внутри цикла содержится единственный вызов метода **HandleMessages**:

```
void TApplication:: HandleMessages(void)
{
    if (! ProcessMethod())  Idle();
}.
```

Во время очередного вызова **HandleMessages** управление передается методу **ProcessMethod**, который, просмотрев очередь, обнаруживает, что в очереди имеется сообщение **WM_LBUTTONDOWN**. Заметим, что в случае отсутствия в очереди приложения сообщений управление передается обработчику **Idle**. Этот метод выполняется в так называемом фоновом режиме (при отсутствии сообщений) и может быть запрограммирован как обработчик события **OnIdle** компонента «экранная форма».

Шаг 5. Обнаружив сообщение **ProcessMethod**, определяет, что курсор находится над кнопкой и посылает аналог сообщения **WM_LBUTTONDOWN** кнопке. Как правило, это сообщение обозначается как **CN_XXX** (от Component Notify — уведомление) или **CM_XXX** (от Component Message). Название сообщений операционной системы WM_XXX произошло от Windows Message.

Шаг 6. Кнопка получает сообщение и посылает себе сообщение. Ее цель — узнать, можно ли обработать сообщение **WM_LBUTTONDOWN** (кнопка может быть заблокирована или у нее не предусмотрена обработка сообщения **WM_LBUTTONDOWN**). Если кнопка может обработать сообщение, то выполнить шаг 7, иначе — перейти к шагу 13.

Шаг 7. Кнопка посылает сообщение вида **CM_XXX** своему владельцу — панели (см. в главе 12 устройство класса **TComponent**, потомком которого является класс **TPanel**). Теперь ее цель — узнать, нет ли препятствий у владельца кнопки для обработки сообщения (если владелец заблокирован, то заблокированы и все его компоненты).

Шаг 8. Выполняется для панели аналогично шагу 6 для кнопки. Если панель может обработать сообщение, то выполнить шаг 9, иначе — перейти к шагу 12.

Шаг 9. Выполняется для панели аналогично шагу 7 для кнопки.

Шаг 10. Форма является самым авторитетным компонентом, который принимает решение о том, что нет препятствий для обработки сообщения **WM_LBUTTONDOWN**. В случае, если форма находится в фокусе, то при наличии обработчика нажатия левой клавиши мыши сообщение может быть обработано.

Шаги 11, 12, 13. Содержат посылку ответа на запрос о возможности обработать сообщение. Ответы могут быть положительными или отрицательными. Если на кнопке сразу обнаружен отрицательный ответ, то остальные компоненты не участвуют.

Шаг 14. Метод **ProcessMethod**, получив отрицательный ответ, вызывает метод обработки сообщений по умолчанию **DefaultHandler**, содержащийся в классе **TObject**. Если получен положительный ответ, то кнопке отправляется сообщение о возможности обработать поступившее сообщение **WM_LBUTTONDOWN**.

Шаг 15. Кнопка начинает обработку сообщения **WM_LBUTTONDOWN**. Возможно, что при этом будут отправлены сообщения другим компонентам, все зависит от того, как реализован метод **OnClickButton**, отвечающий за обработку события нажатия нашей кнопки. В конце приложению поступает ответ, что сообщение обработано.

Приложение извлекает сообщение **WM_LBUTTONDOWN** из очереди сообщений. Цикл обработки события нажатия на левую клавишу мыши завершен.

Контрольные вопросы

1. Опишите понятие «событие».
2. Опишите понятие «сообщение».
3. Опишите понятия «обработчик события», «обработчик сообщения».
4. Почему операционную систему назвали Windows (окна)?
5. Подсчитайте количество отправленных сообщений в рассмотренном примере обработки события нажатия левой клавиши мыши.

10.2. Обработка сообщения в программе

В абсолютном большинстве случаев обработчики стандартных событий создают и настраивают с помощью Проектировщика среды C++Builder. Программист формирует программный обработчик нужного события в окне Проектировщика среды C++Builder. Любой элемент управления, расположенный на форме, описывается не только набором свойств, доступных в Инспекторе объектов, но и набором событий, которые нужно программно обрабатывать. Эти события доступны на закладке Events в Инспекторе объектов. Имена событий имеют вид: OnClick — событие нажатия клавиши мыши над выбранным компонентом, OnExit — событие выхода курсора за пределы компонента, OnMouseDown — событие нажатия на левую клавишу мыши и т. п.

По умолчанию в программе не происходит специальной обработки получаемых сообщений. Программист должен явно сформировать собственный обработчик интересующего события.

Это можно сделать двумя способами:

— дважды щелкнуть мышью на строке Инспектора объектов с именем нужного события;

— дважды щелкнуть на элементе управления на форме (если для этого элемента поддерживается возможность создания подходящего обработчика).

После любого из перечисленных действий открывается окно редактора кода, в котором появится автоматически сгенерированный шаблон обработчика. В тело обработчика нужно вписать действия, которые мы хотим связать с наступлением выбранного события. Обработчик события будет автоматически вызван при наступлении соответствующего события.

Например, чтобы создать обработчик нажатия кнопки, достаточно дважды щелкнуть на образе этой кнопки на форме в окне Проектировщика. Среда C++Builder автоматически сгенерирует программный код обработчика и переключится в режим редактирования исходных текстов:

```
void __fastcall TForm1::Button1Click (TObject * Sender)
{
}.
```

Заметим, что обработчик события всегда принадлежит классу формы, а не классу конкретного элемента управления. Если создается обработчик нажатия кнопки или обработчик выбора пункта меню, он всегда будет принадлежать не классу кнопки или меню, а классу родительской формы элемента управления.

В параметр **Sender** обработчик получает адрес вызвавшего обрабатываемое событие объекта. Если создается обработчик нажатия кнопки, то параметр **Sender** содержит ссылку на эту кнопку. Чтобы задействовать этот объект в программном коде, его надо преобразовать к типу **TButton** с помощью приведения типов вида, например

```
(dynamic_cast<TButton*>(Sender))–>Caption=«Кнопка нажата»;
```

Рассмотрим пример обработки двойного нажатия левой клавиши мыши. В инспекторе объектов выделяем форму, выбираем закладку Events, находим сообщение **OnDblClick** и в правой части строки два раза щелкаем мышью. В редакторе кода вводим действия по обработке сообщения. В итоге описание класса **TForm2** автоматически будет расширено описанием функции **FormDblClick**:

```
class TForm2 : public TForm
{
    __published:
            void __fastcall FormDblClick(TObject *Sender);
    public:
            __fastcall TForm2(TComponent* Owner);
};.
```

Обрабатывающая событие функция может иметь вид:

```
void __fastcall TForm2::FormDblClick(TObject *Sender)
{
    ShowMessage(«Двойное нажатие левой клавиши мыши»);
}
```

Теперь, после запуска приложения и двойного щелчка левой клавишей мыши, на экране будет появляться экранная форма (результат вызова функции **ShowMessage**) с соответствующим текстом.

Контрольные вопросы

1. Какими предусмотренными системой программирования способами можно включить в программу обработчик события?

10.3. Отправка сообщения из программы

Для отправки сообщений из программы используются соответствующие средства Windows API. Частично они были рассмотрены в параграфе 5.6.

Заголовок первой функции имеет вид:

LRESULT SendMessage (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam).

Содержание функции **SendMessage** заключается в следующем: отправить окну **hWnd** сообщение **Msg** и дождаться обработки; **wParam, lParam** содержат дополнительную информацию о сообщении. Если параметр окна **hWnd** имеет значение **HWND_BROADCAST**, то сообщение передается всем окнам верхнего уровня в системе, включая недоступные и невидимые, но кроме дочерних окон. Возвращаемый функцией результат имеет тип адрес.

Вторая функция с заголовком вида:

bool PostMessage (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)

отличается от функции **SendMessage** лишь тем, что помещает сообщение в очередь и возвращается. Если **hWnd=NULL**, то сообщение ставится в очередь текущего приложения. Возвращает «истину» в случае успеха и «ложь» в случае неуспеха.

Имеется еще один метод, который может посылать сообщение непосредственно оконному компоненту. Это метод **Perform**, заголовок которого имеет вид:

int Perform(unsigned Msg, int WParam, int LParam).

Приведем теперь примеры использования функций отправления сообщений.

Закрытие внутренней формы **Form2** приложения может быть выполнено вызовом

SendMessage (Form2–>Handle, WM_CLOSE, 0, 0);.

Этот оператор отправляет сообщение **WM_CLOSE** (второй параметр) форме **Form2**. Первый параметр этой функции содержит дескриптор окна формы **Form2**, задаваемый свойством **Handle**. Сообщение **WM_CLOSE** не нуждается в параметрах, поэтому третий и четвертый параметры вызова функции **SendMessage** равны нулю.

Тот же результат можно получить, отправляя сообщение **WM_CLOSE** непосредственно форме **Form2**:

Form2–>Perform (WM_CLOSE, 0, 0);.

Во втором примере закрывается форма **TForm3** приложения **Application2** путем вызова функции:

SendMessage (FindWindow («TForm3», «Application2»), WM_CLOSE, 0, 0);

Этот оператор использует функцию **FindWindow** для определения дескриптора, интересующего нас окна приложения.

Аналогичный пример закрывает окно калькулятора приложения **SciCalc**:

SendMessage (FindWindow («SciCalc», «Калькулятор»), WM_CLOSE, 0, 0).

Теперь рассмотрим пример посылки сообщения компоненту. Пусть у нас имеется экранная форма **Form2**, на которой расположены две кнопки с имена-

ми **Button1** и **Button2**. Мы хотим с помощью кнопки **Button1** смоделировать управление кнопкой **Button2**.

Событию **OnClick** нажатия кнопки **Button1** сопоставим обработчик:

```
void __fastcall TForm2::Button1Click(TObject *Sender)
{
    int x=SendMessage (Button2–>Handle, WM_LBUTTONDOWN, 0, 0);
    int x1=SendMessage (Button2–>Handle, WM_LBUTTONUP, 0, 0);
}.
```

Событию **OnEnter** кнопки **Button2** сопоставим обработчик:

```
void __fastcall TForm2::Button2Enter(TObject *Sender)
{
  ShowMessage(«Кнопка 2 в фокусе»);
}
```

Наконец, событию **OnClick** кнопки **Button2** сопоставим обработчик:

```
void __fastcall TForm2::Button2Click(TObject *Sender)
{
  ShowMessage(«Кнопка 2 обработана»);
}
```

Запускаем программу и нажимаем кнопку **Button1**. Вызывается метод **Button1Click**. После посылки сообщения

SendMessage (Button2–>Handle, WM_LBUTTONDOWN, 0, 0)

вызывается метод **Button2Enter** и на экране появляется окно с текстом «Кнопка 2 в фокусе», а после обработки отправленного сообщения

SendMessage (Button2–>Handle, WM_LBUTTONUP, 0, 0)

вызывается обработчик **Button2Click** и выводится сообщение «Кнопка 2 обработана».

Контрольные вопросы

1. Какие функции предназначены для отправки сообщений?

10.4. Программирование сообщений и событий

10.4.1. Структура сообщения

Сначала выясним, что представляет собой сообщение и его параметры.

Во-первых, каждое сообщение идентифицируется константой целого типа. Название константы-сообщения начинается с префикса **WM_**, далее следует смысловое обозначение константы. Например, **WM_MOUSEMOVE** — сообщение о перемещении мыши, **WM_CREATE** — сообщение о создании формы. В модуле winuser.h стандартной поставки системы программирования RAD Studio имеется большое количество описаний таких констант. Например,

#define WM_CLOSE 0x0010
#define WM_MOUSEMOVE 0x0200.

Для компонентов RAD Studio константы имеют вид **CM_XXX** (Component Message — сообщение компонента) или **CN_XXX** (Component Notify — уведомление компонента), или **DM_XXX** (Drag Message — сообщение о перетаскивании компонента).

Часто сообщения образуют тематические группы. Например, **WM_KEYDOWN**, **CN_KEYDOWN**, **CN_SYSKEYDOWN**, **CM_CHILDKEY**.

Во-вторых, для каждого вида сообщения имеется структура, с помощью которой сообщение параметризуется. Заглянув в модуль Messages.hpp стандартной поставки RAD studio, можно найти описание структуры параметра сообщения следующего вида (структура приведена в сокращении):

```
struct TMessage
{
    public:
        unsigned Msg;
        union
        {
            struct
               {
                   System::Word WParamLo;
                   System::Word WParamHi;
                   System::Word LParamLo;
                   System::Word LParamHi;
                   System::Word ResultLo;
                   System::Word ResultHi;
               };
            struct
               {
                    NativeUInt WParam;
                   NativeInt LParam;
                   NativeInt Result;
               };
        };
};
```

где **Msg —** идентификатор сообщения, **WParamHi, WParamLo** — комбинация двух полей, содержащая, соответственно, старшие и младшие разряды первого параметра сообщения; **LParamHi, LParamLo** — аналогично, комбинация двух полей, содержащая второй параметр сообщения; **ResultHi, ResultLo** — комбинация двух полей, хранящих результат обработки сообщения.

В нижней части объединения приведена структура, в которой можно хранить аналогичные данные, но с использованием меньшего числа разрядов. Названия первый и второй параметры получили от word parameter (**WParam**) и long parameter (**LParam**). Типы данных **NativeUInt** и **NativeInt** являются синонимами типов, соответственно, **unsigned int** и **int**. Как видим из описания структуры **TMessage**, в процессе развития операционной системы и системы программирования структура параметров изменяется. Совершенствование Windows продолжается, поэтому возможны и дальнейшие изменения. Уточнить со-

держание структуры **TMessage** можно в справочниках по системе программирования или заглянув в модуль Messages.hpp.

Первые два компонента параметра сообщения аналогичны параметрам функции. После получения сообщения Windows из параметра можно извлечь информацию, относящуюся к отправленному сообщению. Например, **WM_LBUTTONDOWN** — это уведомляющее сообщение, которое посылается после щелчка левой клавиши мыши в пределах окна приложения. **WParam** для сообщения **WM_LBUTTONDOWN** хранит информацию о нажатых клавишах в момент возникновения данного события, **LParam** содержит координаты курсора в момент щечка мыши. Координата **X** содержится в младшем слове **WParamLo**, а координата **Y** — в старшем слове **WParamHi**.

Во многих случаях программист не должен непосредственно обращаться к структуре **TMessage**, так как библиотека визуальных компонентов, входящая в состав системы программирования, берет эту работу на себя. Например, если мы создаем стандартными средствами обработчик сообщения для события **OnMouseDown**, автоматически сгенерированная C++Builder функция будет выглядеть следующим образом:

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton
Button,
                                      TShiftState Shift,  int X, int Y)
{

},
```

где параметр **Sender**, как обычно, является ссылкой на вызвавший данную функцию объект. Остальные параметры связаны с рассматриваемой нами темой. Параметр **Button** может принимать значения, определенные в описании его типа: **enum TMouseButton { mbLeft, mbRight, mbMiddle }** (нажата, соответственно, левая или правая или средняя клавиша мыши). Параметр **Shift** содержит множество, определенное типом

```
typedef System::Set<Classes__1, Classes__1::ssShift, Classes__1::ssPen>
TShiftState,
```

где тип **Classes__1** определен как

```
enum Classes__1 { ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble,
ssTouch, ssPen }.
```

Элементы множества **TShiftState** соответствуют нажатию клавиш Shift (**ssShift**), Alt (**ssAlt**), Ctrl (**ssCtrl**) клавиатуры, а также кнопок мыши: левой (**ssLeft**), правой (**ssRight**), средней (**ssMiddle**). Информация о нажатых кнопках в параметрах **Button** и **Shift** имеет различный смысл. Параметр **Button** соответствует кнопке, нажимаемой в последний момент, а параметр **Shift** содержит информацию о том, какие кнопки были нажаты, в том числе и ранее. Например, если пользователь нажмет левую кнопку мыши, а затем, не отпуская ее, нажмет правую, то после первого нажатия множество **Shift** будет равно **[ssLeft]**, а после второго — **[ssLeft, ssRight]**. Если до этого пользователь нажал и не отпустил

какие-то вспомогательные клавиши, то информация о них будет также присутствовать в множестве **Shift**. Значение **ssDouble** говорит о двойном щелчке на соответствующей клавише мыши.

Поскольку **Shift** является множеством, проверять наличие в нем тех или иных элементов надо методом **Contains**, который возвращает логическое значение. Например, если мы хотим прореагировать на событие, заключающееся в нажатии левой кнопки мыши при нажатой клавише Alt, можно использовать оператор:

```
if ((Button == mbLeft) && (Shift.Contains (ssAlt)))   ….
```

Во все сообщения, связанные с мышью, передаются также координаты курсора **X** и **Y**. Эти параметры определяют координаты курсора в клиентской области компонента. Благодаря этому можно обеспечить различную реакцию в зависимости от того, в какой части клиентской области формы расположен курсор.

Рассмотрим пример. Пусть мы хотим обеспечить перетаскивание некоторого компонента на экране при условии, что перетаскивание начинается нажатием левой клавиши мыши при предварительно нажатой клавише Alt. В этом случае на компоненте нужно свойство **DragMode** установить равным **dmManual**. В этом случае начало процесса перетаскивания определяет программист с помощью события **OnMouseDown**. Обработчик сообщения **OnMouseDown** для компонента **TListBox** может иметь вид:

```
void __fastcall TForm1::ListBox1MouseDown(TObject * Sender, TMouseButton Button,
                                          TShiftState Shift, int X, int Y)
{
    if ((Button == mbLeft) && (Shift.Contains(ssAlt))
        ((TControl * ) Sender)–>BeginDrag(False, 5);
}
```

В рассмотренном примере метод **BeginDrag** необходимо вызывать в случае ручного управления перетаскиванием (**DragMode** равно значению **dmManual**). Первый параметр функции **BeginDrag** значением **False** устанавливает, что внешний вид курсора при перетаскивании компонента изменяется не сразу, а после фактического смещения мыши на некоторое число пикселов, устанавливаемое вторым параметром — 5. Это позволяет легко отличить простой щелчок мыши от начала перетаскивания.

Подведем итоги нашего обсуждения. Для каждого вида сообщений в системе программирования предусмотрена аналогичная **TMessage** структура и правила использования параметров сообщения. Например, при обработке сообщений от мыши параметр сообщения может содержать позицию на экране указателя мыши. При обработке событий от устройств параметр может содержать контекст устройства.

Мы не будем подробно рассматривать структуры параметров различных сообщений, их можно взять из справочника, из помощи системы программирования и, наконец, из Интернета. Приведем только некоторые названия структур параметров сообщений: **TWMNoParams**, **TWMMouse**, **TWMKey**, **TWMScroll**.

Обычно при использовании стандартной библиотеки компонентов содержание параметра сообщения находится в параметрах, обрабатывающих событие функций. В случае полностью самостоятельного использования параметра сообщения содержание и использование полей структуры определяются соображениями программиста.

Рассмотрим пример, в котором два и более одинаковых приложения могут обмениваться сообщениями друг с другом. Каждый экземпляр приложения содержит на своей форме две метки класса **TLabel** и кнопку класса **TButton**. При нажатии кнопки всем приложениям отправляется сообщение, параметры которого содержат дескриптор и значение ширины отправляющего сообщение окна. Каждое получившее сообщение приложение отображает параметры сообщения в соответствующих метках.

Мы будем использовать значение дескриптора **HWND_BROADCAST**, который имеет специальное назначение. С ним сообщение помещается в очередях всех окон верхнего уровня в системе, включая заблокированные или невидимые, не имеющие владельца, окна, перекрывающие и выскакивающие окна (сообщение не помещается в очередях дочерних окон).

Прикладные программы, которым необходимо общаться друг с другом при помощи дескриптора **HWND_BROADCAST**, должны использовать функцию **RegisterWindowMessage**, чтобы получить уникальное сообщение для связи между приложениями.

Определяем тип структуры нашего сообщения

```
struct TMyMessage
 {
        Cardinal  Msg;
        HWND Handle;
        Integer Info;
        Integer Result;
  },
```

где **Msg** содержит идентификатор полученного сообщения, **Handle** соответствует полю **WParam** из **TMessage** и содержит дескриптор формы отправившего сообщение приложения, **Info** соответствует полю **LParam** из **TMessage** и содержит передаваемую с сообщением информацию о ширине отправляющего сообщение окна, **Result** соответствует полю **Result** из **TMessage**, поле введено для совместимости со структурой **TMessage**, в нашем примере использоваться не будет.

Описание формы имеет вид:

```
class TForm2 : public TForm
{
__published:
        TLabel *lblFHandle;
        TLabel *lblFProperty;
        TButton *btnFStartMessage;
        void __fastcall btnFStartMessageClick(TObject *Sender);
private:
        DWORD WM_MYMESSAGE;
```

```
public:
        __fastcall TForm2(TComponent* Owner);
        void __fastcall DefaultHandler(void *StandartMessage);  // переопределяем
},
```

где **WM_MYMESSAGE** — идентификатор нашего сообщения, **DefaultHandler** — переопределяемый в нашем приложении стандартный обработчик сообщений.

В конструктор формы поместим процедуру регистрации в операционной системе нашего сообщения:

```
__fastcall TForm2::TForm2(TComponent* Owner)
  : TForm(Owner)
{
  WM_MYMESSAGE=RegisterWindowMessage(L «Наше передаваемое сообщение»);
}
```

Переопределим стандартный обработчик сообщений следующим образом:

```
void __fastcall TForm2::DefaultHandler(void *Message)
{
        TMyMessage MyMessage;
        if ((static_cast<TMyMessage*>(Message))–>Msg == WM_MYMESSAGE)
        {
         MyMessage.Msg=(static_cast<TMyMessage*>(Message))–>Msg;
         MyMessage.Handle=static_cast<HWND>((static_cast<TMyMessage*>(Message))–
>Handle);
         MyMessage.Info=static_cast<TMyMessage*>(Message)–>Info;
         // проверяем, если сообщение не от нас
         if (MyMessage.Handle != Handle)
         {
                lblFHandle–>Caption=L»Дескриптор формы отправителя сообщения «
                                + IntToStr(reinterpret_cast<int>(MyMessage.Handle));
                lblFProperty–>Caption=L»Ширина формы отправителя сообщения «
                                  +IntToStr(static_cast<int>(MyMessage.Info));
                Width=MyMessage.Info;
         }
        }
        else
         TCustomForm::DefaultHandler(Message);
}
```

Поясним содержание нашего обработчика сообщений **DefaultHandler**.

Первой строкой объявляется структура сообщения **MyMessage**, которую нам предстоит заполнить, расшифровав содержание пришедшего сообщения **Message**. В условном операторе проверяем, является ли пришедшее сообщение нашим **WM_MYMESSAGE** сообщением. Параметр **Message** обработчика сообщений имеет тип **void ***, поэтому выполняется с помощью приведения **static_cast** преобразование параметра к типу **TMyMessage***.

Если пришедшее сообщение является сообщением **WM_MYMESSAGE**, то заполняем структуру **MyMessage**. Использование структуры **MyMessage**

в нашем примере имеет методический характер. Для наших целей можно было бы использовать непосредственно параметр **Message**.

Далее, если получившее сообщение приложение не является отправителем сообщения, заполняем поля экранной формы полученными параметрами сообщения. Обратим внимание на использование приведения **reinterpret_cast** для преобразования дескриптора **MyMessage.Handle** в целое число. Функция **IntToStr** требует в качестве параметра значение целого типа. Если мы в этой ситуации выберем приведение **static_cast**, то получим сообщение об ошибке. Даже интуитивно понятно, что преобразование дескриптора в целое число выглядит не очень убедительно. В п. 5.1.7 мы обсудили назначение дескриптора. С помощью дескриптора операционная система Windows идентифицирует используемые ресурсы. Дескриптор ресурса представляет собой ссылку на ресурс, и работа с ним осуществляется только применением стандартных функций. Пользователь не имеет прямого доступа к дескриптору. В нашем случае вывод значения дескриптора на экран носит чисто иллюстративный характер и не является естественным его использованием.

Для большей наглядности выполняется присваивание **Width=MyMessage.Info**. Предлагаем объяснить смысл этого присваивания самостоятельно.

Если пришедшее сообщения не является сообщением **WM_MYMESSAGE**, то вызываем стандартный обработчик сообщения, переслав ему полученное сообщение **Message**.

Наконец рассмотрим функцию обработки события кнопки:

```
void __fastcall TForm2::btnFStartMessageClick(TObject *Sender)
{
  NativeUInt x=reinterpret_cast<NativeUInt> (Handle);
  SendMessage(HWND_BROADCAST, WM_MYMESSAGE, x, Width);
}
```

Переменная **x** имеет тип **NativeUInt**, который объявлен в модуле **sysmac.h** как синоним типа данных **unsigned int**. В стандартном параметре сообщения **TMessage**, структура которого обсуждалась в начале этого параграфа, поля **WParam**, **LParam** и **Result** имеют тип **NativeUInt**. В нашем случае в параметре сообщения **TMyMessage** используется тип данных **HWND**, поэтому для согласования параметров с функцией **SendMessage** введено преобразование типа данных **HWND** к типу данных **NativeUInt**.

Наконец, отправляем сообщение **WM_MYMESSAGE** во все очереди окон верхнего уровня, передавая дескриптор собственного окна и значение ширины собственного окна.

Контрольные вопросы

1. Как выглядит идентификатор сообщения?
2. Как выглядит параметр сообщения?
3. Как использовать параметры сообщения в случае работы со стандартной библиотекой визуальных компонентов?

4. Какой дескриптор используется для передачи сообщения всем окнам верхнего уровня?

5. Каким образом вводится собственное сообщение?

6. Как переопределить стандартный обработчик сообщений для обработки собственного сообщения?

10.4.2. Включение обработки сообщений в программу

К включению сообщений в программу нужно прибегать в двух случаях.

1. В системе программирования не предусмотрена стандартная обработка сообщения, которое имеется в операционной системе. Например, в инспекторе объектов RAD studio отсутствует обработка сообщения, соответствующего событию двойного нажатия правой клавиши мыши **WM_RBUTTONDBLCLK**.

2. Мы хотим из нашего приложения посылать сообщения другим приложениям.

Вернемся к циклу обработки событий:

событие → сообщение → обработка.

Мы уже рассмотрели случай, когда программируется только обработка события: для выделенного на форме компонента в инспекторе объектов выбираем закладку Events — события и ищем имя события, подлежащего обработке. Для выбранного события программируем обработчик события.

Теперь речь идет о программировании звена ***сообщение → обработка***.

В этом случае нашей задачей будет сопоставление сообщению собственного обработчика события. Обработчик должен отвечать требованиям:

— должен быть методом класса;

— принимать по ссылке параметр типа TMessage или другого специализированного класса.

Перехват сообщения в программе осуществляется автоматически. Для сопоставления сообщению нужного обработчика в системе программирования RAD studio предусмотрена карта сообщения.

Карта сообщения представляет собой предложение, которое включается в описание класса, и имеет вид

```
BEGIN_MESSAGE_MAP
      MESSAGE_HANDLER (A, B, C)
            . . .
END_MESSAGE_MAP,
```

где **A** — имя сообщения, **B** — тип параметра сообщения, **C** — имя функции-обработчика сообщения. Основное назначение карты сообщений состоит в том, чтобы собрать перечисленные компоненты в единую систему.

Рассмотрим примеры описания сообщения.

Пример 1. Описание класса, содержащего карту сообщения, имеет вид

```
class TForm2 : public TForm
{
     private:
void __fastcall OnRBtnDblClk(TWMRButtonDblClk &Mes);
```

```
      public:
BEGIN_MESSAGE_MAP
 MES-
SAGE_HANDLER(WM_RBUTTONDBLCLK,TWMRButtonDblClk,OnRBtnDblClk)
END_MESSAGE_MAP(TComponent)
__fastcall TForm2(TComponent* Owner);
};
```

Пусть реализация обработчика сообщения имеет вид:

```
void __fastcall TForm2::OnRBtnDblClk(TWMRButtonDblClk &Mes)
{
      ShowMessage(«Получили сообщение WM_RBUTTONDBLCLK «);
}
```

После запуска программы наведем курсор на нашу форму, выполним двойной щелчок правой клавиши мыши. Над формой появится окно метода **ShowMessage** с надписью: «Получили сообщение WM_RBUTTONDBLCLK». Что же произошло? Windows обнаруживает событие двойного нажатия правой клавиши мыши и классифицирует его как двойное нажатие правой клавиши мыши. Событие произошло в тот момент, когда курсор находился на форме нашего приложения. Поэтому сообщение **WM_RBUTTONDBLCLK** отправляется нашему приложению.

Пример 2. Создадим обработчик сообщения, который будет выводить на экран коды нажатой клавиши клавиатуры:

```
class TForm2 : public TForm
{
  __published:
TLabel *Label1;
  private:
void __fastcall WMCharCode(TWMKey &Mes);
  public:
__fastcall TForm2(TComponent* Owner);
BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER (WM_CHAR,TWMKey , WMCharCode)
END_MESSAGE_MAP(TComponent);
}
void __fastcall TForm2::WMChar(TWMKey &Mes)
{
    Label1–>Caption=IntToStr(Mes.CharCode);
}
```

где структура **TWMKey** определена в системе программирования следующим образом (приведена в сокращении):

```
struct TWMKey
{
public:
      unsigned Msg;
      System::Word CharCode;
```

```
    NativeInt Result;
}
```

Во время обработки события нажатия на клавишу клавиатуры в поле **CharCode** параметра сообщения **TWMKey** автоматически записывается код нажатой клавиши.

Заметим, что решение этой задачи может быть получено стандартным способом использования события **OnKeyPress** в инспекторе объектов:

```
class TForm2 : public TForm
{
__published:
TLabel *Label1;
void __fastcall FormKeyPress(TObject *Sender, System::WideChar &Key);
public:
    __fastcall TForm2(TComponent* Owner);
}
void __fastcall TForm2::FormKeyPress(TObject *Sender, System::WideChar &Key)
{
  Label1–>Caption=IntToStr(Key);
}
```

Пример 3. Рассмотрим теперь пример отправки сообщения другому приложению.

В параграфе 10.3 рассмотрен пример отправки сообщения от компонента к компоненту: кнопка управляет поведением другой кнопки. Распространим теперь этот пример на различные приложения. В приложении **AppSender** на экранной форме **frmFrom** расположена кнопка, в приложении **AppRecipient** также имеется на форме **frmTo** кнопка. Мы хотим управлять кнопкой из приложения **AppRecipient** нажатием кнопки из приложения **AppSender**.

Обработчик события нажатия кнопки из формы **frmFrom** имеет вид:

```
 void __fastcall TfrmFrom::Button1Click(TObject *Sender)
 {
    SendMessage(FindWindow (L«TfrmTo», L«Форма_получатель»),
WM_LBUTTONDOWN, 0, 0);
    SendMessage(FindWindow (L«TfrmTo», L«Форма_получатель»),
WM_LBUTTONUP, 0, 0);
 }
```

При нажатии кнопки отправляется сообщение приложению **AppRecipient**. Для получения дескриптора формы используется функция **FindWindow** с параметрами, соответственно, **«TfrmTo»** — класс формы получателя, **«Форма_получатель»** — заголовок формы-получателя. Отправляется сообщение нажатия кнопки **WM_LBUTTONDOWN**. Аналогично в ответ на событие отпускания кнопки отправляется соответствующее сообщение **WM_LBUTTONUP**.

Рассмотрим теперь приложение-получатель сообщений **AppRecipient**. Класс экранной формы frmTo имеет вид:

```
class TfrmTo : public TForm
{
__published:
TButton *Button1;
void __fastcall Button1Click(TObject *Sender);
void __fastcall Button1Enter(TObject *Sender);
private:
        void __fastcall DownMessage(TWMLButtonDown &Mes);
        void __fastcall UpMessage(TWMLButtonUp &Mes);
public:                 // User declarations
__fastcall TfrmTo(TComponent* Owner);
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_LBUTTONDOWN, TWMLButtonDown, DownMessage)
  MESSAGE_HANDLER(WM_LBUTTONUP, TWMLButtonUp, UpMessage)
END_MESSAGE_MAP(TComponent)
}
```

В описание класса **TfrmTo** включены:

— карта сообщений, в которой содержатся отношения (сообщение, параметр сообщения, обработчик) для событий нажатия и отпускания клавиши мыши;

— обработчики сообщений нажатия и отпускания клавиши мыши;

— обработчики сообщений нажатия кнопки **Button1** и перевода ее в фокус.

Реализация обработчиков имеет вид:

```
void __fastcall TfrmTo::Button1Click(TObject *Sender)
{
  ShowMessage(«Кнопка обработана»);
}
void __fastcall TfrmTo::Button1Enter(TObject *Sender)
{
  ShowMessage(«Кнопка в фокусе»);
}
void __fastcall TfrmTo::DownMessage(TWMLButtonDown &Mes)
{
  ShowMessage(«Получено сообщение: Перевести кнопку в фокус»);
  Button1Enter(this);
}
void __fastcall TfrmTo::UpMessage(TWMLButtonUp &Mes)
{
  ShowMessage(«Получено сообщение: Обработать кнопку»);
  Button1Click(this);
}
```

Запускаем оба приложения. При щелчке мыши на кнопке приложения **AppSender** в приложении **AppRecipient** на экран с помощью функции **ShowMessage** выводятся сообщения в следующем порядке:

1. «Получено сообщение: Перевести кнопку в фокус».
2. «Кнопка в фокусе».

3. «Получено сообщение: Обработать кнопку».
4. «Кнопка обработана».

Контрольные вопросы

1. В каких случаях нужно прибегать к программированию сообщений?
2. Какое назначение карты сообщений?
3. Какой синтаксис карты сообщений?
4. Как обработать событие, отсутствующее в инспекторе объектов, но имеющееся в операционной системе?
5. Как отправить сообщение другому приложению и там его обработать?
6. В примере 2 заменим поле метки на поле редактирования. В обработчике сообщения присваивание заменим на **Edit1–>Text=IntToStr(Key)**. Запустив программу, увидим, что в поле редактирования выводятся не коды символов, а сами символы. Почему выводятся символы, а не их коды?
7. В примере 3 приложения, в каких **AppRecipient** обладает побочным эффектом при щелчке мышью на форме. В чем суть указанного эффекта?

10.4.3. Уведомляющие сообщения

Уведомляющие сообщения, или уведомления (notification), представляют собой сообщения, отсылаемые родительскому окну в том случае, когда в одном из его дочерних элементов управления происходит нечто, заслуживающее внимания родителя. Эти сообщения рассылаются только стандартными элементами управления Windows (кнопками, списками, раскрывающимися списками, полями редактирования) и общими элементами управления Windows (деревом объектов, списком объектов и т. п.). Щелчок или двойной щелчок на элементе управления, выбор текста в поле редактирования или перемещение бегунка полосы прокрутки — вот примеры событий, отсылающие уведомляющие сообщения. Обычно эти сообщения являются результатом действий пользователя, и именно родительское окно — идеальное место для реагирования на эти действия.

Уведомления обрабатываются с помощью соответствующих процедур обработки, принадлежащих той форме, в которой содержится данный элемент управления. В таблице 10.1 перечислены часто используемые уведомляющие сообщения для стандартных элементов управления.

Таблица 10.1

Уведомления для стандартных элементов управления

Уведомление	Описание
Уведомления кнопки	
BN_CLICKED	Пользователь щелкнул на кнопке
BN_DISSABLE	Кнопка переведена в неактивное состояние
BN_DOUBLECLICKED	Пользователь дважды щелкнул на кнопке
BN_PAINT	Кнопка должна быть перерисована
Уведомления раскрывающегося списка	
CBN_CLOSEUP	Раскрытый список был закрыт пользователем
CBN_DBLCLICK	Пользователь дважды щелкнул на строке
CBN_DROPDOWN	Список был раскрыт
CBN_EDITCHANGE	Пользователь изменил текст в поле ввода

Продолжение табл. 10.1

Уведомление	Описание
CBN_EDITUPDATE	Требуется вывести измененный текст
CBN_KILLFOCUS	Список потерял фокус ввода
CBN_SELCHANGE	Выбран новый элемент списка
CBN_SELENDCANCEL	Пользователь отказался от сделанного им выбора
CBN_SETFOCUS	Список получил фокус ввода
Уведомления поля ввода	
EN_CHANGE	Требуется обновление после внесения изменений
EN_ERRSPACE	Элементу управления не хватает памяти
EN_HSCROLL	Пользователь щелкнул на горизонтальной полосе прокрутки
EN_KILLFOCUS	Поле ввода потеряло фокус ввода
EN_MAXTEXT	Введенный текст был обрезан
EN_SETFOCUS	Поле ввода получило фокус
EN_UPDATE	Требуется отображение введенного текста
EN_VSCROLL	Пользователь щелкнул на вертикальной полосе прокрутки
Уведомления списков	
LBN_DBLCLICK	Пользователь дважды щелкнул на строке
LBN_KILLFOCUS	Список потерял фокус ввода
LBN_SETCANCEL	Отмена выделения
LBN_SELCHANGE	Изменение выделения
LBN_SETFOCUS	Список получил фокус ввода

Для того чтобы увидеть, как передаются эти уведомления, можно создать форму с несколькими кнопками и надписью для вывода информации, а затем ввести обработчик сообщений. Однако имеются затруднения непосредственного перехвата указанных уведомляющих сообщений, так как они относятся к внутренним инструментам операционной системы. Поэтому для демонстрации мы воспользуемся сообщением **WM_COMMAND**.

Сообщение **WM_COMMAND** среди прочих применений используется, когда элемент управления посылает уведомительное сообщение родительскому окну.

Параметры **TMessage** сообщения **WM_COMMAND** содержат следующую информацию:

- **HIWORD(WParam)** — старшее слово **WParam** содержит код уведомления, если сообщение пришло от элемента управления;
- **LOWORD(WParam)** — младшее слово **WParam** определяет идентификатор (внутренний) элемента управления;
- **LParam** — дескриптор отправившего сообщение элемента управления.

Далее приведены в сокращении исходные тексты главной формы приложения и обработчика сообщения:

```
class TForm2 : public TForm
{
__published:
        TLabel *Label1;
TLabel *Label2;
        TLabel *Label3;
        TLabel *Label4;
        TButton *Button1;
```

```
        TEdit *Edit1;
        TRadioButton *RadioButton1;
        TScrollBar *ScrollBar1;
        void __fastcall Button1Click(TObject *Sender);
private:
void __fastcall BTNMessage(TMessage &Mes);
public:
        __fastcall TForm2(TComponent* Owner);
        BEGIN_MESSAGE_MAP
                MESSAGE_HANDLER(WM_COMMAND, TMessage, ONMessage)
        END_MESSAGE_MAP(TComponent)

};
void __fastcall TForm2::BTNMessage(TMessage &Mes)
{
        Label1–>Caption=«WM_COMMAND=»+IntToStr((int)Mes.Msg);
        Label2–>Caption=«Код уведомления= HIWORD(Mes.WParam)=»
                        +IntToStr((int)HIWORD(Mes.WParam));
        Label3–>Caption= «Идентификатор элемента
управления=LOWORD(Mes.WParam)=»+ IntToStr((int)LOWORD(Mes.WParam));
        Label4–>Caption=«Дескриптор элемента управления=Mes.LParam
=»+IntToStr((int)Mes.LParam);
}
```

Наш эксперимент с приведенной программой заключается в отслеживании щелчка на элементах управления. На рисунке 10.6 приведен вид экранной формы после щелчка на элементе управления **ScrollBar1**.

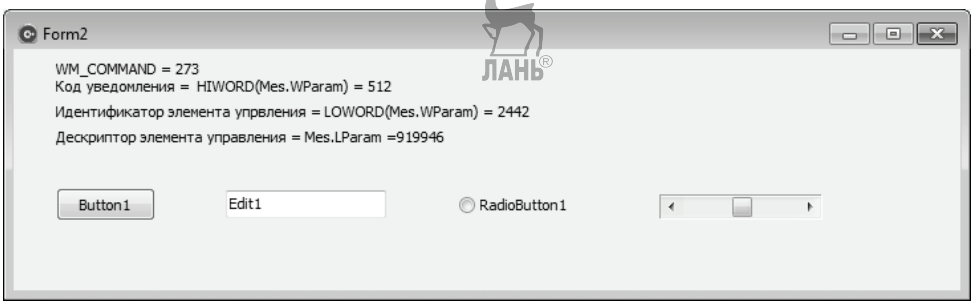


Рис. 10.6
Вид экранной формы

Контрольные вопросы

1. Какие сообщения называются уведомляющими?

10.4.4. Включение событий в программу

Создание новых событий не особенно сложно, но требует некоторой осмотрительности. Рассмотрим пошаговую процедуру создания нового события.

1. Тщательно обдумываем, в чем состоит суть события и какие параметры должны передаваться в его обработчик.
2. Создаем указатель на функцию обработчика события.

3. Создаем виртуальный метод, вызывающий событие.
4. Объявляем поле данных события в скрытом разделе класса компонента.
5. Объявляем событие.
6. Разрабатываем код, генерирующий событие.

Рассмотрим каждое действие более подробно.

Наиболее простая форма события — это событие, просто уведомляющее о том, что оно произошло. Например, событие OnClick, которое присутствует в большинстве визуальных компонентов, имеет вид:

```
void __fastcall TForm1::Button1Click (TObject * Sender)
{
}.
```

ЛАНЬ®

Этот тип события определяется в библиотеке визуальных компонентов как тип **TNotifyEvent**, который представляет собой указатель на функцию. Стандартный тип такого указателя на функцию имеет вид:

```
typedef void __fastcall (__closure *TNotifyEvent) (System::TObject* Sender);.
```

Специальный тип указателя на функцию вводится с помощью служебного слова **__closure**. В стандартном варианте этот тип используется для событий, которые передают обработчику только один обязательный параметр **TObject * Sender**. Поле хранения ссылки на функцию-обработчик события создается обычно в скрытом разделе класса и имеет вид

```
TNotifyEvent FNotifyEvent;
```

Для доступа к обработчику события рекомендуется включать в открытый раздел класса свойство.

Например, мы хотим ввести событие **OnClear**, происходящее в момент очистки окна методом **Clear**. В этом случае объявления нашего события **OnClear** могут иметь вид:

```
private:
    TNotifyEvent FOnClear;
__published:
   _property TNotifyEvent OnClear={read=FOnClear , write=FOnClear };
```

Эти объявления создают поле **FOnClear** типа **TNotifyEvent**, соответствующее событию. А само событие объявляется точно так же, как любое свойство. Только в описатели **read** и **write** записываются не имена функций для чтения и записи, а само поле. Такой прием получил название закрытого вызова метода.

Прежде чем использовать свойство, полю нужно присвоить имя функции-обработчика. Это действие обычно включается в конструктор класса. В нашем примере присваивание может иметь вид

```
FOnClear=MyMethod;
```

ЛАНЬ®

где **MyMethod** — имя функции-обработчика события.

Остается только вызвать в нужный момент обработчик события пользователя, если программист его предусмотрел. Рекомендуется выполнять проверку: имеется ли обработчик пользователя. В нашем примере эта проверка может быть выполнена с помощью оператора

```
if (OnClear) OnClear (this);
```

который вызывает обработчик **MyMethod**.

В некоторых программах одно и то же событие должно вызывать в разных условиях разную реакцию программы. Для динамического изменения способа обработки событий используется прием изменения текущего обработчика (ссылки на функцию). Допустим, надо программно создать обработчик нажатия кнопки **Button1**. Для иллюстрации нам понадобится описание класса (включаем только необходимые для нашего примера описания).

```
class TForm1 : public TForm
{
    __published:
        TButton * Button1;
        // добавляем вручную
        void __fastcall MyClick (TObject * Sender);
    public:
        __fastcall TForm1(TComponent * Owner);
}
```

В реализацию конструктора формы занесем в свойство **OnClick** кнопки ссылку на функцию **MyClick**. Таким образом организуется реакция этой кнопки на нажатие. Подобное динамическое связывание можно выполнить в любом подходящем месте программы, для того, чтобы не усложнять пример, мы включим это действие в конструктор формы:

```
__fastcall TForm1::TForm1(TComponent * Owner)
{
    Button1–>OnClick=MyClick;
}
```

В программе можно поддерживать одновременно несколько обработчиков интересующего события. Ссылки на соответствующие процедуры обработки подставляются в свойство **OnClick** (или другие) элемента управления в зависимости от выполнения каких-то условий.

В заключение рассмотрим пример. Мы хотим на форму поместить часы. Источником времени у нас будет таймер. Вид экранной формы на стадии проектирования представлен на рисунке 10.7.

На рисунке 10.7 поле «Тик» фиксирует количество срабатываний таймера.

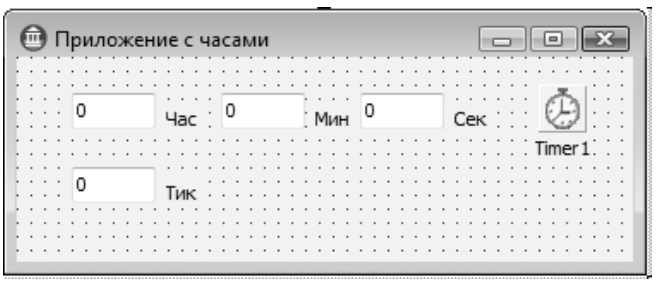


Рис. 10.7
Экранная форма приложения с часами

Реализация задачи «Приложение с часами» выполнена в двух программных модулях: модуль главной формы и модуль часов. Модуль главной формы служит для отображения времени на экранной форме. В модуле часов происходит пересчет показаний таймера в часы, минуты и секунды. Сначала приведем исходный текст приложения, а потом обсудим использованные инструменты и приемы.

```
// Заголовочный файл модуля главной формы задачи «Приложение с часами»
//---------------------------------------------------------------------------
#ifndef uMainFormH
#define uMainFormH
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ExtCtrls.hpp>
#include «uWatch.h»
class TClockForm : public TForm
{
__published:
        TTimer *Timer1;
        TEdit *edtFHour;
        TEdit *edtFSecond;
        TLabel *lblFHour;
        TLabel *lblFSecond;
        TEdit *edtFMinute;
        TLabel *lblFMinute;
        TEdit *edtFTick;
        TLabel *lblTick;
        void __fastcall Timer1Timer(TObject *Sender);
private:
TOnTimeIncrement FOnTimeIncrement;
void __fastcall OnTimeIncrement(TMessage &Mes);
public:
BEGIN_MESSAGE_MAP
  VCL_MESSAGE_HANDLER(WM_SECOND,TMessage, FOnTimeIncrement);
  VCL_MESSAGE_HANDLER(WM_MINUTE,TMessage, FOnTimeIncrement);
  VCL_MESSAGE_HANDLER(WM_HOUR,TMessage, FOnTimeIncrement);
END_MESSAGE_MAP(TComponent)
__fastcall TClockForm(TComponent* Owner);
};
extern PACKAGE TClockForm *ClockForm;
#endif
//---------------------------------------------------------------------------
// файл реализации модуля главной формы задачи «Приложение с часами»
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource «*.dfm»
```

```
#include «uMainForm.h»
#include «uWatch.h»
TClockForm *ClockForm;
TWatch * objWatch;
__fastcall TClockForm::TClockForm(TComponent* Owner)
        : TForm(Owner)
{
  objWatch=new TWatch();
  objWatch–>HandleForm =this–>Handle;
  FOnTimeIncrement=OnTimeIncrement;
  Timer1–>Interval=500;
}
void __fastcall TClockForm::Timer1Timer(TObject *Sender)
{
        objWatch–>intPTick=Timer1–>Interval;
        edtFTick–>Text=objWatch–>intPTick;
}
void __fastcall TClockForm::OnTimeIncrement(TMessage &Mes)
{
        switch (Mes.Msg)
        {
          case WM_SECOND: {edtFSecond–>Text=Mes.WParamLo; break;}
          case WM_MINUTE: {edtFMinute–>Text=Mes.WParamLo;break;}
          case WM_HOUR:   edtFHour–>Text=Mes.WParamLo;
        }
}
//---------------------------------------------------------------------------
// Заголовочный файл модуля «Часы» задачи «Приложение с часами»
//---------------------------------------------------------------------------
#ifndef uWatchH
#define uWatchH
#include <Messages.hpp>
#define WM_SECOND WM_USER
#define WM_MINUTE WM_USER+1
#define WM_HOUR WM_USER+2
typedef void __fastcall (__closure * TOnTimeIncrement)(TMessage &Mes);
class TWatch
{
  private:
int intFTick;
int intFHour;
int intFMinute;
int intFSecond;
void SetTick(int intAInterval);
 public:
HWND HandleForm;
 __property int intPTick={read=intFTick,write=SetTick};
 __property int intPHour={read=intFHour};
```

ЛАНЬ®

```
  __property int intPMinute={read=intFMinute};
  __property int intPSecond={read=intFSecond};
  __fastcall TWatch();
};
//---------------------------------------------------------------------------
// файл реализации модуля «Часы» задачи «Приложение с часами»
//---------------------------------------------------------------------------
#pragma hdrstop
#include «uWatch.h»
void TWatch::SetTick(int intAInterval)
{
  intFTick=intFTick+1;
  int Seconds=intFTick * intAInterval/1000;
  int Minutes=Seconds/60;
  int Hours  =Minutes/60;
  if ((Seconds%60) != intFSecond)
  {
        intFSecond=Seconds % 60;
        SendMessage (HandleForm, WM_SECOND, intFSecond, 0);
  }
  if ((Minutes%60) != intFMinute)
  {
        intFMinute=Minutes % 60;
        SendMessage (HandleForm, WM_MINUTE, intFMinute, 0);
  }
  if ((Hours%24) != intFHour)
  {
        intFHour  =Hours % 24;
        SendMessage (HandleForm, WM_HOUR, intFHour, 0);
  }
}
//---------------------------------------------------------------------------
__fastcall TWatch::TWatch()
{
  intFSecond=0;
  intFMinute=0;
  intFHour  =0;
  intFTick  =0;
}
#pragma package(smart_init)
```

Модуль главной формы включает описание класса **TClockForm**, с помощью которого создан объект **ClockForm**. Модуль часов содержит описание класса **TWatch** (часы), в соответствии с которым введен объект **objWatch**.

Показания времени в объекте **ClockForm** изменяются с помощью событий. События возникают в объекте **objWatch**. Событие «Изменилась секунда» генерируется через каждую секунду, событие «Изменилась минута» генерируется один раз в минуту, событие «Изменился час» генерируется на границе каждого часа.

Для обработки связанных с событиями сообщений в класс **TClockForm** включена карта сообщений, в которой каждому событию изменения времени соответствует отдельное сообщение: **WM_SECOND** — сообщение об изменении секунды, **WM_MINUTE** — сообщение об изменении минуты и, наконец, **WM_HOUR** — сообщение об изменении часа. Объявления сообщений приведены в начале заголовочного файла модуля часов с помощью сообщения **WM_USER**, которое предназначено для введения новых, произвольных сообщений.

Для иллюстрации приемов работы с параметрами сообщения мы используем в программе один обработчик для всех трех сообщений: **WM_SECOND**, **WM_MINUTE** и **WM_HOUR**. Ссылка на обработчик хранится в поле **FOnTimeIncrement**. Определение процедурного типа **TOnTimeIncrement** приведено в заголовочном файле модуля «Часы».

Работа программы происходит следующим образом. В объекте **ClockForm** с помощью функции **Timer1Timer** обрабатывается событие конца интервала таймера. Значение интервала таймера отправляется в объект **objWatch** с помощью присваивания

objWatch–>intPTick=Timer1–>Interval.

В функции **SetTick** свойства **intPTick** вычисляется общее количество прошедших секунд, минут и часов. При изменении каждой из перечисленных величин в главную форму отправляется соответствующее сообщение. Например, при изменении значения секунды отправляется сообщение путем вызова функции

SendMessage (HandleForm, WM_SECOND, intFSecond, 0).

Новое значение количества секунд отправляется в параметре сообщения, тип параметра сообщения — **TMessage**. Отправляемое значение количества секунд размещается в поле **WParamLo** параметра сообщения. Заметим, что значение **HandleForm** передается объекту **objWatch** в конструкторе главной формы.

В конструкторе главной формы полю **FOnTimeIncrement** присваивается обработчик сообщения **OnTimeIncrement**. Поэтому полученное сообщение об изменении составных частей показания времени обрабатывается в функции **OnTimeIncrement**, которая изменяет значения соответствующих полей главной формы приложения.

Контрольные вопросы

1. Какие действия необходимо выполнить для введения нового события в программу?
2. Какими средствами вводится в программу новый обработчик события?
3. Почему в примере «Приложение с часами» введены сообщения **WM_SECOND**, **WM_MINUTE** и **WM_HOUR**?

Задачи

1. Для экранной формы обработать событие: OnActivate.
2. Для экранной формы обработать событие: OnDeactivate.
3. Для экранной формы обработать событие: OnClick.

4. Для экранной формы обработать событие: OnClose.
5. Для экранной формы обработать событие: OnCreate.
6. Для экранной формы обработать событие: OnDestroy.
7. Для экранной формы обработать событие: OnKeyDown.
8. Для экранной формы обработать событие: OnKeyPress.
9. Для экранной формы обработать событие: OnMouseDown.
10. Для экранной формы обработать событие: OnMouseUp.
11. Для экранной формы обработать событие: OnMouseEnter.
12. Для экранной формы обработать событие: OnMouseMove.
13. В обработчике события OnMouseDown для класса TForm1 необходимо определять координаты нажатой клавиши и вид кнопки (левая, средняя, правая).
14. Необходимо создать класс TEdit1, порожденный от класса TEdit, в котором нужно объявить метод обработки сообщения WM_KEYDOWN. В созданном методе будем определять, относится ли введенный символ в поле Edit1 к цифре или к букве.
15. Создать два приложения. В первом приложении SendMessage в поле Edit1 вводится цифра, и при нажатии кнопки «Переслать» значение цифры отправляется с сообщением второму приложению. Во втором приложении GetMessage сообщение принимается, и значение цифры помещается в поле класса TMemo с накоплением переданных цифр.

11. ОБРАБОТКА ОШИБОК

Настоящие программисты не исправляют чужих ошибок. Они убивают их собственными.

С сайта:
http://www.aforizmov.net/tema/tags/programmist/

11.1. Метафора обработки ошибок

Обработка ошибок является существенной частью разработки программ. Программистская мудрость учит:

— нет программ без ошибок;

— любое исправление ошибок программы приводит к тому, что в нее вносятся новые ошибки.

Идеальное решение проблемы борьбы с ошибками таково: создать очень хороший язык программирования, который не будет допускать или будет обрабатывать и устранять ошибки. Очевидно, что в настоящее время это невозможно.

Существует понятие надежности языка программирования. Надежность языка программирования характеризуется степенью обнаружения ошибок на этапе трансляции программы, т. е. отношением числа обнаруженных ошибок на этапе трансляции ко всем обнаруженным ошибкам.

По методам обнаружения и обработки все ошибки можно разделить на три класса: ошибки трансляции (статические ошибки), логические ошибки и динамические ошибки.

Ошибки трансляции. Основная причина появления ошибок трансляции — это недостаточное знание языка программирования и ошибки ввода программы. Чтобы бороться с ними необходимо:

1) выучить язык программирования;
2) использовать систему контекстной подсказки;
3) при создании транслятора предусмотреть хорошую диагностику ошибок.

Логические ошибки. Основная причина появления логических ошибок — это недоработки этапа проектирования программы, непродуманность структур данных и алгоритмов. Некоторые логические ошибки устраняются достаточно легко. Например, в случае, если в приложении содержится калькулятор, в котором нажатие кнопки «Сложение» приводит к выполнению операции вычитания. Устраняется ошибка назначением кнопке «Сложение» обработчика, который на самом деле выполняет операцию сложения. В других случаях логические ошибки могут проявиться один раз из сотни запусков программы и могут быть результатом взаимодействия многих свойств и методов. Именно такой тип ошибок является наиболее трудным для устранения.

Основным способом борьбы с логическими ошибками является тестирование. В общем случае составить тест на проверку всех ветвей алгоритма при всех сочетаниях исходных данных невозможно, следовательно, невозможно обнаружить все логические ошибки.

Для борьбы с логическими ошибками была разработана техника доказательного программирования. К программе прикладывается доказательство ее

правильности[40]. Однако доказательство ее правильности намного превышает объем самой программы, и, возможно, по этой причине доказательное программирование не прижилось.

Отметим некоторые полезные приемы борьбы с логическими ошибками:

1) как правило, программист отлаживает программу с разумными исходными данными. Нужно попробовать запустить ее с неразумными данными. Например, если программа читает данные из файла, сделать файл пустым;

2) выводить и проверять результаты по шагам алгоритма с использованием отладчика.

Динамические ошибки. Динамические ошибки происходят тогда, когда программа запущена на выполнение. Динамические ошибки — это результат неполной обработки всех возможных вариантов работы приложения. Основная причина появления динамических ошибок — это непредвиденные ситуации в вычислительной системе. Хотя нужно оговориться, что довольно часто возникновение динамических ошибок связано с логическими ошибками или с тем, что программист учел не все ситуации в программе.

В качестве примера рассмотрим обработчик события создания экранной формы:

```
void __fastcall TForm2::FormCreate(TObject *Sender)
{
  TBitmap * Bitmap;
// Bitmap=new TBitmap();
  Bitmap–>Height=200;
  Bitmap–>Width=200;
  Image1–>Picture–>Graphic=Bitmap;
}
```

В приведенном примере имеется специально закомментированная строка, что приводит к динамической ошибке. Код проходит синтаксическую проверку, прекрасно компилируется, но выполняется с ошибкой. Когда он выполняется, происходит попытка обратиться к несозданному объекту и это приводит к выдаче сообщения об ошибке, представленного на рисунке 11.1.

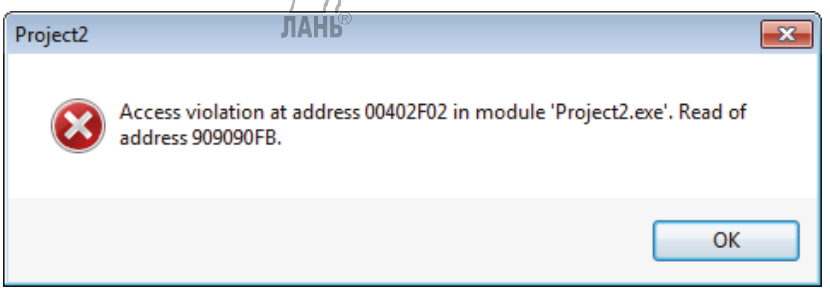


Рис. 11.1
Сообщение о динамической ошибке обращения к несуществующему адресу

[40] Грис Д. Наука программирования: пер. с англ. М.: Мир, 1984. 416 с.

Нашей задачей является не допустить появления подобных сообщений, так как пользователь нашей программы, получив такое сообщение, придет в замешательство (легко сказать!).

Существует три основных способа обработки динамических ошибок.

1. Пустить на самотек, т. е. никак не обрабатывать ошибки с надеждой, что они будут обнаружены встроенной системой обработки ошибок (рис. 11.1). Однако профессиональная гордость не должна позволять поступать таким образом.
2. Использовать технику защищенного программирования.
3. Активно использовать систему обработки ошибок.

Понятие техники защищенного программирования. Как следует из названия, защищенное программирование — это тщательное, осторожное программирование. Чтобы построить надежную программу, мы должны спроектировать каждую компоненту системы так, чтобы она была как можно лучше *защищена*. В защищенном программировании нельзя делать *никаких* допущений. Нельзя предполагать, что *некое событие никогда не случится*. Никогда нельзя предполагать, что все в мире будет происходить так, как нам бы того хотелось. Мы расправимся с неписаными допущениями, сделав для них явные проверки в коде. Таким способом мы попытаемся предотвратить или хотя бы обнаружить такое обращение к нашему коду, которое вызывает некорректное поведение.

Суть техники защищенного программирования рассмотрим с помощью примеров (рис. 11.2).

Контролируемые программистом ситуации

если делитель не равен нулю, **то** разделить **иначе**?
если файл существует, **то** открыть файл **иначе**?
если есть нужный запас оперативной памяти, **то** разместить данные в динамической памяти **иначе**?
если адрес своей программы, **то** обратиться по этому адресу **иначе**?

Рис. 11.2

Возникает вопрос: как поступать в случае «иначе»? Возможны следующие действия программиста.

1. Выдать диагностическое сообщение и остановить программу.
2. Выдать диагностическое сообщение, ликвидировать то, что «натворила» программа (например, освободить динамическую память), и остановить программу.
3. Попытаться исправить ситуацию и продолжить вычисления.

Последнее возможно очень редко: если файла нет, то ситуацию не исправить.

Приведем некоторые аргументы в пользу защищенного программирования:

а) защищенное программирование позволяет намного сократить отладку и заняться чем-то более интересным в освободившееся время. Вспомним закон Мерфи: если нашим кодом *можно* воспользоваться некорректно, так оно и случится;

б) код, который выполняется правильно, но немного медленнее, *значительно* предпочтительнее кода, который почти всегда работает правильно, но временами с большим треском «падает»;

в) иногда защищенный код проектируется так, что его можно удалить из окончательной версии, и таким образом проблема быстродействия снимается. Однако большинство приемов, которые будут здесь рассмотрены, не влекут сколько-нибудь заметных накладных расходов;

г) защищенное программирование избавляет от множества проблем безопасности, что весьма существенно в современных разработках программного обеспечения.

Поскольку рынок требует, чтобы программы писались быстро и были дешевы, мы должны сосредоточить свое внимание на технологиях, которые дают нужные результаты. Небольшой дополнительный труд вначале (если им не пренебречь) избавит в дальнейшем от массы неприятностей и задержек.

Однако рассмотренные на рисунке 11.2 примеры показывают, что использование техники защищенного программирования «захламляет» программу вспомогательными проверками.

Когда начинать программировать в защищенном стиле? После того, как выявятся неполадки? Или когда попадется непонятный код, написанный кем-то другим? Нет, технологии защищенного программирования нужно применять постоянно. Они должны войти в привычку. Опытные программисты знают это по своему опыту; они достаточно часто обжигались, и теперь размещают в программах разумные средства защиты. Защитные средства гораздо проще применять с самого начала написания кода, нежели пытаться вмонтировать в уже существующий код. Если пытаться втиснуть их в код на поздних стадиях работы с ним, трудно ничего не упустить и быть точным. Если мы начинаем добавлять защищенный код уже после того, как возникли неполадки, то фактически занимаемся отладкой, т. е. реакцией на события, а не предупреждением их. Тем не менее в процессе отладки или добавления новых функций часто обнаруживаются условия, которые желательно проверять. Вот тогда также полезно добавить защищенный код.

Использование системы обработки ошибок. В современных языках программирования используется техника обработки ошибок, получившая название «механизм обработки исключительных ситуаций».

Исключительная ситуация (исключение) — это событие периода выполнения, которое может стать причиной отказа программы.

Зачастую исключение будет причиной отказа, но можно предотвратить отказ, написав программу так, что она будет захватывать возникшее исключение, пытаться восстановить состояние, допускающее нормальное продолжение вычислений. Вот почему отказ и исключение — это разные понятия: каждый отказ — это следствие исключения, но не каждое исключение приводит к отказу.

Механизм обработки исключительных ситуации позволяет использовать технику защищенного программирования с умеренными затратами на обработку ошибок.

Чтобы использовать механизм обработки исключительных ситуаций, программисту необходимо записывать текст программы специальным образом:

try
<защищенный текст программы>
catch <квалификатор ошибки> <обработка ошибки>,

где <защищенный текст программы> содержит описание нормального течения событий, а <обработка ошибки> записывается в отдельной секции.

Обнаружение ошибок осуществляется автоматически системой программирования совместно с операционной системой. При обнаружении исключительной ситуации происходит ее классификация с помощью <квалификатор ошибки>, и управление программой передается на <обработка ошибки>. По этой схеме код обработки ошибок не смешивается с основным кодом программы. Если исключительная ситуация возникла, то у программиста имеется возможность перехватить эту ситуацию и отреагировать на нее по-своему. Имеется также возможность искусственно создавать исключительные ситуации и затем обрабатывать их. Например, обнаружение синтаксической ошибки является рядовым результатом работы транслятора и не является исключительной ситуацией. Однако здесь очень удобно искусственно создавать исключительную ситуацию с последующей выдачей диагностического сообщения, при этом используется единый механизм для всех видов ошибок.

Что представляет собой исключительная ситуация? Нетрудно догадаться, что в объектном программировании это специальный объект, работа с которым осуществляется стандартным для объектов способом:

1) возникла исключительная ситуация — создать соответствующий объект (операция выполняется автоматически);

2) поработать с объектом с целью диагностики и обработки исключительной ситуации;

3) после обработки исключительной ситуации ликвидировать объект (операция выполняется автоматически).

Контрольные вопросы

1. Что характеризует понятие «надежность языка программирования»?
2. Что представляют собой ошибки трансляции?
3. Какова причина логических ошибок?
4. Какова причина динамических ошибок?
5. Какие способы борьбы с динамическими ошибками?
6. Опишите технику защищенного программирования.

11.2. Техника защищенного программирования с использованием собственных средств C++

Язык C++ имеет собственный механизм обработки исключений, называемый *обработкой исключений* (exception handling). Назначение этого механизма в том, чтобы в момент возникновения ошибки сохранить информацию о ней в специальном объекте исключения и передать этот объект туда, где известно, как реагировать на ошибочную ситуацию.

11.2.1. Создание и перехват исключений

Механизм исключений использует три ключевых слова: throw, try и catch. Исключительная ситуация может быть искусственно создана при помощи оператора

throw <значение>,

где <значение> идентифицирует возникшее исключение и может быть любого типа. Например,

```
if (y == 0)
    throw “Деление на 0”;
z=x/y;.
```

Если исключение не перехватывается в той же функции, где оно возникло, оператор **throw** завершает функцию подобно оператору **return**.

Защищенный код, т. е. код, который может возбудить исключение, заключают в блок **try**

```
try
{
   // здесь находится защищенный код
}
```

Код обработки исключений помещается в блок **catch**, который всегда непосредственно следует за блоком **try**

```
catch (<исключение>)
{
   // здесь находится код обработки исключения
}
```

Приведем простой пример:

```
double fun(double a, b)
{
          double c;
try
   {
      if (b == 0)
              throw 0;
      double c=a/b;
    }
catch (int)
  {
     ShowMessage(«Ошибка: деление на ноль!»);
return 0;
   }
ShowMessage(“Результат=”+FloatToStr(c));
return c;
}
```

Если выполнить вызов этой функции вида **fun(5, 0)**, то получим сообщение **«Ошибка: деление на ноль!»**. Соответственно, вызов **fun(5, 2)** приведет к получению сообщения **«Результат=2.5».**

Заметим, что блок **try** реагирует не только на исключения, возбуждаемые непосредственно внутри него, но и на все не перехваченные исключения, возбуждаемые внутри функций, прямо или косвенно вызываемых из блока **try**. Приведем пример исключения, вырабатываемого в вызываемой функции:

```
double call_fun(double a, double b)
{
   if (b)
              return a/b;
   else throw 0;
}
void main_fun()
{
   try
      {
       double d=call_fun(3,0);
       ShowMessage(FloatToStr(d));
      }
  catch (int)
      {
       ShowMessage(«Ошибка: деление на ноль!»);
      }
}
```

Возникшая внутри функции **call_fun** исключительная ситуация перехватывается и обрабатывается в вызывающей функции **main_fun**. В общем случае глубина вызванных функций не имеет значения.

Контрольные вопросы

1. Какие ключевые слова используются для обработки исключительных ситуаций?
2. Приведите пример обработки исключительной ситуации «деление на ноль».

11.2.2. Обработка различных исключений

Каждый блок catch способен ловить исключения того типа, который задан в его параметре. Если нужно обрабатывать исключения разных типов, создают несколько блоков catch, которые следуют один за другим. Рассмотрим пример:

```
double x=5.0, y;
try
    {
        if (x == 0)  throw 1;
        if (x < 0)    throw «abc»;
        y=sqrt(1/x);
     }
catch (int)
     {
         ShowMessage(«Ошибка: деление на ноль!»);
      }
```

```
catch (char *)
    {
        ShowMessage(«Ошибка: отрицательное число!»);
     }
```

Однотипные исключения можно различить по их значению, модифицируем предыдущий пример:

```
double x=5.0, y;
try
    {
        if (x == 0)  throw 1;
        if (x < 0)    throw 2;
        y=sqrt(1/x);
     }
catch (int ex)
    {
        switch (ex)
              {
                  case 1:
                        ShowMessage(«Ошибка: деление на ноль!»);
                        break;
                  case 2:
                        ShowMessage(«Ошибка: отрицательное число!»);
                        break;
               }
     }
```

Как видно из последнего примера, в блоке **catch** можно указать не только тип, но и переменную исключения.

Те исключения, которые не будут обработаны в текущем блоке **try,** попадут во внешний блок **try**. Если такой блок программистом не предусмотрен, то исключение будет перехвачено предусмотренными по умолчанию средствами системы программирования. В этом случае вызывается функция **terminate()** и выполнение программы будет остановлено.

Чтобы перехватить в блоке **catch** абсолютно все исключения, его аргументом должно быть многоточие

```
catch (…)
    {
        // здесь располагается код обработки всех исключений
     }.
```

В этом случае использовать в конструкции **catch** переменную невозможно. Такой блок следует располагать последним в серии блоков **catch**.

Контрольные вопросы

1. Какой прием используется для обработки различных исключительных ситуаций?
2. Как различить в программе однотипные исключительные ситуации?
3. Как перехватить в программе все исключения?

11.2.3. Повторная обработка исключений

Обработанное блоком catch исключение автоматически уничтожается. Если необходимо продолжить обработку исключения во внешнем блоке try, его следует повторно возбудить в блоке catch оператором throw без параметра:

```
try
  {
    try
      {
        if(!x) throw 1;
          y=1/x;
      }
    catch (int m)
      {
        if (m == 1)  ShowMessage(«Ошибка: деление на ноль!»);
        throw;
      }
  }
catch (int)
  {
   ShowMessage(«Повторное сообщение: деление на ноль!»);
  }
```

Контрольные вопросы

1. Что происходит с исключением в блоке **catch**?
2. Как передать исключение во внешний блок **try**?

11.3. Техника защищенного программирования с использованием структурной обработки исключений

В литературе по операционным системам механизм обработки исключительных ситуаций получил название «структурная обработка исключений». Структурная обработка исключений позволяет отделить основную работу программы от рутинной обработки ошибок. Введение структурной обработки исключений позволило упростить разработку и повысить надежность операционной системы Windows.

Структурная обработка исключений предоставляет две возможности: *обработку завершения* (termination handling) и *обработку исключений* (exception handling).

Обработка завершения осуществляется применением языковой конструкции **try-finally**, для обработки исключений используется языковая конструкция **try-catch**. В этом параграфе мы рассмотрим реализацию структурной обработки исключений с помощью языковых конструкций **try-finally** и **try-catch**. В самом общем виде использование конструкции **try-catch** было показано в предыдущем параграфе. Теперь мы начнем с **try-finally**.

11.3.1. Использование обработчиков завершения

При работе программы используется целый ряд коллективно используемых ресурсов: динамически распределяемая оперативная память, файлы, принтеры и т. п. Обычно управление этими ресурсами происходит децентрализовано: программа запрашивает соответствующий ресурс, операционная система при наличии резерва запрашиваемого ресурса удовлетворяет запрос программы (блокируя выделенный ресурс от использования другими программами), программа использует необходимый ресурс, и, наконец, программа должна возвратить операционной системе выделенный ресурс.

Если теперь при использовании программой ресурса происходит исключительная ситуация и происходит останов программы, то операторы возврата ресурсов не выполняются. В результате в операционной системе не будет выполнена разблокировка выделенных программе ресурсов. Для корректного управления подобными ситуациями предназначена конструкция **try-finally**.

Приведем синтаксис конструкции **try-finally**.

```
try
    {
        <Оператор1>;
        < Оператор2>;
              …
        < Операторi>;
              …
        < Операторk>;
    }
__finally
    {
        < Операторk+1>;
              …
        < Операторn>;
    }
```

Между служебными словами **try** и **__finally** находится защищенный блок кода. После служебного слова **__finally** располагаются завершающие защищенный блок кода действия.

Пусть во время исполнения конструкции **try-finally** отсутствовали исключительные ситуации. Тогда исполнение фрагмента программы будет происходить в следующей последовательности:

<Оператор$_1$>, < Оператор$_2$>, … , < Оператор$_i$>, … , < Оператор$_k$>,
< Оператор$_{k+1}$>, … , < Оператор$_n$>.

Как видим, в случае отсутствия исключительной ситуации последовательно выполняются как операторы блока защищенного кода, так и операторы блока завершающих действий.

Теперь приведем последовательность выполнения операторов в случае возникновения исключительной ситуации в **< Оператор$_i$>**:

<Оператор$_1$>, < Оператор$_2$>, … , < Оператор$_i$>, < Оператор$_{k+1}$>, … , < Оператор$_n$>.

Сначала последовательно выполняются операторы блока защищенного кода до момента возникновения исключительной ситуации в операторе **<Оператор$_i$>**. Во время возникновения исключительной ситуации в **<Оператор$_i$>** управление передается первому оператору блока завершающих действий **<Оператор$_{k+1}$>**. Наконец, последовательно выполняются все операторы блока завершающих действий.

Таким образом, блок завершающих действий выполняется всегда, независимо от того, возникла или нет исключительная ситуация в защищенном блоке кода.

Рассмотренный механизм обработки исключительных ситуаций позволяет сформулировать следующие правила использования конструкции **try-finally**:

1. В блок защищенного кода помещаются операторы выделения и использования ресурсов (запись в динамическую память, открытие файлов и т. п.).
2. В блок завершающих действий помещаются операторы возврата ресурсов (освобождение динамической памяти, закрытие и (или) удаление файлов и т. п.).

Часто конструкцию **try-finally** называют блоком защиты ресурсов.

Рассмотрим содержательный пример функции языка C++.

```
void TMyClass::DoFile()
{
      FILE * F;
      try
         {
            F=fopen("a.tmp");
            DoSomething();
         }
      __finally
         {
            remove("a.tmp");
         }
}
```

В приведенном примере защищенный блок кода содержит оператор открытия файла (выделения ресурсов) **F=fopen(«a.tmp»)** и оператор обработки данных **DoSomething()**. Во время выполнения завершающих действий временно созданный файл «a.tmp» будет ликвидирован как в случае нормального завершения выполнения блока защищенного кода, так и в случае возникновения исключительной ситуации.

В целом применение обработчика завершения выглядит довольно просто. Однако возникает целый ряд усложнений при использовании внутри защищенного блока кода операций **continue** и **break**. Для иллюстрации рассмотрим пример.

```
Cardinal fun()
{
   Cardinal  cardTemp=0;
   while (cardTemp < 10)
     {
        try
          {
```

```
            if (cardTemp == 2)
              {
                 continue;     // в конец while
               }
               if (cardTemp == 3)
               {
                 break;        // выход из while
               }
            }
          __finally
              {
                 cardTemp++;
               }
          cardTemp++;
    }
    cardTemp+=10;
    return cardTemp;
}
```

Проанализируем выполнение функции **fun** шаг за шагом. Вначале переменная **cardTemp** получает значение 0. Код в блоке **try** выполняется, но ни одно из условий в операторах **if** не выполняется. Управление передается блоку **finally**, где значение переменной **cardTemp** увеличивается до 1. По завершении блока **finally** значение переменной еще раз увеличивается и становится равным 2.

На следующей итерации цикла **while** значение переменной **cardTemp** равно 2, поэтому должна выполниться операция **continue**. Если бы у нас не было блока **finally**, то бесконечный цикл обеспечен! В присутствии обработчика завершения система обнаруживает, что оператор **continue** намерен преждевременно выйти из блока **try** и передает управление блоку **finally**. Значение переменной **cardTemp** увеличивается до 3. Но следующий за блоком **finally** код не выполняется, так как управление снова передается оператору **continue**, который выполняется, и мы вновь оказываемся в начале оператора цикла.

Просматриваем третью итерацию цикла. Значение переменной **cardTemp** равно 3, поэтому должна выполниться операция **break**. Система снова перехватывает попытку преждевременного завершения блока **try** и начинается выполняться блок **finally**. Значение переменной **cardTemp** еще раз увеличивается и становится равным 4. Ситуация преждевременного выхода из блока **try** обработана, и теперь на самом деле оператор **break** передает управление за пределы цикла. После увеличения значения переменной **cardTemp** до 14 функция возвращает (**return cardTemp**) вычисленное значение.

Рассмотренный пример наводит на мысль: лучше не использовать **continue** и **break** в блоке **try**. Нам пример понадобился для того, чтобы в этом убедиться.

Синтаксис и примеры использования конструкции **try-finally** показывают, что в данном контексте нам был важен только сам факт возникновения исключительной ситуации. Часто возникает необходимость распознавания и причины возникновения исключительной ситуации.

Контрольные вопросы

1. Какие сочетания служебных слов используются для оформления обработчиков завершения?
2. Приведите синтаксис обработчика завершения и опишите семантику конструкции.
3. Почему рекомендуется использовать внутри конструкции **try-finally** операторы пропуска части кода **continue** и **break**?

11.3.2. Использование обработчиков исключения

Вначале приведем синтаксис конструкции try-catch:

```
try
    {
        <Оператор1>;
        < Оператор2>;
            …
        < Операторi>;
            …
        < Операторk>;
    }
catch    (<Исключительная ситуация1>) { <Обработчик1> }
catch    (<Исключительная ситуация2>) { <Обработчик2> }

                    …
catch    (<Исключительная ситуацияn>) { <Обработчикn> }
```

Между служебными словами **try** и первым **catch** находится защищенный блок кода. После каждого служебного слова **catch**, в круглых скобках, находится классификатор исключительной ситуации, обозначенный в тексте как **<Исключительная ситуация$_i$>**, i=1, …n. Вслед за классификатором исключительной ситуации расположен обработчик, соответствующий своему классификатору.

Пусть во время исполнения защищенного блока кода отсутствовали исключительные ситуации. Тогда исполнение фрагмента программы будет происходить в следующей последовательности:

<Оператор$_1$>, < Оператор$_2$>, … , < Оператор$_k$>.

Как видим, в случае отсутствия исключительной ситуации последовательно выполняются только блок защищенного кода.

Теперь приведем последовательность выполнения операторов в случае возникновения исключительной ситуации в **< Оператор$_i$>**:

<Оператор$_1$>, < Оператор$_2$>, … , < Оператор$_i$>.

Пусть во время выполнения **< Оператор$_i$>** произошла исключительная ситуация **<Исключительная ситуация>**. После возникновения исключительной ситуации начинается процесс ее классификации и последующей обработки:

если <Исключительная ситуация>=<Исключительная ситуация$_1$> то <Обработчик$_1$>
иначе

если <Исключительная ситуация>=<Исключительная ситуация$_2$> то <Обработчик$_2$>
иначе
…
если <Исключительная ситуация>=<Исключительная ситуация$_n$> то <Обработчик$_n$>
иначе
<предусмотренные по умолчанию системой программирования действия>.

Разумеется, что действия (предусмотренные по умолчанию системой программирования действия) находятся за пределами нашей видимости и приведены с целью повышения наглядности.

Рассмотрим теперь небольшой пример функции языка C++ с конструкцией **try-catch**:

```
int TMyClass::ConvertValue(String text)
{
      int ErrorValue=0;
      try
         {
            return StrToInt(text);
          }
      catch (EConvertError *)
         {
            ShowMessage("В функции ConvertValue неверное значение параметра!");
             return ErrorValue;
         }
}
```

В приведенном примере защищенный блок кода содержит оператор возврата, переведенного в целое число поступившего в виде строки значения параметра функции. Если во время преобразования **StrToInt(text)** произойдет исключительная ситуация **EConvertError** (ошибка преобразования), то обработчик выдаст соответствующее сообщение и функция возвратит предусмотренное при ошибке значение **ErrorValue=0**. Если же в защищенном блоке кода произойдет отличная от **EConvertError** исключительная ситуация, то выполнятся предусмотренные по умолчанию действия.

Рассмотренные синтаксис и пример использования конструкции **try-catch** показывают, что в данном контексте нам был важен не только сам факт возникновения исключительной ситуации, но и ее причина.

Контрольные вопросы

1. Какие сочетания служебных слов используются для оформления обработчиков исключения?
2. Приведите синтаксис обработчика исключения и опишите семантику конструкции.

11.3.3. Исключительная ситуация как класс

Ранее уже отмечалось, что в объектно-ориентированном программировании исключительная ситуация является объектом специального вида и, таким образом, экземпляром класса. Базовым классом для всех классов исключитель-

ных ситуаций является **Exception**. Все исключительные ситуации должны порождаться от класса **Exception** для их распознавания как исключительных ситуаций. По принятым соглашениям все порождаемые от **Exception** классы будут начинать свое имя с буквы **E**.

Класс **Exception** является прямым потомком класса **TObject**. Объявление класса **Exception** очень простое. Он объявлен в модуле **SysUtils**, для рассмотрения основного принципа использования приведем описание класса **Exception** в сокращении:

```
class Exception : public System::TObject
{
    private:
      System::UnicodeString FMessage;
    public:
      __fastcall Exception(const System::UnicodeString Msg);
      __fastcall virtual ~Exception(void);
      __property System::UnicodeString Message={read=FMessage, write=FMessage};
};
```

Объявление класса **Exception** включает объявление поля **FMessage** в секции **private**. Поле **FMessage** является строкой, которая содержит передаваемое исключительной ситуации сообщение. Значение этому полю присваивается через свойство **Message**. Остальные объявления являются конструктором и деструктором.

Конструктор **Exception(const System::UnicodeString Msg)** принимает единственный параметр, константу типа строка. Эта строка описывает исключительную ситуацию. Конструктор устанавливает значение поля **FMessage**.

Например, вызовем исключительную ситуацию:

throw Exception(«Возникла исключительная ситуация»);.

В ответ автоматически создается соответствующий объект, свойство **Message** которого будет содержать наше сообщение. Далее, если не предусмотрена специальная обработка исключительной ситуации, на экране появляется форма, представленная на рисунке 11.3.

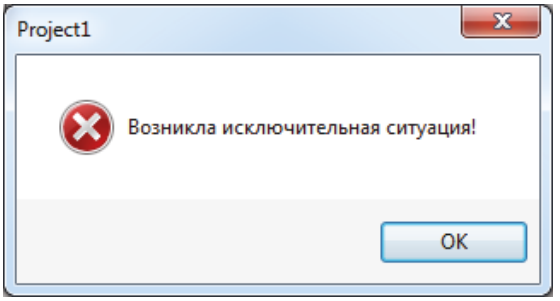


Рис. 11.3
Форма сообщения об исключительной ситуации

В RAD Studio для удобства программирования предусмотрена целая система исключительных ситуаций, порожденных от класса **Exception**. Заглянув в модуль **SysUtils.hpp** стандартной поставки RAD Studio, можно увидеть определе-

ния порожденных от класса **Exception** исключительных ситуаций. Как уже говорилось, в именовании классов используется договоренность: название классов исключительных ситуаций начинается с буквы E от Exception. На рисунке 11.4 приведена часть иерархии наследования классов исключительных ситуаций.

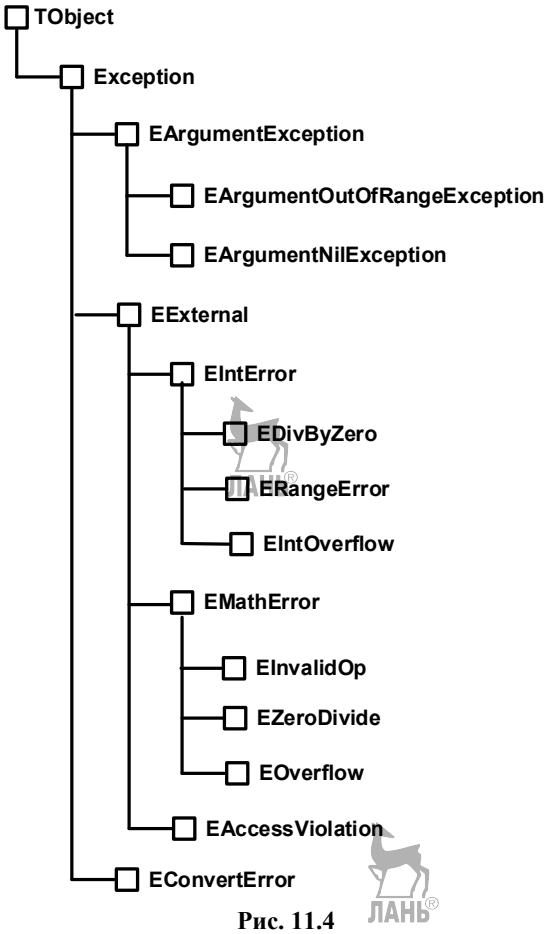


Рис. 11.4
Иерархия наследования классов исключительных ситуаций

На рисунке 11.4 приведены наиболее часто возникающие исключительные ситуации. Всего в модуле **SysUtils.hpp** находится более пятидесяти определений. Все исключительные ситуации можно разбить на категории. Например:

— класс исключительных ситуаций работы с памятью:
  - **EOutOfMemory** — нет места в куче;
  - **EInvalidPointer** — недопустимый адрес;

— класс исключительных ситуаций целочисленной арифметики:
  - **EDivByZero — деление на ноль;**
  - **EIntOverflow — переполнение разрядной сетки для целого числа;**

— класс исключительных ситуаций вещественной арифметики:
  - **EInvalidOp — недопустимая операция для вещественного числа;**
  - **EUnderflow — исчезновение порядка.**

Контрольные вопросы

1. Какой класс является базовым для всех классов исключительных ситуаций?
2. Опишите содержание класса **Exception**.
3. Какие процессы происходят при возникновении исключительной ситуации?
4. Что понимается под иерархией исключительных ситуаций?

11.3.4. Использование экземпляров исключительной ситуации

Программист для особых классов исключительных ситуаций создает экземпляры этих классов, когда исключительная ситуация возникает. Как было показано ранее, исключительная ситуация, порожденная от класса **Exception**, может быть реализована с помощью одного из предопределенных конструкторов. Самое замечательное то, что конструкторы, в общем, имеют один или более параметров, которые позволяют программисту, создающему новые классы, выбрать предопределенную строку сообщения. Эта строка будет использоваться только тогда, когда мы позволим системе программирования обработать исключительную ситуацию за нас. Кроме этого у нас нет ничего, что позволило бы получить доступ к сообщению вызванной исключительной ситуации. Однако в RAD Studio специальный синтаксис исключительной ситуации позволяет сделать это. Фактически он позволяет получить доступ к любым открытым элементам вызванной исключительной ситуации. Рассмотрим пример:

```
try
    {
      ScrollBar1–>Max=(1-2);
    }
catch (EInvalidOperation * e)
   {
      ShowMessage(e–>Message);
   }.
```

Этот пример пытается присвоить недопустимый результат вычисления свойству **Max** объекта **ScrollBar1**. В результате вызывается исключительная ситуация **EInvalidOperation** и выдается сообщение, говорящее о том, что значение свойства объекта вышло за пределы допустимого диапазона. Но обратим внимание на то, что мы отображаем диалоговое окно, но нигде не создаем это сообщение. И обратим внимание на то, что **catch (EInvalidOperation * e)** показывает новый синтаксис этой конструкции. Сочетание **EInvalidOperation * e** создает переменную **e** типа указатель на экземпляр класса **EInvalidOperation**. Когда эта переменная создана, свойство **Message** становится доступным до тех пор, пока не закончится выполнение операторов группы **catch** и не будет удален экземпляр исключительной ситуации.

Замечание. Мы не должны удалять экземпляр исключительной ситуации (в рассмотренном примере **e**), так как при обработке исключительной ситуации это происходит автоматически.

Контрольные вопросы

1. Какой синтаксис доступа к экземпляру исключительной ситуации?

11.3.5. Использование многоуровневых защищенных блоков

Иногда бывает необходимо выполнить оператор или группу операторов независимо от того, произошла исключительная ситуация или нет. Мы уже знаем, что для этого подходит конструкция **try-finally**. Но проблема состоит в том, что она фактически не обрабатывает исключительную ситуацию.

В объектном программировании имеется возможность вкладывать друг в друга конструкции try-catch и try-finally. Общих правил совместного использования блоков защищенного кода на сегодня нет. Поэтому рассмотрим такие приемы на примерах.

Пример 1. В RAD Studio имеется возможность вложить конструкцию try-finally в конструкцию try-catch. Это дает возможность, безусловно, выполнить операторы **try-finally** и дополнительно, в случае возникновения исключительной ситуации, выполнить блок **catch** конструкции try-catch. Следующий пример демонстрирует использование вложенной конструкции:

```
void __fastcall TForm2::Button1Click(TObject *Sender)
{
      class TA
          {
              public:
              int x;
          };
          class TB
          {
              public:
                  TA * A;
                  int y;
         };

         TB * B=new TB();
     //          B–>A=new TA();
   try
    {
            try
            {
              B–>A–>x=5;
              ShowMessage(IntToStr(a–>x));
              B–>y=6;
              ShowMessage(IntToStr(B–>y));
            }
           __finally
            {
              B–>~TB();
              B–>A–>~TA();
             }
```

```
}
catch (EAccessViolation &)
{
      ShowMessage(«Неверный указатель!»);
}
catch (...)
{
   ShowMessage(«Неизвестное исключение!»);
}
      }.
```

В рассмотренном примере создается объект **B** класса **TB**. Когда мы закончим работу с объектом **B**, выделенная память должна быть освобождена независимо от того, произошла исключительная ситуация или нет. Это делает вызов деструктора **B–>~TB()** первым кандидатом на включение в секцию **__finally**.

Одним из полей объекта **B** является объект **A** класса **TA**, поэтому после создания объекта **B** необходимо создать объект **A**. В нашем примере последнее действие «забыли» выполнить (действие **B–>A=new TA()** закомментировано), что приводит к исключительной ситуации при первой попытке обращения к содержимому объекта **A**. В результате возникает исключительная ситуация **EAccessViolation**. Затем выполняются операторы из секции **__finally**. Так как конструкция try-finally фактически не обрабатывает исключительную ситуацию, автоматически происходит выполнение операторов из секций **catch**. В данном случае распознается исключительная ситуация **EAccessViolation** и выводится сообщение **«Неверный указатель!»**

Пример 2. Рассмотрим теперь пример, когда внутри конструкции try-catch находится снова конструкция try-catch:

```
try
    {
        try
            {
                throw EDivByZero(««);
            }
        catch (EIntError &)
            {
                throw;
            }
        catch (...)
            {
               ShowMessage(«Неизвестное исключение»);
            }
    }
catch (EDivByZero &)
    {
        ShowMessage(«Деление на ноль»);
    }
catch (...)
```

```
    {
        ShowMessage(«Неизвестное  EIntError исключение»);
    }
```

Этот пример показывает преимущество вложенных блоков кода try-catch. Совершенно нормально, когда они используются вместе с оператором **throw**, добавляя возможность подключения еще нескольких уровней обработки исключительной ситуации. Ошибка создается оператором **throw EDivByZero(«»)**. Сначала выполняется внутренний оператор **catch**. Если исключительная ситуация имеет тип **EIntError** или ее потомков, то она вызывается. Если она другого типа, то выполняется **catch (...)**, который отображает сообщение о том, что исключительная ситуация является неизвестной. Поскольку пример генерирует исключительную ситуацию **EIntError**, будет вызвана исключительная ситуация и с помощью оператора **throw** передана на следующий уровень. Следующий уровень более детально оценивает исключительную ситуацию. Если исключительной ситуацией является **EDivByZero**, то отображается сообщение **ShowMessage(«Деление на ноль»)**. В противном случае отображается сообщение **ShowMessage(«Неизвестное EIntError исключение»)**, которое уведомляет о том, что мы имеем дело с неидентифицированным потомком исключительной ситуации **EIntError**.

Понимание рассмотренного примера должно привести к осознанию преимуществ обработки исключительных ситуаций на уровне классов.

Пример 3. Мы можем включить конструкцию try-finally как внешнюю по отношению к конструкции try-catch. Модифицируем идею примера 1 с добавлением новых возможностей:

```
void __fastcall TForm2::Button1Click(TObject *Sender)
{
     class TA
     {
       public:
         int x;
     };
     class TB
     {
       public:
       TA * A;
       int y;
     };
     TB * B=new TB();
     try
     {
          try
          {
//          B–>A=new TA();
            B–>A–>x=5;
            ShowMessage(IntToStr(B–>A–>x));
            B–>y=6;
```

```
            ShowMessage(IntToStr(B–>y));
            MessageBeep(0);
          }
          catch (EAccessViolation&)
          {
              ShowMessage(«Неверный указатель!»);
              GlobalExceptionHandler();
          }
          catch (...)
          {
             throw;
          }
    }
    __finally
     {
         B–>A–>~TA();
         B–>~TB();
     }
}
```

Этот пример выбирает для выполнения определенную пользователем глобальную функцию обработки ошибок **GlobalExceptionHandler()** в том случае, если происходит очень серьезная исключительная ситуация, такая как **EAccessViolation** или порожденная от **EAccessViolation**. Глобальный обработчик может записать некоторые данные, закрыть файлы и прервать программу. Если произошедшая исключительная ситуация не является **EAccessViolation**, то она не обрабатывается. Тогда исключительная ситуация переходит на следующий уровень операторов try-catch или try-finally, которые она встречает. В данном случае выполняются деструкторы из секции **finally**, а оператор **MessageBeep(0)** (раздается «пик» или «бип») игнорируется.

Так как обработчик **finally** на самом деле не обрабатывает исключительную ситуацию, RAD Studio продолжает искать для выполнения другой уровень обработчиков завершения try-catch или try-finally. Этот процесс продолжается до тех пор, пока исключительная ситуация фактически не будет обработана. Если обработчиков завершения больше нет, то исключительная ситуация обрабатывается предопределенным обработчиком. Если, как в этом примере, настоящая исключительная ситуация никогда не произойдет, то будет выполнена операция **MessageBeep(0)** и затем — секция **finally**. Если операция **MessageBeep** сама станет причиной исключительной ситуации, то все равно будут выполнены операторы из секции **finally**, а исключительная ситуация будет искать соответствующую обработку.

Пример 4. В качестве последнего примера рассмотрим содержание главной функции RAD Studio-программы:

```
int WINAPI _tWinMain(HINSTANCE, HINSTANCE, LPTSTR, int)
{
        try
        {
                Application–>Initialize();
```

```
        Application–>MainFormOnTaskBar=true;
        Application–>CreateForm(__classid(TForm2), &Form2);
        Application–>Run();
    }
    catch (Exception &exception)
    {
        Application–>ShowException(&exception);
    }
    catch (...)
    {
        try
        {
            throw Exception(««);
        }
        catch (Exception &exception)
        {
            Application–>ShowException(&exception);
        }
    }
    return 0;
}
```

Выполнение приложения **Application** заключено в блок защиты кода try-catch с перехватом наиболее универсального исключения **Exception**. Тем самым обеспечивается реакция на любую необработанную внутри приложения исключительную ситуацию. В случае, когда и это (Exception) исключение окажется неопознанным, в программе предусмотрен блок **catch (...)**. Во вложенном блоке try-catch искусственно вызывается исключительная ситуация **throw Exception(««)**, которая затем обрабатывается. Такой прием обеспечивает гарантированную обработку любой исключительной ситуации, возникшей в приложении.

В рассмотренном коде используется функция **ShowException()** для отображения окна сообщения с описанием возникшей ошибки. Обычно эта стандартная функция используется для выдачи сообщений о возникших ошибках приложения, но мы можем с равным успехом использовать ее внутри приложения. Чтобы вывести окно сообщения об ошибке, нужно передать объект **exception** класса **Exception** функции **ShowException**: **Application–>ShowException(&exception)**.

Контрольные вопросы

1. Объясните понятие «многоуровневый защищенный блок кода».
2. Какой прием используется в главной функции приложения для гарантированной обработки любой ошибки программы?
3. Как можно использовать стандартное окно выдачи сообщений об ошибках в программе?

11.3.6. Создание новых исключительных ситуаций

При более внимательном рассмотрении модуля SysUtils откроются несколько классов, порожденных непосредственно от класса **Exception**, а именно: **ERangeError**, **EStackOverflow**, **EMonitor** и др. При использовании системы

программирования нам, конечно, могут понадобиться не только определенные в модуле SysUtils исключительные ситуации.

Создание собственных типов исключительных ситуаций приемлемо даже в простом приложении.

Новый класс исключительной ситуации порождается от базового класса Exception или его потомков. Новый класс исключительной ситуации должен обеспечивать более конкретную информацию о существующей ошибке или даже описывать новую ветвь условий возникновения ошибок. Например, создание нового класса исключительной ситуации начинается введением класса:

```
class ECustomException : public Exception
{
   private:
int intFCustomProperty;
   public:
     __property int intPCustomProperty={read=intFCustomProperty,
write=intFCustomProperty};
ECustomException(String strAMessage);
}
```

В исключительных ситуациях, как и в других классах, могут быть добавлены элементы, переопределены конструкторы и изменены предопределенные свойства.

Работая с этим примером исключительной ситуации, мы можем вызывать ее следующим образом:

```
if (ErrorCondition) throw ECustomException(«Произошла пользовательская
исключительная ситуация»);
```

Этот пример просто проверяет гипотетическую переменную **ErrorCondition** типа **bool**. Если значение переменной равно **True**, то с помощью ключевого слова **throw** вызывается конструктор **ECustomException** исключительной ситуации. В конструктор передается строка, описывающая обстоятельства возникновения исключительной ситуации.

Класс исключительной ситуации, который мы создаем, имеет смысл разместить в самых глубоких уровнях кода нашего приложения. Это значит, что исключительная ситуация может быть вызвана на нескольких уровнях. Мы можем определить все наши исключительные ситуации в отдельном модуле (рекомендуется) и добавить его имя в конструкцию **#include** любого приложения, которое использует исключительные ситуации. Любой код высокого уровня, который вызывает исключительную ситуацию, может перехватить вызванную исключительную ситуацию при помощи блоков **try**. Обработка исключительной ситуации, созданной нами, осуществляется точно так же, как и обработка исключительных ситуаций, которые были созданы в системе программирования. Если исключительная ситуация не была обработана, то она вызывает предопределенный обработчик, который и сообщает об ошибке.

Рассмотрим более подробный пример создания собственных исключительных ситуаций.

Наш пример использует актуальную идею проверки данных. Общий вид экранной формы ввода данных приведен на рисунке 11.5. На форме находится три поля ввода. Условия ввода данных приведены в сопровождающих надписях.

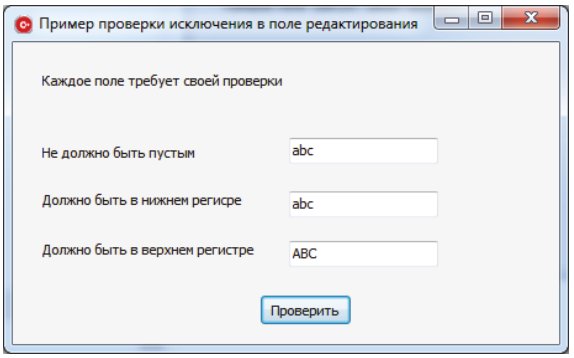


Рис. 11.5
Экранная форма для ввода данных

Рассмотрим содержание заголовочного файла:

```
enum TValidationType {NotEmpty,LowerRegister,UpperRegister};
//---------------------------------------------------------------------------
class EValidationError : public EAbort
{
 private:
      TValidationType enumFValidationType;
 public:
   __fastcall EValidationError(const String strAMessage,TValidationType enu-
mAValidationType);
};
//---------------------------------------------------------------------------
class TValidateEdit : public TEdit
{
 private:
      TValidationType enumFValidationType;
 public:
     __fastcall TValidateEdit(TComponent* Owner, TValidationType
enumAValidationType);
       virtual void ValidateData();
       bool IsLowerRoutine();
       bool IsUpperRoutine();
};
//---------------------------------------------------------------------------
class TfrmMain : public TForm
{
__published:        // IDE-managed Components
        TLabel *lblFHeading;
        TLabel *lblFNotEmpty;
        TLabel *lblFLowerRegister;
        TLabel *lblFUpperRegister;
```

```
        TButton *btnFCheck;
        void __fastcall btnFCheckClick(TObject *Sender);
private:
        TValidateEdit *edtFNotEmpty;
        TValidateEdit *edtFLowerRegister;
          TValidateEdit *edtFUpperRegister;
        __fastcall TfrmMain(TComponent* Owner);
}
```

В начале заголовочного файла введением типа **TValidationType** перечисляются виды проверок (**NotEmpty** — поле не должно быть пустым, **LowerRegister** — данные должны быть записаны в нижнем регистре, **UpperRegister** — данные должны быть записаны в верхнем регистре).

Далее приведено описание класса **EValidationError**. Класс **EValidationError** создается как потомок класса исключительной ситуации **EAbort**. Исключительная ситуация **EAbort** является, в свою очередь, потомком класса **Exception,** однако ее поведение отличается от поведения экземпляров родительского класса и его потомков тем, что она игнорируется предопределенным обработчиком исключительных ситуаций. Поэтому исключительная ситуация **EAbort** получила в литературе название «тихая исключительная ситуация» **EAbort** используется преимущественно как способ выхода из неудавшейся операции обработки данных. Поясним сказанное на примере. Пусть в цикле выполняется обработка каких-либо данных. Если во время выполнения цикла произойдет исключительная ситуация, то для продолжения вычислений мы обрабатываем исключение и продолжаем выполнение цикла. Обычно тип **EAbort** используется тогда, когда исключительная ситуация не должна действовать как ошибка приложения, но должна способствовать работе приложения. В нашем примере нарушение договоренности о форме значений вводимых данных не является ошибкой приложения. Однако приложение гарантирует правильную обработку данных только в случае, если указанные договоренности не нарушены.

В секции **public** добавляется новый конструктор **EValidationError()**. В конструктор **EValidationError** передается строка, описывающая ошибку вместе со значением, соответствующем типу проверки, при проведении которой произошла ошибка.

Листинг продолжается объявлением нового класса **TValidateEdit**, который порожден от класса **TEdit**. Класс **TValidateEdit** имеет новый конструктор, который требует передачи идентификатора владельца в первом параметре и вида проверки во втором. Значение второго параметра присваивается защищенному полю **enumFValidationType**. Далее в коде его значение влияет на результат проверки в методе **ValidateData**. Секция **public** класса **TValidateEdit** продолжается объявлением реальных методов проверки:

— **IsLowerRoutine** — возвращает значение **True**, если все символы являются малыми буквами;

— **IsUpperRoutine** — возвращает значение **True**, если все символы являются заглавными буквами.

Далее секция **public** класса **TfrmMain** содержит три управляющих элемента типа **TValidateEdit**: **edtFNotEmpty**, **edtFLowerRegister**, **edtFUpperRegister**.

Поскольку **TValidateEdit** не является компонентом, его экземпляр должен быть создан динамически. Эти действия включены в конструктор класса **TfrmMain**. Для каждого элемента управления устанавливается разное значение параметра **enumAValidationType**. Свойство **Parent** для каждого экземпляра должно быть установлено в **this**, а также должны быть определены свойства **Left** и **Top**.

Рассмотрим теперь текст реализационного файла:

```
#include <vcl.h>
#pragma hdrstop
#include «ufrmMain.h»
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource «*.dfm»
TfrmMain *frmMain;
//---------------------------------------------------------------------------
__fastcall TfrmMain::TfrmMain(TComponent* Owner)
        : TForm(Owner)
{
  edtFNotEmpty=new TValidateEdit(this, NotEmpty);
  edtFNotEmpty–>Parent=this;
  edtFNotEmpty–>Left=224;
  edtFNotEmpty–>Top =77;
  edtFLowerRegister=new TValidateEdit(this, LowerRegister);
  edtFLowerRegister–>Parent=this;
  edtFLowerRegister–>Left=224;
  edtFLowerRegister–>Top=120;
  edtFUpperRegister=new TValidateEdit(this, UpperRegister);
  edtFUpperRegister–>Parent=this;
  edtFUpperRegister–>Left=224;
  edtFUpperRegister–>Top=160;
  edtFNotEmpty–>Text=«abc»;
  edtFLowerRegister–>Text=«abc»;
  edtFUpperRegister–>Text=«ABC»;
}
//---------------------------------------------------------------------------
__fastcall EValidationError::EValidationError(const String strAMessage, TValida-
tionType enumAValidationType): EAbort(strAMessage)
{
  enumFValidationType=enumAValidationType;
}
//---------------------------------------------------------------------------
__fastcall TValidateEdit::TValidateEdit(TComponent* Owner, TValidationType
enumAValidationType):TEdit(Owner)
{
  enumFValidationType=enumAValidationType;
}
//---------------------------------------------------------------------------
void TValidateEdit::ValidateData()
```

```
{
  String s;
  switch (enumFValidationType)
  {
 case NotEmpty      :
      {
       if (this–>Text.Length() < 1)
           throw EValidationError(«Первое данное не должно быть пустым!»,
NotEmpty);
        else s=«Первое»;
            break;
      };
  case LowerRegister :
       {
        if (!IsLowerRoutine())
          throw EValidationError(«Второе данное должно быть малыми буквами!»,
                                  LowerRegister);
         else  s=«Второе»;
           break;
         };
  case UpperRegister :
       {
        if (!IsUpperRoutine())
          throw EValidationError(«Третье данное должно быть заглавными буквами!»,
                                                        UpperRegister);
         else s=«Третье»;
           break;
         };
  }
  ShowMessage(s+«данное верно!»);
}
//---------------------------------------------------------------------------
bool TValidateEdit::IsLowerRoutine()
{
  for (int i=1; i <= this–>Text.Length(); i++)
          if (this–>Text[i] < 'a' || this–>Text[i] > 'z') return False;
  return True;
}
//---------------------------------------------------------------------------
bool TValidateEdit::IsUpperRoutine()
{
    for (int i=1; i <= this–>Text.Length(); i++)
          if (this–>Text[i] < 'A' || this–>Text[i] > 'Z') return False;
    return True;
}
//---------------------------------------------------------------------------
void __fastcall TfrmMain::btnFCheckClick(TObject *Sender)
{
```

```
        for (int i=0; i < this–>ComponentCount; i++)
        {
                try
                {
                        if (this–>Components[i]–>ClassParent()–>ClassNameIs(«TEdit»))
                                ((TValidateEdit*)this–>Components[i])–>ValidateData();
                }
                catch (EValidationError * E)
                {
                        ShowMessage(E–>Message);
                }
        }
}
//---------------------------------------------------------------------- .
```

Когда мы нажимаем кнопку **btnFCheck** (Проверить), вызывается метод **btnFCheckClick**. Основным содержание метода **btnFCheckClick** является цикл по компонентам экранной формы[41]. Если текущий компонент является потомком класса **TEdit**, то вызывается метод **ValidateData** для проверки корректности введенной в поле информации. Вызов находится внутри конструкции **try**…**catch**. Если в методе **ValidateData** возникнет исключительная ситуация, связанная с некорректностью введенных данных, то она обрабатывается в секции **catch** и цикл продолжает выполняться.

Содержание других функций достаточно очевидно, поэтому предлагаем изучить их самостоятельно.

Контрольные вопросы

1. Приведите пример вычислительной ситуации, для которой целесообразно создавать новый класс исключительной ситуации.
2. Какие описания нужно создавать при введении нового класса исключительной ситуации?
3. В каких местах программного кода целесообразно вводить новые классы исключительных ситуаций?
4. Чем операция throw отличается от операции return?

Задачи

Функции с обработкой исключений.

Во всех заданиях реализуемые функции или методы должны генерировать подходящие исключения. Обработку исключений нужно выполнять главной функцией, которая должна демонстрировать обработку всех перехватываемых исключений.

Функции, реализуемые в заданиях, обязаны выполнять проверку передаваемых параметров и генерировать исключение в случае ошибочных данных. Все функции реализуются в четырех вариантах:

[41] В приведенном примере используется предок класса TForm — TComponent. Подробнее об использовании класса TComponent можно узнать в главе 12 «Библиотека визуальных компонентов».

— без спецификации исключений;
— со спецификацией throw();
— с конкретной спецификацией с подходящим стандартным исключением;
— спецификация с собственным реализованным исключением.

Собственное исключение должно быть реализовано в трех вариантах: как пустой класс, как независимый класс с полями-параметрами функции, как наследник от стандартного исключения с полями. Перехват и обработку исключений должна выполнять главная функция.

1. Функция вычисляет площадь треугольника по трем сторонам

$$S = \sqrt{p(p-a)(p-b)(p-c)},$$

где $p=(a+b+c)/2$.

2. Функция вычисляет корень линейного уравнения $ax+b=0$.
3. Функция вычисляет периметр треугольника.
4. Функция переводит часы и минуты в секунды.
5. Функция вычисляет корень квадратного уравнения $ax^2+bx+c=0$.
6. Функция вычисляет сумму геометрической прогрессии

$$S_n=(a_o - a_n \cdot r)/(1-r).$$

7. Функция выполняет деление нечетких чисел A и B. Числа представлены структурой с двумя полями (см. задание 1.31).
8. Функция вычисляет нечеткое число, обратное заданному. Числа представлены структурой с двумя полями (см. задание 1.31).
9. Функция выполняет деление комплексных чисел A и B. Комплексные числа представлены структурой-парой (см. задание 1.21).
10. Функция вычисляет целую часть неправильной дроби, представленной числителем и знаменателем — целыми числами.
11. Функция переводит комплексное число $z=x+i \cdot y$ из алгебраической формы в тригонометрическую $z=$radius(cos(angle)$+i \cdot$sin(angle)). Комплексное число z представлено структурой-парой (см. задание 1.21). Преобразованное число тоже представляется структурой-парой (radius, angle):

$$radius = \sqrt{x^2 + y^2}; \text{angle} = \arcsin\frac{y}{x}.$$

12. Функция вычисляет разность между двумя датами в днях. Даты представлены структурой с тремя полями: год, месяц, день.
13. Функция вычисляет продолжительность телефонного разговора в минутах, принимая время начала и окончания. Время представлено структурой с тремя полями: час, минута, секунда. Неполная минута считается за полную.
14. Функция вычисляет день недели по дате. Даты представлены структурой с тремя полями: год, месяц, день. Первое января считается понедельником.
15. Функция вычисляет углы прямоугольного треугольника. В качестве параметров передаются катеты A и B. (Синус угла $A1$, противолежащего катету A, вычисляется по формуле $\sin(A1)=a/c$, где c — гипотенуза треугольника.)
16. Функция проверяет, является ли передаваемая строка палиндромом.

17. Функция определяет, существуют ли прямые $A_1\ x+B_1\ y+C_1=0$ и $A_2\ x+B_2\ y+C_2=0$, если выражение $d=A1B2 - A2B1$ не равно нулю. Прямые задаются структурой с тремя полями.

18. Функция вычисляет расстояние между двумя точками $P1(x_1,\ y_1)$ и $P2(x_2, y_2)$ по формуле

$$D = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Исключение генерируется, когда P_1 и P_2 — одна и та же точка.

19. Функция вычисляет расстояние от точки $P(x_1,\ y_1)$ до прямой $Ax+By+C$=O по формуле

$$D = \frac{|Ax_1 + By_1 + C|}{\sqrt{A^2 + B^2}}.$$

20. Функция выясняет, является ли год високосным. Високосность определяется следующим образом: если номер года не делится на 100, то високосным считается тот, который делится на 4 без остатка; если номер года делится на 100, то номер високосного года делится на 400 без остатка.

Классы с обработкой исключений

Реализовать обработку исключений в трех вариантах:

- использовать подходящие стандартные исключения;
- использовать исключения-наследники от стандартных исключений;
- определить собственные исключения.

Передачу объекта-исключения выполнить разными способами: по значению, по ссылке, по указателю. Конструкторы и методы, генерирующие исключения, должны быть объявлены со спецификацией исключений.

Выполнить задания с конструкторами и обработкой исключений.

1. Комплексное число представляется парой действительных чисел (a, b), где a — действительная часть, b — мнимая часть. Реализовать класс TComplex для работы с комплексными числами. Обязательно должны присутствовать операции:

— сложения

$$(a, b)+(c, d)=(a+c, b+d);$$

— вычитания

$$(a, b) - (c, d)=(a - c, b - d);$$

— умножения

$$(a, b)\times(c, d)=(ac+bd, ad+bc);$$

— деления

$$(a, b)/(c, d)=(ac+bd, bc+ad)/(c2+d2);$$

— сравнение

$(a, b)=(c, d)$, если $(a=c)$ и $(b=d)$;

— сопряженное

(a, b) число $(a, -b)$.

2. Создать класс TVector3D, задаваемый тройкой координат. Обязательно должны быть реализованы: сложение и вычитание векторов, произведение

векторов, скалярное произведение векторов, умножение на скаляр, сравнение векторов, вычисление длины вектора, сравнение длины векторов.

3. Создать класс TModeWindow для работы с моделями экранных окон. В качестве полей задаются: заголовок окна, координаты левого верхнего угла, размер по горизонтали, размер по вертикали, цвет окна, состояние «видимое/невидимое», состояние «с рамкой/без рамки». Координаты указываются в целых числах. Реализовать операции: передвижение окна по горизонтали, по вертикали; изменение высоты и/или ширины окна; изменение цвета; изменение состояния, опрос состояния. Операции передвижения и изменения размера должны осуществлять проверку на пересечение границ экрана. Функция вывода на экран должна вырабатывать состояние полей объекта.

4. Создать класс TMoney для работы с денежными суммами. Число должно быть представлено двумя полями типа unsigned int для рублей и типа unsigned char — для копеек. Дробная часть (копейки) должна быть отделена от целой части запятой. Реализовать сложение, вычитание, деление сумм, деление суммы на дробное число, умножение на дробное число и операции сравнения.

5. Создать класс TTriangle для представления треугольника. Поля данных должны включать углы и стороны. Требуется реализовать операции получения и изменения полей данных, вычисления площади, вычисление периметра, вычисления высот, а также определения вида треугольника (равносторонний, равнобедренный или прямоугольный).

6. Создать класс Tangle для работы с углами на плоскости, задаваемыми величиной в градусах и минутах. Обязательно должны быть реализованы: перевод в радианы, приведение к диапазону 0–360, увеличение и уменьшение угла на заданную величину, получение синуса, сравнение углов.

7. Создать класс TPoint для работы с точками на плоскости. Координаты точки — декартовы. Обязательно должны быть реализованы: перемещение точки по оси X, перемещение по оси Y, определение расстояния до начала координат, расстояния между двумя точками, преобразование в полярные координаты, сравнение на совпадение и несовпадение.

8. Рациональная (несократимая) дробь представляется парой целых чисел (a, b), где a — числитель, b — знаменатель. Создать класс TRational для работы с рациональными дробями. Обязательно должны быть реализованы операции:

— сложения

$$(a, b)+(c, d)=(ad+bc, bd);$$

— вычитания

$$(a, b) – (c, d)=(ad – bc, bd);$$

— умножение

$$(a, b)\times(c, d)=(ac, bd);$$

— деление

$$(a, b)/(c, d)=(ad, bc);$$

— сравнение (равно, больше, меньше).

Должна быть реализована приватная функция сокращения дроби, которая обязательно вызывается при выполнении арифметических операций.

9. Создать класс TDate для работы с датами в формате «год, месяц, день». Дата представляется структурой с тремя полями типа unsigned int: для года, месяца и дня. Класс должен включать не менее трех функций инициализации: числами, строкой вида «год, месяц, день» (например, «2018.03.31») и датой. Обязательными операциями являются: вычисление даты через заданное количество дней, вычитание заданного количества дней из даты, определение високосности года, присвоение и получение отдельных частей (год, месяц, день), сравнение дат (равно, до, после), вычисление количества дней между датами.

10. Создать класс TTime для работы со временем в формате «час:минута:секунда». Класс должен включать в себя не менее четырех функций инициализации числами, строкой (например, «23:39:42»), секундами и временем. Обязательными операциями являются: вычисление разницы между двумя моментами времени в секундах, сложение времени и заданного количества секунд, вычитание из времени заданного количества секунд, сравнение моментов времени, перевод в секунды, перевод в минуты с округлением до целой минуты.

11. Реализовать класс TFazzyNumber для работы с нечеткими числами, которые представляются тройками чисел $(x - e_1, x, x+e_2)$. Для чисел $A=(A - a_l, A, A+a_r)$ и $B=(B - b_l, B, B+b_r)$ арифметические операции выполняются по формулам:

— сложение

$$A+B=(A+B - \text{al} - bl, A+B, A+B+ar+br);$$

— вычитание

$$A - B=(A - B - al - bl, A - B, A - B+ar+br);$$

— умножение

$$A \times B=(A \times B - B \times al - A \times bl+al \times bl, A \times B, A \times B+B \times ar+A \times br+ar \times br);$$

— обратное число

$$A=(1/(A - al), 1/A, 1/(A - ar)), A>0;$$

— деление

$$A/B=((A - al)/(B+br), A/B, (A+ar)/(B+bl)), B>0.$$

12. Реализовать класс TAccount, представляющий собой банковский счет. В классе должны быть четыре поля: фамилия владельца, номер счета, процент начисления и сумма в рублях. Открытие нового счета выполняется операцией инициализации. Необходимо выполнять следующие операции: сменить владельца счета, снять некоторую сумму денег со счета, положить деньги на счет, начислить проценты, перевести сумму в доллары, перевести сумму в евро, получить сумму прописью (преобразовать в числительное).

13. Номиналы российских рублей могут принимать значения 1, 2, 5, 10, 50, 100, 200, 500, 1000, 2000, 5000. Копейки представить, как 0.01 (1 копейка), 0.05 (5 копеек), 0.1 (10 копеек), 0.5 (50 копеек). Создать класс TMoney для работы с денежными суммами. Сумма должна быть представлена полями-номиналами, значениями которых должно быть количество купюр данного достоинства. Реализовать сложение сумм, вычитание сумм, деление сумм, деление суммы на дробное число, умножение на дробное число и операцию сравнения. Дробная часть (копейки) при выводе на экран должна быть отделена от целой части запятой.

14. Реализовать класс TBankomat, моделирующий работу банкомата. В классе должны содержаться поля для хранения идентификационного номера банкомата, информации о текущей сумме денег, оставшейся в банкомате, минимальной и максимальной суммах, которые разрешается клиенту снять в один день. Сумма денег представляется полями-номиналами 10–1000. Реализовать метод инициализации банкомата, метод загрузки купюр в банкомат и метод снятия определенной суммы денег. Метод снятия денег должен выполнять проверку на корректность снимаемой суммы: она не должна быть меньше минимального значения и не должна превышать максимальное значение. Метод ToString() должен преобразовать в строку сумму денег, оставшуюся в банкомате.

15. Создать класс TFraction для работы с дробными числами. Число должно быть представлено двумя полями: целая часть — длинное целое со знаком, дробная часть — беззнаковое короткое целое. Реализовать арифметические операции сложения, вычитания, умножения и операции сравнения.

16. Создать класс TGoods (товар). В классе должны быть представлены поля: наименование товара, дата оформления, цена товара, количество единиц товара, номер накладной, по которой товар поступил на склад. Реализовать методы изменения цены товара, изменения количества товара (увеличения и уменьшения). Метод ToString() должен выдавать в виде строки стоимости товара.

17. Создать класс TBitString для работы с 64-битовыми строками. Битовая строка должна быть представлена двумя полями типа unsigned long. Должны быть реализованы все традиционные операции для работы с битами: and, or, xor, not. Реализовать сдвиг влево ShiftLeft и сдвиг вправо ShiftRight на заданное количество битов.

18. Создать класс TLongLong для работы с целыми числами из 64 бит. Число должно быть представлено двумя полями: long — старшая часть, unsigned long — младшая часть. Должны быть реализованы арифметические операции, присутствующие в C++, и равнения.

19. Создать класс TPaynent (зарплата). В классе должны быть представлены поля: фамилия — имя — отчество, оклад, год поступления на работу, процент надбавки, подоходный налог, количество отработанных дней в месяце, начисления и удержания суммы. Реализовать методы вычисления начисленной суммы, вычисления удержанной суммы, вычисления выдаваемой на руки суммы, вычисление стажа. Стаж вычисляется как полное количество лет, прошедших от года поступления на работу, до текущего года. Начисления представляют собой сумму, начисленную за отработанные дни, и надбавки, т. е. доли от первой суммы. Удержания представляют собой отчисления в пенсионный фонд (1% от начисленной суммы) и подоходный налог. Подоходный налог составляет 13% от начисленной суммы без отчислений в пенсионный фонд.

20. Реализовать класс TCursor. Полями являются координаты курсора по горизонтали и вертикали — целые положительные числа, вид курсора — горизонтальный и вертикальный, размер курсора — целое от 1 до 15. Реализовать методы изменения координат курсора, изменения вида курсора, изменения размера курсора, метод гашения и восстановления курсора.

12. БИБЛИОТЕКА ВИЗУАЛЬНЫХ КОМПОНЕНТОВ

Программист — это не тот, кто пишет программы, а тот, чьи программы работают.

С сайта:
http://www.aforizmov.net/tema/tags/programmist/page/5/

Эта глава посвящена основе построения приложений в C++Builder — Библиотеке визуальных компонентов (VCL — Visual Component Library). Библиотека VCL интегрирована в среду C++Builder, что, в отличие от других систем программирования, позволяет манипулировать классами визуальных компонент при проектировании приложения, на стадии создания его прототипа. Поведение и вид компонент определяются по мере разработки приложения, хотя можно модифицировать их и в процессе выполнения программы.

Рассматриваются принципы построения компонентов как потомков базовых классов, изучаются свойства и поведение базовых классов. Проследив иерархию наследования, мы всегда можем составить представление о свойствах интересующего нас компонента. Изучение конкретных компонентов, таких как, например, компонент TButton или компонент TLabel предполагается самостоятельно на базе рассмотренной в этой главе методики.

12.1. Метафора библиотеки визуальных компонентов

12.1.1. Назначение библиотеки визуальных компонентов

Библиотека визуальных компонентов позволяет программистам визуально создавать программные приложения, не прибегая более в целом ряде случаев к кодированию классов «вручную». Подавляющее большинство приложений разрабатываются визуально с помощью Редактора форм C++Builder, добавляя лишь несколько строк к обработчикам ключевых событий компонента.

Прикладной программист создает законченное приложение посредством интерактивного взаимодействия с интегрированной визуальной средой C++Builder, используя компоненты VCL для создания интерфейса программы с пользователем и с другими управляющими элементами: обслуживания баз данных, контролируемого ввода параметров и т. д. Характерная для C++Builder методика визуального стиля разработки программного обеспечения постепенно осваивается другими системами программирования.

Программисты должны знать свойства, методы и события, присущие используемым компонентам. Более того, понимание архитектуры VCL позволяет совершенствовать программу в тех местах, где ощущается необходимость развития существующих или создания новых компонент. Прежде чем изобретать новый элемент, нужно удостовериться, не создал ли уже кто-то компонент с нужными характеристиками.

Системные программисты развивают существующую Библиотеку — либо добавляя в нее новые элементы, либо расширяя функциональность уже имеющихся компонент. Разработчики компонент должны иметь более глубокие знания о внутреннем устройстве VCL, нежели прикладные программисты. Нуж-

но четко представлять себе, какой прием быстрее приведет к поставленной цели: развитие имеющегося или написание нового компонента. Написание компонент представляет собой более традиционную задачу программирования и сопряжено с большими условностями, нежели визуальное создание приложений.

Варианты C++Builder Professional и C++Builder Client/Server Suite поставляются вместе с исходными текстами VCL. Наличие исходных текстов облегчает задачу программистов, которые занимаются разработкой новых компонент и расширением функциональных возможностей уже имеющихся компонент Библиотеки.

Для создания новых компонентов можно с одинаковым успехом пользоваться средствами C++Builder или Delphi, однако если разработанные компоненты предлагаются для внешнего применения, автор обязан удостовериться, что они работают в рамках обеих систем.

Компоненты — это строительные кирпичи, из которых конструируется интерфейс программы с пользователем, с помощью которых «здание» программы приобретает новый внешний облик и скрытые особенности. Для прикладного программиста любой компонент VCL представляет собой объект, который можно «перетащить» из вкладок Палитры компонент на форму создаваемого приложения. Поместив компонент на форму, можно манипулировать его свойствами (посредством Редактора форм) и кодом (с помощью Редактора кода), придавая компоненту специфическое поведение.

Для разработчика компоненты представляют собой объекты на C++ или на Объектном Паскале. Некоторые компоненты инкапсулируют поведение типовых элементов управления, предоставляемых операционными системами Windows. Другие компоненты вводят совершенно новые видимые и невидимые элементы, программный код которых полностью определяет их поведение.

Сложность компонент различна. Так, компонент TLabel из вкладки **Standard** Палитры компонент способен лишь отображать статический текст. Можно сконструировать значительно более сложный компонент, который, например, инкапсулирует законченное обслуживание бухгалтерских документов специализированной базы данных.

Уточним понятие «компонент».

Формально: компонент — это любой класс, порожденный от класса **TComponent**. Верно, но пока мало о чем говорит.

Интуитивно: компонент — это объект того класса, который содержится в Палитре компонентов. Это является достаточным, но не необходимым условием — есть компоненты, которые там не содержатся. С другой стороны, разработчики RAD Studio поместили в Палитру компонентов именно то, что считают оптимальным набором «строительных кирпичиков» для создания любых приложений. Архитектура RAD Studio является наращиваемой — мы можем создавать и включать в Палитру свои компоненты.

Содержательно: компонент — это класс, имеющий опубликованные свойства, с помощью которых взаимодействует со средой RAD Studio во время разработки (значения опубликованных свойств можно изменять как во время проектирования, так и во время исполнения программы). И это определение

нельзя назвать исчерпывающим — могут быть опубликованные свойства и у некоторых классов. Правда, такие классы обычно предназначены для использования в составе компонентов в качестве их свойств.

Дополнительно: компонент — это класс, имеющий в качестве опубликованного свойства свое имя. Под этим именем он и появляется в Палитре компонентов.

Таким образом, компонентом будем считать повторно используемый блок приложений RAD Studio. С точки зрения прикладного программиста компонентный объект представляет собой законченную конструкцию, содержащую свойства, методы и события. В отличие от разработчика компонент, пользователю программы безразлично, от какого класса произведена каждая компонента.

С методической точки зрения удобно ввести понятия «компонент-класс» и «компонент-объект». Во время проектирования, т. е. при создании и редактировании формы, компонент-класс выбирается из палитры компонентов, и экземпляр компонента-класса (т. е. компонент-объект) помещается на форму. Обратим внимание на то, что из палитры компонентов выбирается класс, а во время переноса класса на форму компонент-класс превращается в компонент-объект с автоматически сгенерированным именем. Например, при многократном переносе на форму компонента-класса **TButton** (кнопка) создаются компоненты-объекты с именами **Button1**, **Button2**, **Button3** и т. д. Одновременно каждый компонент-объект помещается в Инспектор объектов. От обычных объектов объект-компонент отличается тем, что он существует как на стадии проектирования программы, так и во время ее исполнения. Обычно в литературе не акцентируется внимание на рассмотренных особенностях уточнения понятия «компонент». Например, даже исходное понятие «библиотека визуальных компонентов» обозначает одновременно и совокупность компонентов-классов, и совокупность компонентов-объектов.

Подобным образом мы можем разместить на форме несколько однотипных компонентов, можем также размещать компоненты внутри других компонентов. Например, на форму можно поместить экземпляр компонента класса **TPanel** (панель), а на поверхность экземпляра **TPanel** — поместить экземпляр компонента **TEdit**, отвечающий за редактирование данных. При этом и форма, и панель должны запомнить наши действия и при очередном сеансе проектирования должны воспроизвести наши действия.

RAD Studio является средой программирования Windows. Многие компоненты RAD Studio инкапсулируют стандартные элементы управления и функции Windows. Обычно можно положиться на компоненты RAD Studio и не обращаться к Windows API (Application Programming Interface), но бывают случаи, когда приходится прибегать непосредственно и к API. В таких случаях необходимо просто написать вызов нужной подпрограммы. Достоинства системы программирования RAD Studio можно по-настоящему оценить, изучив библиотеку визуальных компонентов VCL (Visual Component Library). Библиотека визуальных компонентов написана на языке C++ и является великолепным примером использования технологии объектно-ориентированной разработки программ.

В последующих параграфах главы рассмотрены важнейшие свойства и методы базовых классов, лежащих в основе VCL. Основное внимание уделяется принципам построения VCL, которые базируются на основополагающих понятиях объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм. Рассматривается иерархия базовых классов и их возможности. При этом основное внимание уделяется формированию свойств и методов классов, начиная с базового класса TObject и переходя к его ближайшим потомкам. Представленные здесь сведения помогут разобраться с заложенными в библиотеку визуальных компонентов механизмами функционирования рассматриваемых компонентов. Настоящая глава не содержит исчерпывающие справочные материалы о компонентах, однако при этом, приступая к освоению не рассмотренного здесь компонента (например, **TButton** или **TMemo,** или **TPrinter** и т. д.), рекомендуется в первую очередь познакомиться с его «родословной». Получив информацию об унаследованных методах и свойствах, можно приступать к освоению собственных методов и свойств компонента. Данную процедуру можно рассматривать как методический прием при изучении компонентов библиотеки.

Контрольные вопросы

1. Какие задачи решает прикладной программист, используя VCL?
2. Какие задачи решает системный программист, используя VCL?
3. Охарактеризуйте понятие «Компонент».

12.1.2. Типы компонентов

Незадачливые программисты принимают как факт, что каждый компонент имеет свойства **Top** и **Left**, которые определяют положение компонента на форме-владельце; для них не существенно, что эти свойства унаследованы от общего предшественника **TComponent**. Напротив, когда мы создаем компонент, то обязаны знать, от какого родителя можно заимствовать нужные свойства, а также все остальные его характеристики так, чтобы их можно было наследовать, а не создавать вновь.

Понимание VCL основано на трех фундаментальных принципах. Во-первых, нам придется ознакомиться со специальными характеристиками четырех базисных типов компонент: стандартного управления, оригинального управления (custom control), графического управления и невидимых компонентов. Во-вторых, мы должны понимать структуру Библиотеки, в которую встроены описания и реализации кодов компонентов. В-третьих, мы должны знать положение упомянутых четырех типов компонентов в иерархии VCL.

12.1.2.1. Стандартные компоненты

Некоторые компоненты VCL, такие как **TButton** (кнопка), **TListbox** (список) и **TEdit** (поле редактирования текста), инкапсулируют поведение типовых элементов управления операционной системы Windows. Мы найдем стандартные компоненты на вкладках **Standard и Win95** Палитры компонент.

Любой стандартный компонент выглядит и ведет себя точно так же, как и инкапсулированный им элемент управления Windows. VCL добавляет обрамление, которое никак не меняет свойств исходного элемента управления, а лишь

делает доступной модификацию вида и поведения компонента посредством свойств и методов.

Если мы собираемся использовать стандартные компоненты без изменений, то не нужно вникать в правила построения обрамлений VCL. Разработчик компонентов может открыть файл исходных текстов стандартных компонентов, входящий в поставку версий C++Builder Professional или C++Builder Client/Server Suite, чтобы понять, каким образом известные элементы управления Windows обрамляются при включении в Библиотеку.

Например, компонент **TListBox** отображает элементы списка в один столбец, хотя инкапсулирует класс простого списка **LISTBOX** из Windows, который может отображать список в несколько столбцов. Чтобы изменить поведение компонента, нам придется реализовать переопределение метода создания данного компонента, принятого по умолчанию.

12.1.2.2. Оригинальные компоненты

В отличие от стандартных компонентов, оригинальные компоненты представляют собой элементы управления, у которых нет ни метода для собственного отображения, ни заранее определенного поведения. Разработчик компонентов должен предусмотреть код, реализующий рисование самого компонента, и код, определяющий поведение компонента, когда пользователь взаимодействует с ним. Примерами оригинальных компонентов являются **TPanel** (панель) и **TStringGrid** (массив строк).

Следует отметить, что как стандартные, так и оригинальные компоненты всегда ассоциируются с некоторым окном управления, поэтому иногда называются оконными (windowed components). Данный аспект подробно обсуждается в параграфе, описывающем класс **TWinControl**. Оконные компоненты обладают следующими свойствами: они могут быть активизированы (принять фокус ввода), используют системные ресурсы и могут служить контейнерами, т. е. являться родителями других элементов управления. Примером контейнерного компонента является **TPanel**.

12.1.2.3. Графические компоненты

Графические компоненты представляют собой видимые элементы управления, которые не могут принять фокус ввода, так как не являются оконными. Графические компоненты обеспечивают отображение объектов без использования системных ресурсов, они требуют меньших «накладных расходов», нежели стандартные или оригинальные компоненты. Примерами графических компонент являются **TImage** (битовая картинка) и **TShape** (индикатор).

Графические компоненты не могут служить контейнерами для других элементов управления, т. е. не могут владеть другими компонентами.

12.1.2.4. Невидимые компоненты

Во время выполнения программы невидимые компоненты не появляются на форме в виде каких-либо элементов управления. Поведение невидимых компонентов определяется на этапе проектирования путем инкапсуляции нужных свойств объекта.

С помощью Инспектора объектов мы можем модифицировать свойства невидимых компонентов и предусматривать код обработчиков событий для них. Примерами таких компонентов являются **TOpenDialog** (диалог открытия файлов), **TTable** (таблица базы данных) или **TTimer** (датчик времени).

12.1.2.5. Контейнерные компоненты

Как уже упоминалось, некоторые компоненты VCL, такие как **TForm**, **TPanel**, могут владеть другими компонентами или являться родителями других компонентов. Указанные аспекты имеют разное смысловое значение, что и проясняется далее.

Право владения. Любой компонент может находиться во *владении* (ownership) других компонентов, но не все компоненты могут являться владельцами. Свойство компонента **Owner** (владелец) содержит ссылку на компонент, который им владеет. Свойство **Owner** устанавливается в момент создания компонента в процессе выполнения приложения. Владелец компонента ответствен за автоматическое создание и освобождение тех компонентов, которыми владеет. Этим и ограничивается связь между владельцем и компонентами, которыми он владеет. По умолчанию владельцем всех компонентов является экранная форма — потомок класса **TForm**.

В процессе конструирования форма автоматически становится владельцем всех компонентов, размещенных на ней, даже если часть их размещена на другом компоненте, таком как **TPanel**. Владение применимо не только к видимым, но и к невидимым (**TTimer**, **TDataSource**) компонентам.

Когда компонент создается динамически в процессе выполнения программы, конструктору компонента передается ее владелец в качестве параметра. В следующем примере неявный владелец формы **(this)** передается конструктору компонента **TButton** как параметр. **TButton** выполнит присваивание значения переданного параметра свойству **Owner** кнопки **MyButton**:

```
MyButton=new TButton(this);
```

где **this** — указатель на форму, в методах которой выполняется данное действие.

Когда форма, владеющая компонентом **TButton**, освобождается, автоматически уничтожается и кнопка **MyButton**.

Мы можем создать компонент, у которого нет владельца, передавая значение параметра 0 конструктору компонента. В этом случае, если этот компонент перестает быть нужным, его уничтожение должно выполняется принудительно (с помощью оператора **delete).** Следующий пример иллюстрирует обращение с компонентом **TTable**, не имеющим владельца:

```
TTable * MyTable=new TTable(0);
// Код, реализующий работу с MyTable
delete MyTable;
```

Свойство **Components** типа массив содержит перечень компонентов, которыми владеет данный компонент. Список владения некоторой формы класса **TMyForm** можно получить в обработчике события OnClick с циклом отображения имен классов всех компонентов, которыми владеет наша форма:

```
void _fastcall TMyForm::ButtonlClick(TObject *Sender)
{
   for (int i=0; i<ComponentCount; i++)
       ShowMessage(Components[i]–>ClassName()) ;
}.
```

Родительское право. Понятие *родительского права* (parentship) существенно отличается от права владения и применимо только к видимым (оконным) компонентам. У каждого компонента есть родитель. Им может быть форма или другой оконный компонент. В процессе выполнения приложения мы можем узнать родителя того или иного компонента по значению свойства **Parent** (Родитель). Это свойство можно читать и изменять только во время выполнения приложения, в Инспекторе объектов оно отсутствует.

Компонент может использовать многие свойства своего родителя (не путать с механизмом наследования классов). Например, для всех визуальных компонентов мы можем увидеть в Инспекторе объектов такие свойства, как **ParentFont** или **ParentShowHint,** или **ParentCtl3D** и др. Эти свойства указывают (если их значения равны **True**), что компонент использует, соответственно, родительские атрибуты шрифта, показа ярлычков, атрибуты оформления. Аналогично известные свойства **Top** и **Left**, определяющие положение левого верхнего угла любого визуального компонента, измеряются в системе координат родительского компонента. Поэтому при перемещении родительского компонента синхронно перемещаются все его дочерние компоненты. Имеются еще два важных свойства, которые связывают компоненты с родителями. Это свойства **Visible** — видимый, и **Enabled** — доступный. Установка в **True** свойства **Visible** сделает видимыми как сам компонент, так и все его дочерние. Аналогично используется свойство **Enabled**. При установке в **False** значения свойства **Enabled** пользователь не сможет нажимать кнопки и производить любые другие действия в пределах данного родительского компонента.

Таким образом, понятие родительского компонента очень важное и его смысл выходит далеко за рамки просто эстетического оформления окна формы и других, схожих по свойствам компонентов.

К оконным компонентам относятся такие видимые элементы, как **TEdit**, **TListBox** и **TMemo**. Чтобы отобразить оконный компонент, ему надо присвоить родителя, ответственного за отображение. Это присваивание выполняется автоматически на стадии проектирования, когда мы перетаскиваем нужный компонент из Палитры компонент на форму. Напротив, при создании компонента во время выполнения программы мы должны явно записать это присваивание, иначе компонент не будет отображен. Например, при создании компонента **TEdit** во время выполнения программы:

```
void _fastcall TForm1::FormCreate(TObject *Sender)
{
   MyEdit=new TEdit(this); // Передать this как владельца

   MyEdit–>Parent=this;    // Передать this как родителя
}
```

автоматически выполнится присваивание **MyEdit–>Owner=this;**, а для корректной работы программы необходимо явным образом выполнить **MyEdit–> Parent=this;**.

В заключение обсуждения темы проведем эксперимент. На рисунке 12.1 приведена экранная форма после выполнения обработчика события ее создания:

```
void __fastcall TForm2::FormCreate(TObject *Sender)
{
    Edit1–>Parent=Panel2;
    Edit1–>Top=20;
    Edit1–>Left=50;
    StringGrid1–>Cells[0][0]=«Компонент»;
    StringGrid1–>Cells[1][0]=«Родитель»;
    for (int i=0; i < this–>ComponentCount; i++)
       {
          StringGrid1–>Cells[0][i+1]=Components[i]–>Name;
         StringGrid1–>Cells[1][i+1]=(Components[i]–>GetParentComponent())–>Name;
       }
   StringGrid2–>Cells[0][0]=«Компонент»;
   StringGrid2–>Cells[1][0]=«Владелец»;
   for (int i=0; i < this–>ComponentCount; i++)
      {
        StringGrid2–>Cells[0][i+1]=Components[i]–>Name;
        StringGrid2–>Cells[1][i+1]=Components[i]–>Owner–>Name;
       }
}.
```

Обратим внимание на следующее обстоятельство. Компонент **Edit1** при попытке размещения на компоненте **Panel2** никак «не хотел» следовать нашему желанию. Результат виден на рисунке 12.2, в котором приведена часть экранной формы проектирования приложения в режиме Design. В левой верхней части экранной формы приведено окно Structure, в котором показаны родительские отношения между компонентами. Как видно из рисунка 12.2, родителем компонента **Edit1** оказался компонент **TPanel1**. Чтобы исправить ситуацию, в текст программы включены следующие действия:

```
Edit1–>Parent=Panel2;
Edit1–>Top=20;
Edit1–>Left=50;
```

по изменению родителя компонента и требующегося при этом пересчета его положения в системе координат нового родителя. Справедливости ради заметим, что поменять родителя можно во время проектирования перетаскиванием компонента в окне структуры в нужное место, с последущей подгонкой его положения.

Для полного понимания результатов нашего эксперимента рекомендуем внимательно сопоставить тексту программы результаты вычислений.

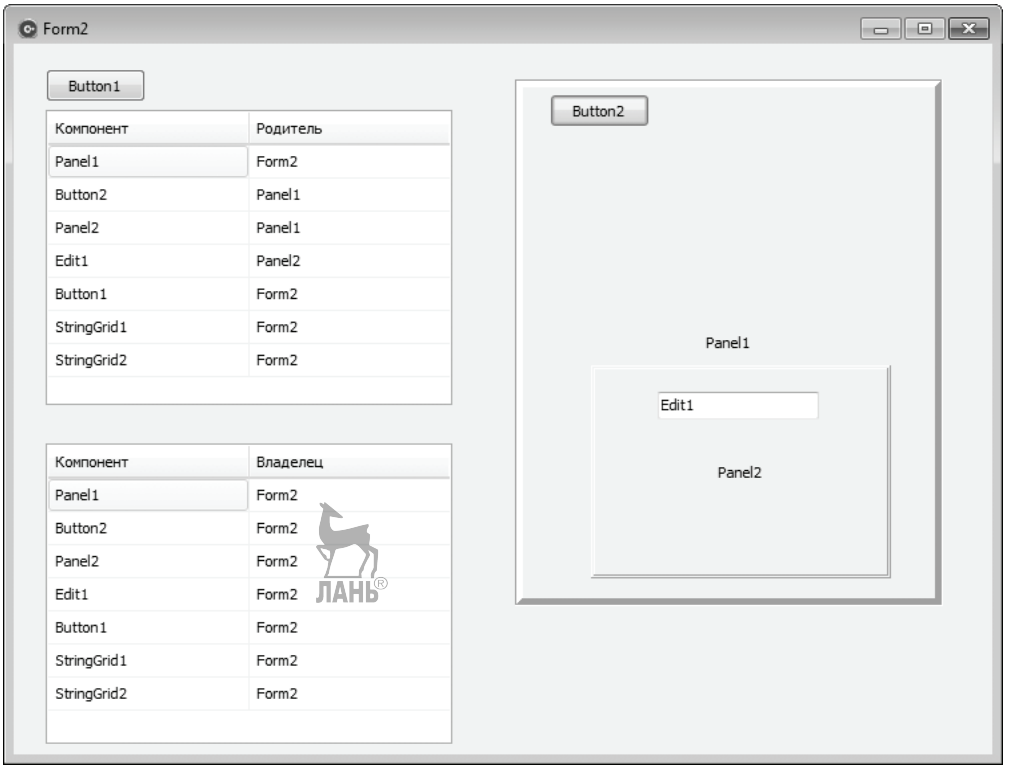


Рис. 12.1

Вид экранной формы изучения родительского права и права владения

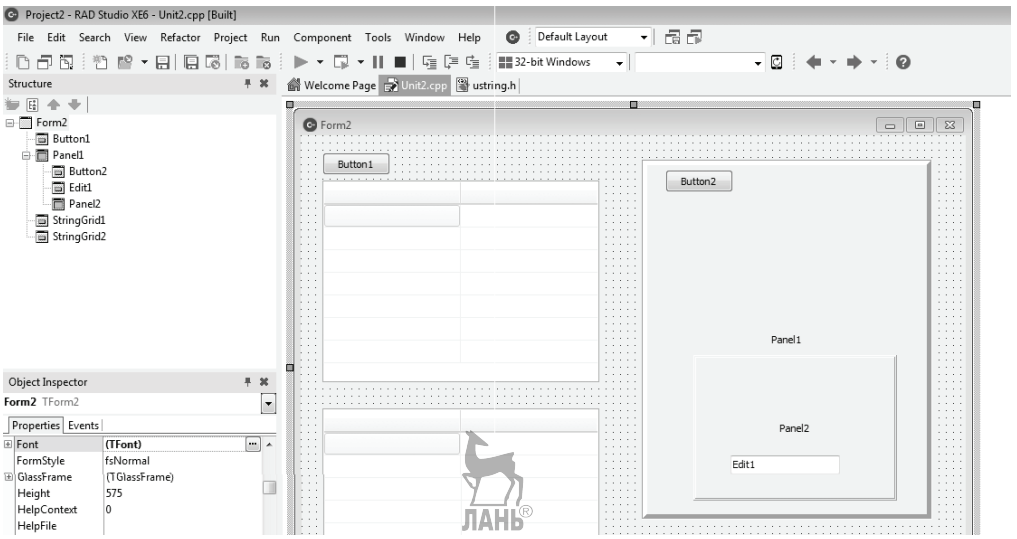


Рис. 12.2

Вид окна структуры компонентов во время проектирования

12.1.3. Свойства компонент

Определение класса в языке C++ содержит инкапсуляцию полей и методов, оперирующих с данными и определяющих поведение объекта. Эта концепция всех систем объектно-ориентированного программирования принята и в VCL.

VCL позволяет манипулировать видом и функциональным поведением компонентов не только с помощью методов (как это делается с обычными классами), но и посредством свойств и событий, присущих только классам компонентов.

Разрешены любые типы свойств, за исключением файлового типа. В отличие от поля, свойство не хранит данные, однако его методы чтения и записи позволяют получить доступ к защищенному полю объекта.

Присваивание значения полю посредством присваивания свойству вызывает «побочный эффект», за которым могут скрываться сложные операции над компонентом. Например, побочный эффект при изменении свойства **Caption** (заголовок) некоторой формы проявляется в немедленном изменении текста заголовка окна этой формы, при отсутствии явного обращения к методу, реализующему операцию изменения заголовка.

Использование свойств в VCL. Наиболее очевидное достоинство свойств проявляется в том, что они представлены в окне Инспектора объектов. Это существенно упрощает работу программиста: вместо того, чтобы реализовывать ввод и обработку параметров при конструировании объектов во время выполнения пограммы, мы всего лишь задаем в Инспекторе объектов значения опубликованных свойств.

С точки зрения прикладного программиста, компоненты выглядят как обычные объектные переменные. Можно считывать или менять значения свойств точно так же, как это делает программа с полями во время выполнения. Единственное, что нельзя делать, это передавать указатели на свойства методам в качестве параметров. С точки зрения разработчика компонентов, в свойствах заключены гораздо большие возможности, нежели в обычных полях. Действительно:

— программисты могут устанавливать значения свойств на стадии проектирования. Это очень важная возможность, поскольку в отличие от методов, проявляющих себя во время работы программы, свойства дают возможность манипулировать поведением компонента до момента запуска приложения. В целом, наши компоненты не должны содержать много методов: большинство из них, вероятно, будут инкапсулированы свойствами;

— как обычно, в отличие от поля, свойство может прятать детали реализации, например, некоторые свойства могут храниться в зашифрованном виде и появляться в нормальном виде только при чтении или установке его значений. Хотя фактическим значением свойства может являться обычное число, компонент может, скажем, выбирать его из базы данных или выполнять сложные вычисления, чтобы выдать это значение;

— применение свойств приводит к побочным эффектам в операторах присваивания полям объекта. Эти внешне простые операторы могут подразу-

мевать вызов метода, который, в принципе, может сделать все, что угодно. Простым примером служит свойство всех компонент под названием **Top**. Присваивание не только меняет значение этого свойства, но и, благодаря побочному эффекту, приводит к перемене положения компонента на форме и перерисовке ее содержания. Отметим, что побочный эффект установки свойств может не ограничиваться данным компонентом. Например, установка значения **True** свойства **Down** в одном из компонентов группы кнопок приводит к изменению соответствующего свойства всех других кнопок этой группы на значение **False**;

— методы реализации свойства могут быть виртуальными, т. е. в разных компонентах одно и то же свойство может быть связано с различными действиями.

Объявление свойств. Описание и использование свойств рассматривались в параграфе 6.8. Однако, только рассматривая использование свойств в составе VCL, мы можем проследить все их возможности. Начнем с полного описания синтаксиса свойства:

__property <тип свойства> <имя свойства>={<список атрибутов>} ;

где <список атрибутов> содержит перечисление следующих атрибутов свойства:

write=< *поле или метод записи* >;
read=< *поле или метод чтения* >;
default=< *булева константа, управляющая сохранением значения*>;
stored=< *булева константа или функция, сохраняющая значение* >.

Информация, необходимая разработчикам новых компонент для реализации оригинальных операций со значениями свойств, требует более детального изложения.

Доступ к внутренним данным свойств. Как известно, C++Builder не вводит ограничений на способ хранения значений свойств. Однако все разработчики компонент VCL придерживаются следующих соглашений:

— значения свойств хранят поля объекта;

— идентификаторы полей, хранящих значения свойств, образуются добавлением префикса F к имени поля. Например, значение свойства **Width** компонента **TControl** хранит поле с именем **FWidth**;

— идентификаторы полей, хранящих значения свойств, должны быть объявлены как **private**; при этом компонент, объявивший эти свойства, имеет к ним доступ, а пользователь данного компонента и его производные — нет;

— производные компоненты должны использовать само наследованное свойство, не пытаясь осуществить прямой доступ к памяти внутренних данных.

Целесообразность изложенных соглашений поддерживает простой принцип объектно-ориентированного программирования: только методы, реализующие свойство, имеют право доступа к своим значениям. Если какому-то другому методу или компоненту понадобилось изменить эти значения, они должны осуществлять это посредством данного свойства, не обращаясь напрямую к его внутренним данным.

Прямой доступ является простейшим способом обращения к значениям свойств. Атрибуты **read** и **write** объявления свойства указывают, что чтение

или присваивание значений внутренним полям свойства происходит непосредственно, без вызова соответствующих методов. Прямой доступ чаще всего используется только для чтения значений свойств. При этом побочный эффект не возникает, однако данное свойство будет представлено в окне Инспектора объектов.

Методы чтения и записи замещают имена полей в атрибутах **read** и **write** объявления свойства. Вне зависимости от конкретной реализации методов, они должны быть объявлены как приватные, а производные компоненты должны использовать наследованные методы для чтения и записи значений свойства. Объявление методов приватными защищает пользователя производного компонента от случайного вызова неадекватных методов, модифицирующих свойства не так, как ожидалось.

Значение свойства по умолчанию представляет собой именно то начальное значение инициализации свойства, которое устанавливает конструктор данного компонента. C++Builder использует объявленное значение свойства по умолчанию **default**, чтобы определить, следует ли сохранять свойство в файле формы с расширением **.dfm** (если атрибут **stored** явно не запрещает это).

Приведем пример объявления компонента с единственным свойством **IsTrue**, имеющим значение по умолчанию **true**, а также конструктор, который устанавливает это значение при инициализации компонентного объекта. Заметим, что если свойство имеет значение по умолчанию **False**, то не нужно явно устанавливать его в конструкторе, поскольку все объекты (включая компоненты) всегда инициализируют поля значением 0, т. е. **False**:

```
class TMyComponent : public TComponent
{
    private:
        Boolean FIsTrue;
    public:
         virtual _fastcall TMyComponent(TComponent* AOwner);
    _published:
        _property Boolean IsTrue={ read=FIsTrue,write=FIsTrue, default=True };
};

_fastcall TMyComponent::TMyComponent (TComponent* AOwner) : TComponent
(AOwner)
{
        FIsTrue=True;
}.
```

Таким образом, рассмотренный пример показывает, как осуществляется установка конструктором значения свойства по умолчанию.

Свойства обеспечивают доступ к полям. Существует два способа, посредством которых свойства обеспечивают доступ к внутренним полям компонент: прямой или косвенный, посредством методов чтения/записи. Например, рассмотрим фрагмент кода объявления компонента **TCustomEdit**, взятый из VCL. **TCustomEdit —** это базовый класс для таких компонент редактирования, как **TEdit** и **TMemo.**

```
class DELPHICLASS TCustomEdit;
class PASCALIMPLEMENTATION TCustomEdit : public Vcl::Controls::TWinControl
{
    private:
        int FMaxLength;
        void __fastcall SetMaxLength(int Value);
     protected:
        __property int MaxLength={read=FMaxLength, write=SetMaxLength, default=0};
         };.
```

В рассмотренном примере **TCustomEdit** имеет внутреннее поле **FMaxLength** типа **int,** которое определяет максимальное значение длины символьной строки, которую пользователь может ввести в данный элемент управления. Вместо прямого присваивания **FMaxLength** этого значения выполняется присваивание свойству **MaxLength**, которое обеспечивает доступ к полю **FMaxLength**.

Определение свойства **MaxLength** содержит тип, имя свойства и объявления атрибутов: прямого чтения поля **FMaxLength**, косвенного присваивания свойству **MaxLength** посредством метода записи **SetMaxLength** и значения по умолчанию 0.

Методы записи и чтения свойств. *Метод записи* имеет единственный параметр того же типа, что и свойство, и не возвращает значения. По соглашению, название функции образуется из слова «Set», за которым следует имя свойства. Например, метод записи свойства **MaxLength** имеет имя **SetMaxLength** и тип **void.** Метод записи свойства присвоит значение своего параметра соответствующему полю. Одна из причин использования метода записи — вызвать побочный эффект как результат операции присваивания свойству.

Рассмотрим пример реализации ранее объявленного метода записи **SetMaxLength**.

```
void TCustomEdit::SetMaxLength(int Value)
{
      if (FMaxLength != Value)
      {
          FMaxLength=Value;
          if (HandleAllocated) SendMessage(Handle, EM_LIMITTEXT, Value, 0);
       }
}.
```

Метод **SetMaxLength** проверяет, не присваивается ли свойству то значение, которое уже в нем хранится. Если нет, то новое значение присваивается полю **FMaxLength**. Кроме того, метод выдает сообщение **EM_LIMITTEXT** Windows в то окно, которое инкапсулирует компонент **TCustomEdit**.

По этому сообщению устанавливается верхний предел длины текста, который пользователь может ввести в данный элемент управления. Последнее действие представляет собой простейший вариант побочного эффекта, вызываемого методом записи свойства.

Метод чтения представляет собой функцию без параметров (или с параметрами индексов для свойства типа массив), которая возвращает типизированное значение свойства. По соглашению, название функции образуется из слова «Get», за которым следует имя свойства. Например, метод чтения свойства **MaxLength** может иметь имя **GetMaxLength** и возвращать значение типа **int.**

Метод чтения может осуществлять преобразование типа поля.

Переопределение свойств. Все компоненты наследуют свойства своих предшественников, причем абстрактные базовые классы обычно объявляют свои свойства преимущественно в секциях **public** или **protected.** Чтобы такие свойства стали доступными пользователям производных компонент (как на стадии проектирования, так и во время выполнения программы), они обязаны переопределить их с ключевым словом **_published.**

Как видно из предыдущего примера, свойство **MaxLength** было определено в секции **protected** базового компонента **TCustomEdit**. Следующий пример содержит переопределение свойства **MaxLength** как **_published** в производном компоненте **TEdit**.

```
class TEdit : public TCustomEdit
{
    __published:
        __property int MaxLength={ nodefault } ;
        // Другие объявления
};.
```

Такое переопределение только снимает ограничения свойства, т. е. мы можем переопределить **protected-**свойство как **public**-свойство. При переопределении свойства достаточно указать его имя, однако мы можем изменить значения атрибутов **stored** и **default**. Отметим также, что свойство **MaxLength** теперь вообще не имеет значения по умолчанию, хотя в унаследованном свойстве оно было задано.

Другая принципиальная особенность свойств заключается в том, что их можно изменять во время разработки программы с помощью Инспектора объектов.

Правила видимости, объявленные ключевыми словами **private, protected и public,** действуют на свойства так же, как и на обычные поля и методы. Единственное отличие объявлений, сделанных в секции **_published,** от объявлений в секции **public,** проявляется в том, что во время разработки программы Инспектору объектов передается информация RTTI (Run-Time Type Information) о типах полей и свойств.

Типы свойств. Свойство может быть любого типа, который способна возвратить функция (так как реализация свойства может возлагаться на функцию). Разные типы свойств по-разному представлены в окне Инспектора объектов и определяют разные варианты их редактирования, предлагаемые Инспектором. Более того, некоторые свойства имеют собственные редакторы.

Правилами языка C++ устанавливаются следующие обобщенные группы типов компонентных свойств:

Тип свойства	Действия Инспектора объектов
Simple	Простые числовые, символьные и строчные свойства показываются Инспектором в виде чисел, символов или символьных строк, соответственно. Можно непосредственно вводить и редактировать значения простых свойств
Enumerated	Свойства перечисляемого типа (в том числе булевы) показываются Инспектором в виде значений, определенных в исходном тексте программы. Можно выбирать возможные значения из выпадающего списка перечислений
Set	Свойства типа множества показываются Инспектором в виде элементов множества. При расширении множества следует обращаться с каждым его элементом как с булевым значением: true, если элемент принадлежит множеству, или false в противном случае
Object	Свойства, которые сами по себе являются объектами, обычно обслуживаются своими собственными редакторами. Инспектор позволяет индивидуально редактировать те объектные свойства, которые объявлены как published. Объектные свойства должны быть производными от компонента **TPersistent**[42]
Array	Свойства типа массив должны обслуживаться своими собственными редакторами свойств. Инспектор не имеет встроенных средств для редактирования таких свойств

Свойства типа множество. Как известно, C++Builder объявляет несколько шаблонных классов для встроенных типов Delphi, которых нет в языке C++.

В частности, типы стиля шрифта определяются следующим образом:

```
enum TFontStyle { fsBold, fsItalic, fsUnderline, fsStrikeOut } ;
typedef Set <TFontStyle, fsBold, fsStrikeOut>TFontStyles ;
```

TFontStyle является перечисляемым типом и определен как множество — неупорядоченная коллекция типа **TFontStyle**. Инспектор объектов позволяет переключать булевы значения элементов множества, выбирая значения **false** или **true**.

Свойства типа массив. Эти свойства имеют множественные элементы, каждый из которых соответствует некоторому индексному значению. Например, свойство **Lines** стандартного компонента **TMemo** представляет собой индексируемый список (массив) текстовых строк, составляющих многострочное поле редактирования. В данном случае свойство **Lines** предоставляет пользователю естественный доступ к указанному элементу массива (строке) в большом тексте.

Следующий пример содержит объявление свойства **String** и метода чтения **GetNumberSize**, возвращающего индексируемую строку массива:

```
class TDemoComponent : public TComponent
{
    private:
        String _fastcall GetNumberSize(int Index);
    public:
        __property String Number[int Index]={ read=GetNumber };
```

[42] Компонент **TPersistent** рассмотрен далее в параграфе 12.3.

```
        // Другие объявления
};
String _fastcall TDemoComponent::GetNumberSize(int Index)
 {
     String Result;
     switch (Index)
     {
         case 0: Result=«Zero»; break;
         case 100: Result=«Medium»; break;
         case 1000: Result=«Large»; break;
         default: Result=«Unknow size»;
     }
     return Result;
}.
```

Объявление свойств типа массив имеет следующие особенности:

— объявление свойства включает один или несколько индексов любого типа (по числу размерностей массива). В этом состоит первое отличие от обычных массивов, где индекс всегда имеет тип **int**;

— доступ к элементам массива реализуется посредством методов чтения/записи, которые имеют дополнительные параметры-индексы массива, перечисляемые в том же порядке, что и при объявлении методов. В этом состоит второе отличие от обычных массивов, где допускаются ссылки не только на отдельные элементы, но и на весь массив.

Контрольные вопросы

1. Какие компоненты считаются стандартными?
2. Какие компоненты считаются оригинальными?
3. Какие компоненты считаются графическими?
4. Какие компоненты считаются стандартными?
5. Какие компоненты считаются контейнерными?
6. Какие свойства придает «право владения» контейнерным компонентам?
7. Какие свойства придает «родительское право» контейнерным компонентам?
8. Как объявление свойств влияет на поведение компонента?

12.1.4. События компонентов

События представляют собой средства, с помощью которых компонент сообщает пользователю о том, что на него оказано некоторое предопределенное воздействие. Типичные простые события — нажатие кнопки на форме или клавиши на клавиатуре. Вкладка «События» (Events) Инспектора объектов позволяет получить доступ к событиям выбранного компонента.

Назначение событий. В общих словах событие определяется как механизм связи происшествия с некоторым кодом. Более точно событие — это **closure**-указатель на специфический метод в специфическом экземпляре класса.

С точки зрения прикладного программиста, событие представляет собой переменную, которой можно присвоить некоторый вызываемый метод. Например, экземпляр компонента **TButton** — кнопка Button1 имеет переменную для события **OnClick**. По умолчанию C++Builder генерирует обработчик события с именем **ButtonlClick** и присваивает его событию **OnClick**. Когда происходит нажатие кнопки, объект вызывает метод, присвоенный событию **OnClick**, в данном случае **ButtonlClick**.

Таким образом, прикладной программист воспринимает событие как способ указания, к какому из написанных им методов должна обращаться программа, когда данное событие происходит. С точки зрения разработчика компонент, в понятии события заключено значительно большее. Автор компонента обязан предусмотреть программный интерфейс, к которому можно подключить методы, вызываемые при определенных воздействиях на компонент. Наш компонент предоставляет «разъемы», в которые мы сможем вставлять специфические коды реакции на события.

Определение событий компонентов. Формально C++Builder определяет событие как типизированный указатель на метод в специфическом экземпляре класса:

<тип> **(__closure *** <имя метода>**) (**<список параметров>**)**.

Для разработчика компонент **__closure** представляет собой некоторую программную заглушку: когда пользователь определяет реакцию на некоторое событие, место заглушки занимает его обработчик, который вызывается нашей программой при возникновении этого события.

Когда пользователь выполняет присваивание некоторому событию, происходит присваивание не просто метода с конкретным именем, а метода в специфическом экземпляре класса. В качестве экземпляра класса, указатель которого **this** передается как скрытый параметр, обычно (но не всегда) выступает форма, которая содержит данный компонент.

Например, мы хотим привести все вводимые символы к верхнему регистру. Для этого надо определить следующий обработчик события нажатия клавиши:

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, System::WideChar &Key)
{
Key=UpCase(Key);
}.
```

Передача адресных аргументов может также использоваться для переопределения поведения обработчика события по умолчанию.

Присваивание обработчиков всем возможным событиям нашего компонента вовсе необязательно. Этот принцип оказывает существенное влияние на разработку наших компонент и их событий. Очевидно, работа нашего компонента не должна нарушаться из-за того, что пользователь просто не предусмотрел обработчик какого-то события.

При разработке компонент нужно учитывать следующие аспекты обработчиков событий:

— прикладные программисты не обязаны обрабатывать все события. Различные события возникают практически постоянно при работе любого приложения Windows. Простое смещение курсора по компоненту вызывает передачу многочисленных сообщений Windows данному компоненту о передвижении мыши, которые компонент транслирует в события **OnMouseMove**. Если поведение компонента не зависит от манипуляций мышью, то в большинстве случаев программа просто не реагирует на такие события;

— прикладные программисты могут написать любой код обработки события. Компоненты VCL реализуют свои события так, чтобы свести к минимуму риск неверной реакции вследствие логических ошибок в обработчике события. Конечно, невозможно защититься от всех ошибок, однако можно, например, перед вызовом обработчика выполнить инициализацию всех структур данных, чтобы пользователи не получали неопределенной информации.

Стандартные события компонентов. Все компоненты VCL наследуют большинство общих сообщений Windows на стандартные события. Такие события встроены в защищенные секции компонентов, поэтому пользователи не могут подсоединять к ним свои обработчики.

Существует две категории стандартных событий: определенные для всех компонент и определенные только для оконных компонент (стандартных и оригинальных). Все компоненты наследуют от базового абстрактного класса **TControl** следующие стандартные события[43]:

OnClick	**OnDragDrop**	**OnEndDrag**	**OnMouseMove**
OnDblClick	**OnDragOver**	**OnMouseDown**	**OnMouseUp**

В дополнение к этим событиям оконные компоненты наследуют от базового абстрактного класса **TWinControl** еще и следующие стандартные события[44]:

OnEnter	**OnKeyDown**	**OnKeyPress**
OnKeyUp	**OnExit**	

Чтобы стандартные события нашего компонента были видимы Инспектору объектов на стадии проектирования или во время выполнения программы, мы должны переопределить свойства событии в секции **__published.** Следующий пример показывает, как сделать стандартное событие **OnClick** видимым:

```
class TMyControl : public TCustomControl
{
   _published:
         _property OnClick; // OnClick стало видимым Инспектору
};.
```

Все стандартные события имеют соответствующие защищенные динамические методы, унаследованные от **TControl**, имена которых образованы от названия события без частицы «On». Например, события **OnClick** вызывают метод **Click**.

[43] Подробная информация о событиях класса **TControl** приведена в п. 12.7.2.

[44] Подробная информация о событиях класса **TWinControl** приведена в п. 12.8.2.

Как правило, мы сначала обращаемся к наследованному методу, разрешая пользовательскому обработчику события произвести свои действия перед тем, как сработает код нашего переопределения. Предположим, мы пишем новый компонент, в котором хотим модифицировать реакцию на щелчки мышью. Вместо того, чтобы присвоить соответствующий обработчик события **OnClick**, как это сделал бы прикладной программист, мы переопределяем защищенный метод Click:

```
void _fastcall TMyControl::Click()
{
    …
    // Стандартное обслуживание, включающее вызов обработчика
    TWinControl::Click();
   // Необходимо также привести наш код переопределения метода
}.
```

Собственные события компонентов. Необходимость в определении совершенно новых событий возникает довольно редко. Скорее всего, окажется достаточным переопределить обработку уже существующих событий. Если мы все же придумали компонент с совершенно оригинальным поведением, то придется ввести и новые события.

Существует две причины возникновения событий: воздействие пользователя на компонент и изменение его состояния. События в результате воздействия пользователя почти всегда вызываются сообщениями Windows, указывающими, что на это воздействие может потребоваться некоторая реакция. События в результате изменения состояния компонента могут также соотноситься с сообщениями Windows (например, изменение фокуса), хотя чаще они являются следствием изменений свойств компонента. Разработчик компонентов должен полностью контролировать причины возникновения собственных событий, чтобы пользователи знали, как их использовать.

Определение события проходит через четыре этапа, для реализации которых нам необходимо следующее.

1. Знать, какое действие вызывает событие. Для некоторых событий ответ очевиден. Например, при нажатии левой кнопки мыши Windows посылает сообщение **WM_LBUTTONDOWN**. Принимая это сообщение, компонент вызывает метод **MouseDown**, который, в свою очередь, обращается к обработчику события, который пользователь подсоединил к событию **OnMouseDown**. Некоторые события менее очевидно связаны с внешними воздействиями. Так, линейка прокрутки имеет событие **OnChange**, которое возникает по разным причинам — в результате щелчков мышью, нажатия управляющих клавиш или изменений в других родственных компонентах.

2. Определить тип обработчика события. Обработчик события может просто опознавать *событие нотификации* (объектного типа **TNotifyEvent**) или содержать обработку событий специфического типа. **TNotifyEvent** задает единственный аргумент указатель объекта, благодаря которому обработчик «узнает» компонент, сгенерировавший это событие. Например, щелчки мышью генерируют события нотификации. Обработчик этих событий знает только то, на каком компоненте пользователь щелкнул мышью. Передача дополнительных аргумен-

тов используется, например, обработчиком события типа **TKeyPressEvent**, вызываемого при нажатии любой клавиши на клавиатуре.

3. Объявить тип и свойство для события. Инспектор объектов определяет, что данное свойство является событием, обнаруживая тип свойства **closure**, и представляет его на вкладке События. Событиям необходимо назначать значимые и описательные имена, по которым пользователь компонента догадается о том, что это за событие. C++Builder рекомендует начинать имена событий с префикса «On».

4. Создать виртуальный метод, который вызывает обработчик события пользователя и обеспечивает обработку по умолчанию. Правильное функционирование нашего компонента не должно зависеть от конкретной реакции, которую пользователь заложил в обработчик события. В частности, пустой обработчик события так же допустим, как и его отсутствие. Более того, пользователь имеет право переопределить обработку по умолчанию. Чтобы предоставить ему такую возможность, нужно передать в обработчик дополнительный адресный аргумент, значение которого можно проверять при возврате. При этом пустой обработчик события не изменит значения аргумента, и обработка по умолчанию всегда будет иметь место после возврата из пустого обработчика.

События и сообщения Windows. Опытный программист определенно заметит сходство некоторых событий C++Builder и сообщений Windows. В следующей таблице приведен краткий список событий объекта **TForm**[45] и соответствующих сообщений Windows, которые мы использовали бы в обычной программе на языке C:

Событие VCL	Сообщение Windows
OnCreate	WM_CREATE
OnClose	WM_DESTROY
OnResize	WM_SIZE
OnActivate, OnDeactivate	WM_ACTIVATE
OnShow, OnHide	WM_SHOWWINDOW
OnKeyDown	WM_KEYDOWN
OnKeyUp	WM_KEYUP
OnMouseDown	WM_LBUTTONDOWN, WM_RBUTTONDOWN
On Mouse Up	WM_LBUTTONUP,WM_RBUTTONUP
OnMouseMove	WM_MOUSEMOVE
OnDblClkk	WM_ LBUTTONDBLCLK, WM_RBUTTONDBLCLK
OnPaint	WM_PAINT

Не всякому сообщению Windows можно найти соответствующее событие VCL. Например, в обычной программе на языке C для Windows сообщение **WM_COMMAND** используется как для обслуживания нажатий кнопок, так и выбора команд из меню. В C++Builder для этих целей используются разные события, например: **TButton::OnClick** и **TMenuItem::OnClick** соответственно.

[45] Подробная информация о событиях класса **TForm** приведена в п. 12.14.3.

С другой стороны, некоторые события VCL расширяют функциональность встроенных сообщений Windows. Так, события **OnDragOver** и **OnDragDrop** экземпляра класса **TForm** просто и прямолинейно реализуют операции перетаскивания (drag-and-drop) в нашей программе. Большинство компонентов на вкладках **Standard и Win95** Палитры компонентов лишь специальным образом обрамляют известные элементы управления Windows. Компоненты на других вкладках представляют совершенно новые элементы управления (и события) для особых областей функционирования.

Компоненты вкладок **Standard и Win95** инкапсулируют стандартные элементы управления Windows. За взаимодействие между пользователем и программой, которое ранее поддерживалось реакцией на сообщения Windows, теперь отвечают обработчики событий компонентов VCL. Однако в некоторых ситуациях возникает необходимость «взять на себя» те сообщения Windows, которые не имеют соответствующих событий VCL или неадекватны им. Для таких случаев в VCL предусмотрена методика объектно-ориентированного программирования, обеспечивающая непосредственный отклик на события Windows, подобно средствам библиотек базовых классов OWL или MFC. Эта методика, реализуемая с помощью макросов **BEGIN_MESSAGE_MAP**, **MESSAGE.HANDLER** и **END_MESSAGE_MAP**, весьма трудоемка. Поэтому рекомедуется предварительно тщательно просмотреть имеющиеся в VCL компоненты, которые могут содержать подходящие события.

Контрольные вопросы

1. Как используются «стандартные» события при создании компонентов?
2. Как используются «собственные» события при создании компонентов?
3. Какие рекомендации используются при необходимости обработки событий и сообщений Windows?

12.1.5. Методы компонентов

Компонентные методы ничем не отличаются от других объектных методов. Хотя C++Builder не вводит никаких специальных ограничений на оформление компонентных методов, имеется ряд правил, которых стоит придерживаться.

1. Необходимо минимизировать число методов, которые вызывает прикладной программист, чтобы использовать наш компонент. Многое из того, что мы намеревались реализовать в виде методов, вероятно, лучше инкапсулировать в свойства компонента. В отличие от свойств, методы работают только во время исполнения программы.

2. Необходимо избегать взаимной зависимости методов, максимально изолируя методы друг от друга. Нельзя требовать, чтобы методы выполнялись в определенном порядке. Никакой метод не должен переводить компонент в такое состояние, когда вызовы других методов станут запрещенными, так как смогут нарушить нормальное функционирование компонента.

3. Необходимо придерживаться соглашения об именах методов. Названия должны быть описательными, т. е. включать глагол действия и отражать возвращаемое значение (например, метод **PasteFromClipboard** передает в компонент данные из доски объявлений, а метод **GetHorizontalPosition** возвращает горизонтальную позицию некоторой величины).

4. Необходимо правильно защищать методы. Все методы (включая конструкторы и деструкторы), к которым имеют право обращаться пользователи нашего компонента, следует объявлять как **public.** Все методы, к которым имеют право обращаться производные объекты нашего компонента, следует *объявлять* как **protected,** что запрещает пользователям преждевременно вызывать методы, данные для которых еще не созданы. Только те компонентные методы, которые реализуют доступ к свойствам компонента, должны быть объявлены как **private,** поскольку в этом случае пользователи оперируют со значениями свойств напрямую.

5. Необходимо объявлять методы виртуальными, когда мы желаем обеспечить полиморфное поведение переопределенных версий в разных классах.

Вызовы статических методов. Все объектные методы определяются как статические **(static)** по умолчанию, если мы не объявили их иначе. Вызовы статических методов выглядят как вызовы обычных функций: определяемый при компиляции адрес метода связывает его с объектом. Поэтому обращения к статическим методам выполняются быстрее, нежели обращения к виртуальным методам, адреса которых неизвестны на этапе компиляции. Напомним правила перекрытия статических методов примером:

```
class TFirstComponent : public TComponent
{
      public:
           void _fastcall Move() ;
           void _fastcall Flash();
};
class TSecondComponent : public TFirstComponent
{
      public:
          void _fastcall Move();              // отличен от унаследованного метода
          int _fastcall Flash(int HowOften);     // отличается сигнатурой
};.
```

Адреса статических методов не меняются при наследовании, а значит, статические методы не следует переопределять — они призваны выполнять одинаковые действия во всех родственных компонентных объектах. Рассмотренный пример показывает, что переопределение статических методов в производном классе, по сути, перекрывает родительские методы.

Вызовы виртуальных методов. Вызов виртуального метода записывается точно так же, как вызовы других функций, однако механизм обращения слегка усложняется, делая его более гибким. Поскольку виртуальные методы могут быть переопределены в производных классах, их адреса невозможно определить на стадии компиляции. Фактические адреса виртуальных методов становятся известными только во время выполнения программы.

Ключевое слово **virtual** в объявлении метода создает запись в таблице виртуальных методов объекта. При выполнении программы эти записи преобразуются в фактические адреса всех виртуальных методов данного компонентного класса. Когда мы производим новый объектный класс, он наследует таблицу виртуальных методов всех своих предшественников, добавляя к таблице дополнительные виртуальные методы, объявленные в производном классе. Кроме того, производный класс может переопределять наследованные виртуальные методы. Любой конструктор компонента всегда должен быть виртуальным.

Контрольные вопросы

1. Какие рекомендации существуют для включения методов в компонент?

12.2. Базовые классы

Все классы библиотеки визуальных компонентов произошли от группы базовых классов, которые лежат в основе иерархии VCL. Самый общий предок компонентов — это класс **TObject**, инкапсулирующий простейший объект. Как известно (см. главы 1–8), каждый объект наследует свойства и методы родительского класса. К объекту можно добавить новые свойства и методы, но нельзя удалить унаследованные. Объект-наследник, в свою очередь, может стать родительским для нового класса, который унаследует возможности всех своих предков. Иерархия базовых классов VCL продумана чрезвычайно тщательно — ведь на ее основе создано все множество компонентов, используемых в RAD Studio, и программисту предоставлены широкие возможности для создания новых компонентов. Особое место среди базовых классов, помимо **TObject**, занимают **TComponent** (от него происходят все компоненты) и **TControl** (от него происходят все элементы управления).

В основе всего многообразия классов и компонентов, используемых в RAD Studio, лежат всего лишь семь базовых классов, иерархия наследования которых показана на рисунке 12.3. Они обеспечивают выполнение основных функций любого объекта — будь это стандартный компонент VCL или специализированный объект, выполняющий некоторые операции в приложении.

Благодаря механизму наследования свойств и методов, потомки базовых классов умеют общаться друг с другом; работают в среде разработки, взаимодействуя с Палитрой компонентов и Инспектором объектов; распознаются операционной системой как элементы управления и окна.

В основе иерархии классов находится класс **TObject**. Он обеспечивает выполнение важнейших функций жизнедеятельности любого объекта. Благодаря ему каждый класс получает в наследство механизмы создания и использования экземпляра класса, а также — механизм ликвидации объекта.

Класс **TPersistent** происходит непосредственно от класса **TObject**. Класс **TPersistent** является базовым классом для всех объектов VCL, допускающих операцию присваивания или участвующих своими свойствами в операциях с потоками. Для этого определен специальный метод присваивания Assign. Класс обеспечивает своих потомков возможностью взаимодействовать с другими объектами и процессами на уровне данных. Его методы позволяют передавать

данные в потоки, обмениваться информацией с буфером обмена, а также обеспечивают взаимодействие объекта с Инспектором объектов.

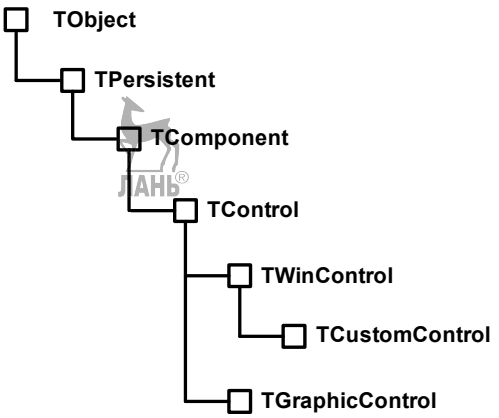


Рис. 12.3
Базовые классы библиотеки визуальных компонентов

Класс **TComponent** является важнейшим для всех компонентов. Наследуя его, можно создавать любые компоненты. Механизмы, реализованные в классе **TComponent**, обеспечивают взаимодействие компонента со средой разработки, главным образом с Палитрой компонентов и Инспектором объектов. Благодаря возможностям этого класса, компоненты могут работать на форме проекта как объекты уже на этапе разработки.

Класс **TControl** происходит от класса **TComponent**, является базовым классом для всех визуальных компонентов. Его основное назначение — обеспечить функционирование визуальных компонентов. Каждый визуальный компонент, произошедший от **TControl**, наделяется основными признаками элемента управления. Благодаря этому каждый визуальный компонент умеет работать с GUI (Graphic User Interface — графический интерфейс пользователя операционной системы Windows) и отображать себя на экране. В классе определено множество свойств, описывающих размещение компонента, его размеры, выравнивание, видимость, перетаскивание и т. д. Определено много свойств и все основные события, связанные с клавиатурой и мышью.

Класс **TWinControl** расширяет возможности разработчиков по созданию элементов управления. Он наследует класс **TControl** и обеспечивает создание оконных элементов управления. Определены свойства и методы, характеризующие вид окна, фокусировку, последовательность табуляции и т. д.

На основе класса **TWinControl** создан еще один дополнительный класс — **TCustomControl**. Он обеспечивает создаваемые на его основе компоненты возможностями по использованию канвы — специального объекта класса **TCanvas**, предназначенного для отображения графики. Класс **TCustomControl** является общим предком для целой группы классов, обеспечивающих создание различных нестандартных типов оконных (получающих фокус) элементов управления Windows: редакторов, списков и т. д.

Для создания не оконных (не получающих фокус) элементов управления используется класс **TGraphicControl**, являющийся потомком класса **TControl**.

В целом, иерархия базовых классов обеспечивает программиста полноценными возможностями работы в RAD Studio и позволяет проектировать многие виды приложений.

Ниже мы остановимся на основных свойствах и методах базовых классов, выделяя только те, которые с высокой вероятностью могут пригодиться в реальной работе. Часть из них доступна в Инспекторе объектов, часть может быть использована в программном коде.

Библиотека визуальных компонентов является объектно-ориентированной надстройкой над функциями Windows API. VCL, по сути, полностью прячет Windows API от программиста, не требуя от него глубоких знаний операционной системы.

При построении библиотеки визуальных компонентов интенсивно использовалась идея наследования классов, поэтому мы имеем дело с очень развитой иерархией. На рисунке 12.4 приведен фрагмент иерархии компонентов. Перед названием класса каждого компонента указано наименование пространства имен стандартной поставки RAD Studio, в котором находится определение компонента.

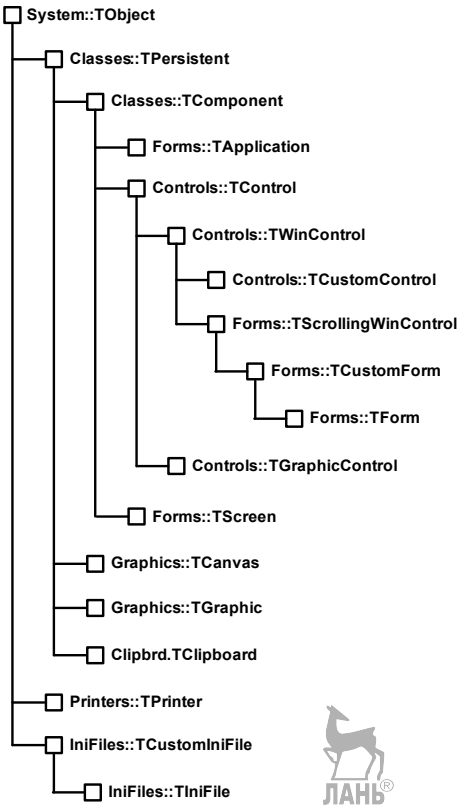


Рис. 12.4
Фрагмент иерархии компонентов

Наша задача состоит в том, чтобы изучить принципы построения компонентов и библиотеки визуальных компонентов. Для этого мы будем продвигаться по иерархии от корня к листьям. Изучение каждого компонента начинается с рассмотрения метафоры его построения. Затем приводятся справочные данные по важнейшим свойствам и методам, которые впервые появляются в рассматриваемом компоненте.

Класс TObject, подробно рассмотренный в главе 6, сосредоточил все свойства «быть объектом». Расположение этого класса в корне иерархии означает, что все потомки автоматически содержат средства для конструирования и удаления объектов.

Контрольные вопросы

1. Какой класс инкапсулирует свойства «быть объектом»?
2. Сформулируйте понятие «компонент».
3. Опишите назначение библиотеки визуальных компонентов.

12.3. Класс TObject

Класс **TObject** является базовым классом для всех прочих порождаемых классов. **TObject** инкапсулирует общее для всех объектов системы C++Builder функциональное поведение, обусловленное методами, которые обеспечивают:

— способность конструктора создавать, а деструктора разрушать объект-экземпляр класса в динамической памяти. Конструктор TObject возвращает указатель на создаваемый объект;

— информацию об имени, типе производного объекта и его свойствах, которые объявлены как **_published**;

— поддержку обработки сообщений.

Иерархия отсутствует, так как этот класс является основным родителем всех других классов.

Свойства и события у класса **TObject** отсутствуют.

Методы класса TObject рассмотрены в параграфе 6.2. Большинство методов предназначены для внутреннего использования средой C++Builder, поэтому не следует прямо обращаться к ним из программы. Часть методов TObject объявлена как статические (с ключевым словом **static).** Это означает, что нам не нужно создавать экземпляр данного класса для того, чтобы обратиться к его статическим методам.

Контрольные вопросы

1. Какое функциональное поведение обеспечивает класс **TObject**?

12.4. Класс TPersistent

Класс **TPersistent** непосредственно произведен от класса **TObject**. Этот абстрактный класс не определяет никаких специальных свойств или событий, однако его производные приобретают особые способности присваивания и поточности. «Persistent» в переводе с английского означает «устойчивый», «по-

стоянный». Что же такого постоянного в одноименном классе? Ответ таков: виртуальный метод

virtual void __fastcall Assign(TPersistent * Source);.

Этот важнейший метод осуществляет копирование содержимого одного объекта (**Source**) в другой объект, вызвавший метод **Assign**. При этом объект-получатель остается самим собой, чего нельзя достигнуть, используя простое присваивание переменных объектного типа вида

FirstObject=SecondObject;.

В последнем случае указатель на одну область адресного пространства, содержащую экземпляр класса (объект), замещается указателем на другую область адресного пространства, содержащую другой объект.

Метод **Assign** копирует данные одного объекта в другой. Метод **Assign** переопределяется в классах, производных от класса **TPersistent**. Некоторое число классов C++Builder поддерживает присваивание объектов разных типов. Для большинства же классов, производных от **TPersistent**, применение **Assign** к несовпадающим типам объектов источника и назначения ведет к генерации исключения **EConvertError**.

В отличие от обычного присваивания, в котором указатель на <объект-назначение> начинает указывать на <обьект-источник>, метод **Assign** создает новую копию объекта. После применения **Assign** имеется два объекта с одинаковыми данными.

Если объекты разного типа, то при вызове вида **X–>Assign(Y)** класс объекта **X** должен «знать», как скопировать в него тип **Y** (тип **Y** может ничего не знать о преобразовании типов). Если метод **Assign** не может осуществить преобразование типов, то он вызывает защищенный метод **AssignTo**, объявленный в классе **TPersistent** и переопределенный в классах, производных от него. Вызов имеет вид **X–>AssignTo(Y)**. Если и метод **AssignTo** не может осуществить преобразование или если он не переопределен, то вызывается **AssignTo** класса **TPersistent** и генерируется исключение.

Таким образом, метод **Assign** позволяет продублировать объект — присвоить одному объекту значения всех свойств другого объекта. При этом объекты не обязательно должны быть одного и того же класса; более того, они не обязательно должны находиться в отношениях «родитель-потомок». Данный метод тем и хорош, что позволяет полиморфное присваивание. Например, конструкция

Clipboard–>Assign(Picture);

позволяет скопировать содержимое картинки **Picture** в буфер обмена Windows (объект **Clipboard**), а вызов

Picture–>Assign(Clipboard);

приводит к обратному действию — вставке содержимого буфера обмена в битовую картинку.

Возникает вопрос: не проще ли выполнить присваивание **Clipboard=Picture**?

Ответ однозначный: нет. Даже транслятор не позволит написать это присваивание. Эти два класса находятся на разных ветвях иерархии и не совместимы по присваиванию. В параграфе 6.3 мы выяснили, что присваивание объекта возможно только объекту, который является экземпляром класса-предка первого объекта. Использование метода **Assign** позволяет сделать возможным невозможное! Какова здесь логика? Известно, что в папку обмена можно поместить растровую картинку, текст, метафайл, мультимедийные данные и т. п. Метод **Assign** класса **TClipboard** переписан таким образом, чтобы обеспечить присвоение (т. е. реальное перемещение в папку обмена) всех этих данных.

Для обеспечения взаимодействия потомков класса **TPersistent** со средой разработки предназначен метод

System::UnicodeString __fastcall GetNamePath(void).

Он возвращает имя объекта для передачи его Инспектору объектов.

Для взаимодействия с потоками при загрузке и сохранении компонентов предназначен следующий метод:

virtual void __fastcall DefineProperties(TFiler* Filer).

Благодаря методам **TPersistent** появилась возможность пересылки компонентов из оперативной памяти во внешнюю память и обратно.

Класс **TPersistent** никогда не используется напрямую, от него порождаются потомки, которые должны уметь передавать другим объектам значения своих свойств, но не являться при этом компонентами. Экземпляры класса **TPersistent** создаваться не могут. Класс используется только для создания производных классов. События класса **TPersistent** отсутствуют.

Контрольные вопросы

1. Поясните назначение класса **TPersistent**.
2. Какие возможности предоставляет программисту метод **Assign**?
3. Какие возможности класс **TPersistent** унаследовал от класса **TObject**?

12.5. Класс TComponent

Класс **TComponent** непосредственно произведен от класса **TPersistent**. Класс **TComponent** используется в качестве основы для создания компонентов и реализует основные механизмы, которые обеспечивают функционирование любого компонента. Все компоненты являются производными от **TComponent** и могут находиться в его владении.

TComponent инкапсулирует общее для всех компонентов функциональное поведение, обусловленное свойствами и методами, которые обеспечивают:

— перенос компонентов на форму из Палитры компонентов и манипуляции в окне Редактора форм;

— способность владения и обслуживания других компонентов;

— специальные характеристики поточности (ввода-вывода), с которыми может манипулировать Инспектор объектов на этапе проектирования;

— возможность манипулирования некоторыми невидимыми компонентами на стадии проектирования.

Класс **TComponent** является предком для всех компонентов библиотеки. В общем случае любой компонент является хранилищем других компонентов. Эта метафора изображена на рисунке 12.5. Каждый компонент представляет собой массив (список) компонентов. Компонент, являющийся элементом массива компонентов, в свою очередь, сам может являться массивом компонентов.

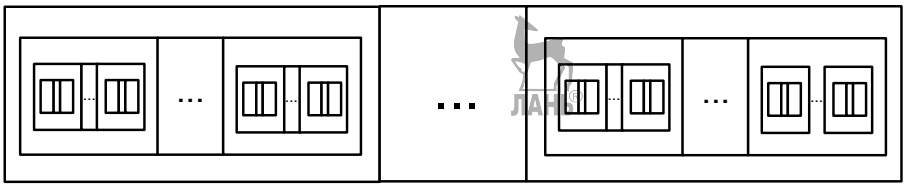


Рис. 12.5
Массив компонентов TComponent

В отличие от традиционного массива, доступ к компоненту может осуществляться как по индексу, так и по имени. Наличие имени компонента имеет принципиальное значение. Под этим именем компонент находится в списке Инспектора объектов и других составных частей среды RAD Studio. Имя компонента, используемое для идентификации компонента в приложении, хранится в опубликованном свойстве:

__property TComponentName Name;

где **TComponentName** является синонимом типа **UnicodeString**.

Имя компонента не может быть пустой строкой; в модуле не может быть двух компонентов с одинаковыми именами. Размещая компонент-класс на форме, мы фактически имеем дело с готовым объектом, при этом имя компонента используется для автоматической генерации имени соответствующего объекта. Компонент-объект знает свое имя на этапе выполнения программы, но что еще более важно — на этапе визуального проектирования приложения.

Задание. Выделив на форме компонент, найдите свойство **Name** в инспекторе объектов. Измените имя компонента и убедитесь, что в тексте программы это имя стало новым именем компонента.

Список компонентов (рис. 12.5) представляет собой индексированный список указателей всех компонентов, для которых данный компонент является владельцем (owner). Список компонентов доступен через свойство:

__property TComponent* Components[int Index].

В классе **TComponent** предусмотрен также набор свойств и методов для управления этим списком (табл. 12.1).

Таблица 12.1

Свойства и методы для управления списком компонентов

__property int ComponentCount	Свойство хранит информацию о количестве компонентов в списке
TComponent * __fastcall FindComponent (const System::UnicodeString AName);	Функция возвращает ссылку на найденный в списке компонент с заданным именем **AName**
void __fastcall InsertComponent (TComponent*AComponent);	Функция вставляет компонент **AComponent** в конец списка
void __fastcall RemoveComponent (TComponent*AComponent);	Функция удаляет компонент **AComponent** из списка
void __fastcall DestroyComponents (void);	Функция удаляет все компоненты из списка

Для компонентов существует иерархия владения, поэтому в классе введен механизм учета и управления компонентами, для которых данный компонент является владельцем: каждый компонент имеет владельца и сам может быть владельцем других компонентов, которые сведены в единый список. Указатель на владельца есть свойство:

__property TComponent * Owner.

Положение компонента в списке его владельца можно узнать и изменить, пользуясь свойством:

__property int ComponentIndex.

В конструкторе каждому компоненту передается указатель на **Owner**, это означает, что компонент будет находиться в списке этого владельца:

__fastcall virtual TComponent(TComponent* AOwner).

Владелец (**Owner)** — это тот объект, который при создании вызывает конструкторы всех объектов, владельцем которых он является, а при уничтожении — их деструкторы. Таким образом, поместив при разработке компонент на форму, мы можем теперь не заботиться о его создании и уничтожении в программе. Все свойства таких компонентов и все связи между ними запоминаются в файле формы **.dfm** и при запуске приложения создаются вновь автоматически.

Задание. Создайте новый проект и разместите на форме любой компонент. Запомните положение компонента на форме. Сохраните и закройте проект. Найдите в папке сохраненного проекта файл с расширением .dfm и откройте его в блокноте. Измените значения свойств **Top** и **Left** нашего компонента и сохраните файл. Запустив приложение, убедитесь, что положение компонента на форме изменилось.

Деструктор компонента:

__fastcall virtual ~TComponent(void).

уничтожает все дочерние компоненты (вызывая их деструкторы) и вычеркивает себя из списка компонента-владельца.

Конструкторы нужно вызывать только для тех компонентов, которые создаются нами во время исполнения программы. При этом они получают владельца, поэтому в дальнейшем явный вызов деструктора не понадобится. Практически все конструкторы и деструкторы описанных ниже компонентов имеют тот же вид, что приводится здесь. Поэтому в изложении их описания опущены, где это возможно.

Рассмотрим другие свойства компонента. В данной ситуации компонент рассматривается как элемент управления. Элемент управления может находиться в одном из множества состояний:

__property TComponentState ComponentState;,

где **TComponentState** представляет собой множество вида **{csLoading, csReading, csWriting, csDestroying, csDesigning, csAncestor, csUpdating, csFixups, csFreeNotification, csInline, csDesignInstance}**.

Эти состояния используются средой во время визуального проектирования, при загрузке/выгрузке компонентов из файла ресурсов. Описание элементов множества состояний компонента приведено в таблице 12.2.

Таблица 12.2

Возможные состояния элемента управления

Состояние	Описание
csLoading	Устанавливается при загрузке компонента из потока
csReading	Устанавливается при чтении значений свойств из потока
csWriting	Устанавливается при записи значений свойств в поток
csDestroying	Устанавливается при уничтожении компонента
csDesigning	Состояние разработки. Устанавливается при работе с формой во время проектирования
csAncestor	Устанавливается при переносе компонента на форму. Для перехода в это состояние должно быть уже установлено состояние **csDesigning**
csUpdating	Устанавливается при изменении значений свойств и отображения результата на форме-владельце. Для перехода в это состояние должно быть уже установлено состояние **csAncestor**
csFixups	Устанавливается, если компонент связан с компонентом другой формы, которая еще не загружена в среду разработки
csFreeNotification	Если это состояние устанавливается, другие компоненты, связанные с данным, уведомляются об его уничтожении
cslnline	Определяет компонент верхнего уровня в иерархии владения. Используется для обозначения корневого объекта в разворачивающихся свойствах
csDesignlnstance	Определяет корневой компонент на этапе разработки

Во время исполнения приложения множество состояний пусто. Лишь метод

void __fastcall Destroying(void);

устанавливает состояние в значение **csDestroying** и вызывает аналогичный метод у всех потомков из списка. Он также используется в деструкторе компонента.

Свойство

__property int Tag

является вспомогательным и не влияет на работу компонента. Оно отдано на откуп прикладному программисту, который может присваивать ему значения по своему усмотрению. Например, это свойство можно использовать для дополнительной, более удобной идентификации компонентов.

Таким образом, класс **TComponent** сосредоточил в себе свойства «быть программным компонентом» и является основой для повторного использования компонентов и для создания собственных компонентов.

Экземпляры класса **TComponent** не создаются. Класс **TComponent** используется как базовый, когда объявляется класс невизуального компонента, который может присутствовать в палитре компонентов и применяться в процессе проектирования.

Контрольные вопросы

1. Поясните назначение класса **TComponent**.
2. Какие способы доступа к элементам списка компонентов предусмотрены в классе **TComponent**?
3. Опишите смысл понятия «Владелец» («Owner»).
4. Как создаются и уничтожаются стандартные компоненты?
5. Какие возможности класс **TComponent** унаследовал от класса **TObject**?

12.6. Класс TCanvas

Многие компоненты C++Builder имеют свойство **Canvas** (канва), представляющее собой область компонента, на которой можно рисовать или отбражать готовые изображения. Это свойство имеют формы, графические компоненты **Image**, **PaintBox**, **Bitmap** и многие другие. Канва содержит свойства и методы, существенно упрощающие программирование графики. Все сложные взаимодействия с системой спрятаны для пользователя, поэтому рисовать в C++Builder может программист, совершенно не искушенный в машинной графике. Использование графики в операционной системе рассмотрено в параграфе 5.4. Мы рассмотрим использование графической информации с помощью компонента **TCanvas**.

Иерахия наследования имеет вид **TCanvas → TCustomCanvas → TPersistent → TObject**. Класс **TCustomCanvas** имеет вспомогательное назначение. В нем определены многие методы как абстрактные, в классе **TCanvas** эти же методы переопределяются. Это сделано с целью предоставить возможность пользователю создавать собственные модификации класса канва путем создания помков **TCustomCanvas** или потомков **TCanvas** с последующим переопределением некоторых или всех методов.

Обычно компьютерную графику разделяют на векторную и растровую, хотя обособляют еще и фрактальный тип представления изображений.

Векторная графика представляет изображение как набор геометрических примитивов. Обычно в качестве них выбираются точки, прямые, окружности, прямоугольники, а также как общий случай, кривые некоторого порядка. Объектам присваиваются специальные атрибуты, например, толщина линий, цвет заполнения. Рисунок хранится как набор координат, векторов и других чисел, характеризующих набор примитивов. При воспроизведении перекрывающихся объектов имеет значение их порядок.

Изображение в векторном формате дает простор для редактирования. Изображение может без потерь масштабироваться, поворачиваться, деформироваться, также имитация трехмерности в векторной графике проще, чем в растровой. Дело в том, что каждое такое преобразование фактически выполняется так: старое изображение (или фрагмент) стирается, и вместо него строится новое. Математическое описание векторного рисунка остается прежним, изменяются только значения некоторых параметров, например, коэффициентов.

Вместе с тем, не всякое изображение можно представить как набор из примитивов. Такой способ представления хорош для схем, используется также для масштабируемых шрифтов, деловой графики.

Растровая графика всегда оперирует двумерным массивом (матрицей) пикселей. Каждому пикселу сопоставляется значение яркости, цвета, прозрачности или комбинация этих значений. Растровый образ имеет некоторое число строк и столбцов.

Без особых потерь растровые изображения можно только лишь уменьшать, хотя некоторые детали изображения тогда исчезнут навсегда, что иначе в векторном представлении. Увеличение же растровых изображений оборачивается видом на увеличенные квадраты того или иного цвета, которые раньше были пикселами.

В растровом виде представимо любое изображение, однако этот способ хранения имеет свои недостатки: больший объем памяти, необходимый для работы с изображениями, потери при редактировании.

Фрактальная графика. Фрактал — это объект, отдельные элементы которого наследуют свойства родительских структур. С чем можно сравнить фрактальное изображение? Ну, например, со сложной структурой кристалла, со снежинкой, элементы которой выстраиваются в одну сложную композицию. Это свойство фрактального объекта может быть удачно использовано для создания орнамента или декоративной композиции. Фрактальная графика обычно реализуется только в специальных системах машинной графики.

Каждая точка канвы имеет координаты X и Y. Система координат канвы, как и везде в C++Builder, имеет началом левый верхний угол канвы. Координата X возрастает при перемещении слева направо, а координата Y — при перемещении сверху вниз.

Пиксел — это наименьший элемент поверхности рисунка, которым можно манипулировать. Важнейшее свойство пиксела — его цвет, для описания цвета используется тип **TColor**.

Рисование в растровой графике осуществляется по пикселам. Для этого используется свойство канвы

__property System::Uitypes::TColor Pixels[int X][int Y],

которое представляет собой двумерный массив и отвечает за цвет канвы. Например, можно прочитать цвет пиксела (5, 10): **TColor Picsel=Picsels[5][10].** С массивом пикселов можно обращаться как с любым свойством: изменить цвет, задавая пикселу новое значение, или определять его цвет по хранящемуся в нем значению. Например, **Canvas–>Picsels[5][10]=clRed** — это придание пикселу красного цвета.

Рассмотрим более содержательный пример. Необходимо нарисовать график некоторой функции **F*(x)*** на канве компонента **Image1**, если известен диапазон ее изменения **Ymin**, **Ymax** и диапазон изменения аргумента **Xmin**, **Xmax**. Эту задачу для функции **Y=Sin(X)** можно решить следующим образом:

```
void __fastcall TForm2::Button1Click(TObject *Sender)
{
   #define Pi 3.14159
   int Xmin=0; int Xmax=4 * Pi;
   int Ymin=-1; int Ymax=1;
   double X, Y;          // координаты функции
   int PX, PY;           // координаты пикселов
   for (PX=0; PX <= Image1–>Width; PX++)
       {
                X=PX * 4 * Pi/Image1–>Width;
                Y=sin(X);
                PY=Image1–>Height – (Y+1) * Image1–>Height/2;
          //    устанавливаем цвет пиксела
                Image1–>Canvas–>Pixels[PX][PY]=clBlack;
       }
}.
```

Результат представлен на рисунке 12.6.

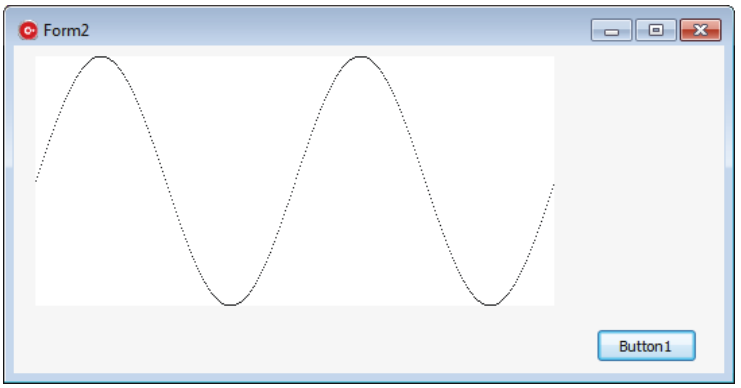


Рис. 12.6
Рисование графика функции **Y=Sin(X)** на компоненте **TImage**

Рисование в векторной графике осуществляется с помощью пера. У канвы имеется свойство **Pen** — перо вида **__property TPen*Pen**. В свою очередь, класс **TPen** имеет четыре опубликованных свойства

__property System::Uitypes::TColor Color,
__property TPenMode Mode,
__property TPenStyle Style,
__property int Width.

Свойство **Color** моделирует цвет, которым наносится рисунок.

Свойство **Mode** позволяет комбинировать использования цветов пера и пиксела. По умолчанию значение **Mode** равно **pmCopy**. Это означает, что линии проводятся цветом, заданным в свойстве **Color** пера. Но возможны и другие режимы, в которых учитывается не только цвет пера, но и цвет пиксела. Например, режим **pmNotXor** означает побитное сложение по инверсному исключающего или цвета пиксела и цвета пера: при равенстве битов операндов результат равен **1** иначе **0**. Пусть цвет пиксела имеет значение **01011011**, а цвет пера **11100011**. Тогда результирующий цвет пиксела будет равен **01000111**.

Свойство **Style** определяет вид линии (сплошная, пунктирная, штрих-пунктирная и т. д.).

Свойство **Width** задает ширину (толщину) линии в пикселах. По умолчанию **Width=1**.

Мы рассмотрели устройство пера. Теперь речь пойдет о том, как перо рисует. У канвы имеется свойство **__property System::Types::TPoint PenPos**. Свойство **PenPos** определяет в координатах канвы текущую позицию пера. Перо может переместиться от текущей позиции в позицию заданную. Операция перемещения пера имеет две разновидности. Первая разновидность операции перемещения имеет вид **MoveTo (X, Y)**: перо перемещается от текущей позиции к позиции **(X, Y)** в «поднятом» состоянии, т. е. не оставляя следа на канве. Вторая разновидность операции перемещения пера **LineTo (X, Y)** выполняет аналогичное действие, но с «опущенным» пером, т. е. оставляя на канве отрезок прямой линии. Термины «поднятое», «опущенное» перо унаследованы от графопостроителей — электромеханических устройств, которые могли обрабатывать лист ватмана пером, которое могло перемещаться в поднятом или опущенном состоянии. После выполнения обеих операций текущей позицией пера становится позиция **(X, Y)**, указанная в параметрах операции.

В качестве примера векторной графики переделаем пример рисования функции **Y=Sin(X)**.

```
void __fastcall TForm2::Button1Click(TObject *Sender)
{
   #define Pi 3.14159
   int Xmin=0; int Xmax=4 * Pi;
   int Ymin=-1; int Ymax=1;
   double X, Y;        // координаты функции
   int PX, PY;         // координаты пикселов
   Image1–>Canvas–>MoveTo(0, Image1–> Height/2);
   for (PX=0; PX <= Image1–>Width; PX++)
      {
        X=PX * 4 * Pi/Image1–>Width;
        Y=sin(X);
```

```
        PY=Image1–>Height – (Y+1) * Image1–>Height/2;
        Image1–>Canvas–>LineTo(PX, PY);
    }
}.
```

Результат будет таким же, как на рисунке 12.6, за исключением того, что линия графика функции будет выглядеть непрерывной.

Рисование любых фигур осуществляется с помощью рассмотренных методов. В классе **TCanvas** определены также методы, позволяющие нарисовать: **Arc** — дугу окружности или эллипса; **Chord** — замкнутую фигуру, ограниченную дугой окружности или эллипса и хордой; **Ellipse** — окружность или эллипс; **Pie** — сектор окружности или эллипса; **Poligon —** замкнутую фигуру с кусочно-линейной границей; **Poliline** — кусочно-линейную кривую; **Rectangle** — прямоугольник; **RoundRect** — прямоугольник со скругленными углами.

В заключение отметим, что все рассмотренные в этом параграфе операции базируются на операциях графического интерфейса GDI, рассмотренного в параграфе 5.5.

Контрольные вопросы

1. Каково назначение компонента **TCanvas**?
2. Какую иерархию наследования имеет класс **TCanvas**?
3. Какие виды графики реализуются в компьютерной графике?
4. Какие виды графики используются в компоненте **TCanvas**?
5. Как реализуется растровая графика в компоненте **TCanvas**?
6. Как реализуется векторная графика в компоненте **TCanvas**?
7. Какие средства операционной системы используются в реализации компонента **TCanvas**?
8. Какие возможности класс **TCanvas** унаследовал от класса **TObject**?

12.7. Класс TControl

Класс **TControl** является базовым классом для всех визуальных компонентов C++Builder, т. е. для компонентов, которые пользователь может видеть и которыми манипулирует во время выполнения приложения. Все они имеют общие свойства, методы и события, определяющие место их размещения, расцветку, реакцию на нажатие клавиш или кнопок мыши и т. д.

Защищенные свойства и методы класса **TControl** используются в их потомках. Если требуется создать новый класс визуального компонента, его надо создавать как производный от **TControl** или от его потомков.

12.7.1. Свойства и методы

Класс **TControl** является потомком класса **TComponent** и предназначен для придания компонентам свойства визуальности, т. е. обеспечения механизмами отображения компонента на экране. Визуальные компоненты — это разнообразные стандартные для Windows и специальные элементы управления. В классе **TControl** используется множество новых свойств и методов. Недаром в Инспекторе объектов RAD Studio появилась категоризация методов и свойств.

Большинство из них вводятся как раз в классах **TControl** и **TWinControl**. Рассмотрим только важнейшие свойства и методы по категориям.

Местоположение и размер визуального компонента. Для определения местоположения и размеров визуального компонента введены два опубликованных свойства для задания координат левого верхнего угла:

__property int Top;
__property int Left;

и два опубликованных свойства для определения размеров:

__property int Height;
__property int Width;.

Смысл перечисленных свойств изображен на рисунке 12.7. Значения свойств задаются в пикселах. Для определения местоположения используется система координат (X, Y) рабочей области владельца данного компонента. Начало отсчета находится в левом верхнем углу. Оси направлены вправо и вниз. Под рабочей областью понимается та часть площади владельца (формы, панели), которая предназначена для размещения дочерних элементов. Эти свойства можно использовать как на этапе разработки, так и во время выполнения.

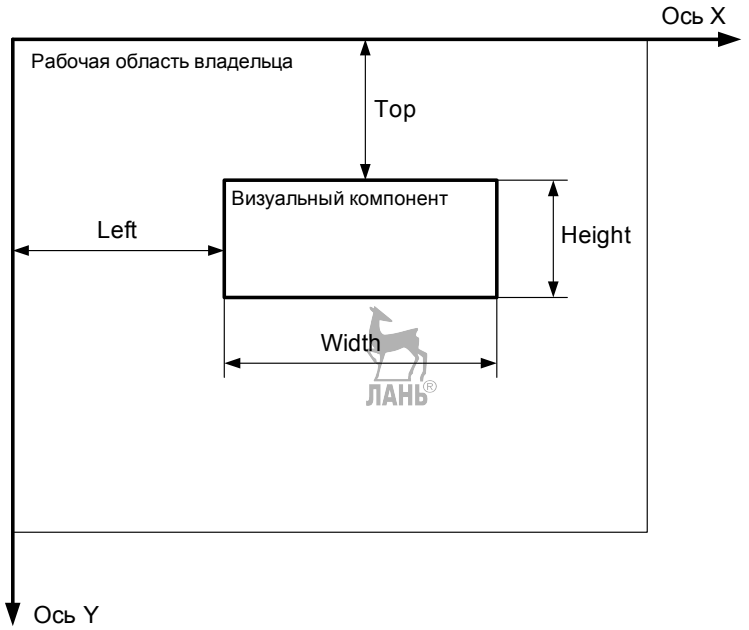


Рис. 12.7
Местоположение и размер визуального компонента

Для приведения визуального компонента в соответствие текущим значениям указанных выше свойств используется метод

void __fastcall AdjustSize(void);.

Свойство

__property Types::TPoint ClientOrigin

содержит координаты левого верхнего угла элемента управления в системе координат экрана, которая задается аналогичным рисунку 12.7 образом. Координаты любой точки можно пересчитать в экранные координаты при помощи метода

Types::TPoint __fastcall ClientToScreen(const Types::TPoint &Point);

и наоборот:

Types::TPoint __fastcall ScreenToClient(const Types::TPoint &Point).

Параметры рабочей области компонента определяются следующими свойствами:

__property int ClientHeight,
__property int ClientWidth,

которые хранят значения, соответственно, высоты и ширины рабочей области в пикселах.

Значение свойства

__property Types::TRect ClientRect

есть не что иное, как прямоугольник, заданный точками (**0**, **0**) и (**ClientWidth**, **ClientHeight**). Кому-то будет удобнее пользоваться этим свойством.

Если разработчику неизвестны текущие параметры рабочей области, то он может воспользоваться следующими методами.

Функция

virtual Types::TPoint __fastcall GetClientOrigin(void)

возвращает координаты левого верхнего угла рабочей области.

Функция

virtual Types::TRect __fastcall GetClientRect(void)

возвращает размеры прямоугольника рабочей области.

Выравнивание визуального компонента. Для выравнивания компонента в рабочей области его владельца (обычно это форма) применяется свойство

__property TAlign Align.

Тип **TAlign** объявлен следующим образом:

enum TAlign { alNone, alTop, alBottom, alLeft, alRight, alClient, alCustom };.

При значении **alNone** выравнивание отсутствует. При следующих четырех значениях выравнивание осуществляется по соответствующей стороне. Значение **alClient** приводит к тому, что элемент управления изменяет свои размеры до размеров рабочей области владельца.

Свойство

__property TAnchors Anchors;

представляет собой подмножество множества

typedef System::Set<TAnchorKind, TAnchorKind::akLeft,
TAnchorKind::akBottom>TAnchors;
enum TAnchorKind { akLeft, akTop, akRight, akBottom };

и обеспечивает при изменении размеров владельца фиксацию визуального компонента относительно сторон владельца. Положение компонента можно зафиксировать по одной, двум, трем или четырем сторонам владельца. При фиксации положения компонента по любой стороне расстояние между стороной и компонентом сохраняется неизменным. Комбинируя значения свойства **Anchors**, можно добиться различных вариантов поведения компонента при изменении размеров владельца.

Если по вертикали или горизонтали фиксация не установлена вообще, то при изменении размеров владельца компонент остается на том же месте с учетом изменившегося размера владельца.

Если положение компонента зафиксировано по вертикали или горизонтали, то при изменении размеров владельца размер компонента изменяется таким образом, чтобы расстояния до сторон владельца оставались неизменными.

Свойство

__property bool AutoSize

обеспечивает изменение размеров компонента в соответствии с размерами его содержимого (текста, изображения, списка, иерархического дерева и т. д.).

Внешний вид визуального компонента. При рассмотрении этих свойств компонент удобнее рассматривать как элемент управления.

Для определения цвета компонента используется свойство

__property Graphics::TColor Color.

При нахождении над компонентом указателя мыши его изображение может изменяться в соответствии со значением свойства

__property TCursor Cursor.

Свойство

__property bool DesktopFont

определяет возможность использования шрифта для отображения, который применяется операционной системой в значках.

Сам текст задается свойством

__property TCaption Text;

где тип **TCaption** определен в модуле Controls.hpp как

typedef System::UnicodeString TCaption;.

Длину текста можно определить при помощи функции

int __fastcall GetTextLen(void);,

которая возвращает число символов в тексте.

Свойство

__property TCaption Caption

хранит текст заголовка визуального компонента. Расположение заголовка компонента зависит от вида самого визуального компонента, обнаружить расположение заголовка нетрудно экспериментально.

Компонент можно сделать видимым или невидимым. Для этого применяется свойство

__property bool Visible.

Этих же результатов можно достичь методами **Show** (компонент видим) и **Hide** (компонент невидим).

Для каждого элемента управления можно создать собственное всплывающее меню. Ссылка на экземпляр класса **TPopupMenu**, инкапсулирующего такое меню, хранится в свойстве

__property Menus::TPopupMenu* PopupMenu.

Текущее состояние элемента управления определяется свойством **ControlState**:

__property TControlState ControlState,

где тип **TControlState** определен методом перечисления.

Описание возможных состояний элемента управления представлено в таблице 12.3.

Таблица 12.3

Возможные состояния элемента управления

Состояние	Описание
csLButtonDown	Левая кнопка мыши нажата, но еще не отпущена. Используется для реализации события **OnMouseDown**
csClicked	Левая кнопка мыши нажата, но еще не отпущена. Используется для реализации события **OnClick**
csPalette	Состояние соответствует режиму изменения палитры. Это реакция на сообщение **WM_ PALETTCHANGED**
csReadingState	Осуществляется чтение значений свойств из потока (см. табл. 12.2)
csAlignmentNeeded	Осуществляется выравнивание компонента
csFocusing	Элемент управления получает фокус
csCreating	Элемент управления и его дочерние элементы создаются
csPaintCopy	Отрисовывается копия элемента управления
csCustomPaint	Элемент управления выполняет нестандартные операции отрисовки, заданные разработчиком
csDestroyingHandle	Указатель на объект элемента управления уничтожается
csDocking	Элемент управления находится в режиме присоединения

В зависимости от совокупности установленных свойств, элемент управления может соответствовать одному из возможных стилей, который задается свойством

__property TControlStyle ControlStyle.

Доступность элемента управления в целом определяется свойством

__property bool Enabled.

При значении **True** элемент управления полностью работоспособен. При значении **False** элемент управления неактивен и отображается серым цветом.

Для получения контекста устройства элемента управления используется метод

virtual HDC __fastcall GetDeviceContext(HWND &WindowHandle).

Набор свойств и методов класса **TControl** обеспечивает функционирование механизма перетаскивания (Drag-and-Drop) и механизма присоединения (Drag-and-Dock).

Связь компонента с владельцем. Механизм связывания визуального компонента с родительским компонентом (владельцем) позволяет автоматически задавать для нового элемента управления некоторые свойства, отвечающие за его внешний вид (см. выше). В результате все дочерние элементы управления для одного родительского (формы, панели) будут выглядеть одинаково оформленными.

Родительский компонент задается свойством

__property TWinControl* Parent.

Для каждого дочернего элемента управления можно задать значения соответствия родительским нескольких свойств:

__property bool ParentBiDiMode,
__property bool ParentColor,
__property bool ParentFont,
__property bool ParentShowHint.

Каждое из них управляет одной характеристикой визуализации элемента.

В классе **TControl** впервые появляются методы-обработчики событий, которые обеспечивают передачу в элемент действия мыши, присоединение и перетаскивание.

Класс **TControl** редко используется непосредственно, поэтому большинство его свойств объявлены в секции **protected.** Разработчики производных компонентов могут, таким образом, выбирать, какие свойства и события перенести в секцию **public** или **_published**, расширяя тем самым права доступа.

Использование в визуальных компонентах подсказки. Свойства **Hint** и **ShowHint** визуального компонента обеспечивают текст подсказки, появляющийся в ярлычке (всплывающем окне подсказки) или в заданном месте окна, например, в строке состояния. Подсказка появляется в маленькой рамке при остановке мыши на элементе управления. Указанные свойства введены в классе **TControl** и затем наследуются визуальными компонентами.

Расположенное в области видимости **_published** объявление свойства **Hint** имеет вид:

__property System::UnicodeString Hint.

Свойство **ShowHint** объявлено в области видимости **public**

__property bool ShowHint.

В классе **TForm** свойство **ShowHint** переведено в область видимости **_published**.

При значении свойства **ShowHint**, равном **True**, ярлык начинает работать, при значении **False** ярлык выключается.

Использование значений свойств **Hint** и **ShowHint**, а также реализация процедуры появления подсказки происходят во взаимодействии с классом **TApplication**, который выполняет связанные с подсказкой завершающие действия. Поэтому более подробно взаимодействие визуальных компонентов и приложения рассмотрено в параграфе 12.12 «Класс TApplication».

Контрольные вопросы

1. Поясните назначение класса **TControl**.
2. Какие свойства класса **TControl** отвечают за местоположение и размер визуального компонента?
3. Объясните понятие «рабочая область компонента» и назовите свойства класса **TControl**, которые позволяют управлять параметрами рабочей области компонента.
4. Какие свойства класса **TControl** отвечают за внешний вид визуального компонента?
5. Какие свойства класса **TControl** отвечают за связь визуального компонента с родительским компонентом (владельцем)?
6. Какие возможности класс **TControl** унаследовал от класса **TObject**?
7. Какие возможности класс **TControl** унаследовал от класса **TComponent**?

12.7.2. События

Наиболее используемые события класса **TControl** относятся к классу событий мыши и наследуются большинством визуальных компонентов.

Широко используется событие **OnClick**. Обычно оно наступает, если пользователь щелкнул на компоненте, т. е. нажал и отпустил кнопку мыши, когда указатель мыши находился на компоненте. Но это событие происходит также и при некоторых других действиях пользователя. Оно наступает в следующих случаях (особенности перечисленных действий предлагаем освоить самостоятельно):

— пользователь выбрал элемент в таблице, дереве, списке, выпадающем списке, нажав клавишу со стрелкой;

— пользователь нажал клавишу пробела, когда кнопка или индикатор были в фокусе;

— пользователь нажал клавишу Enter, а активная форма имеет кнопку по умолчанию, указанную свойством **Default**;

— пользователь нажал клавишу Esc, а активная форма имеет кнопку прерывания, указанную свойством **Cancel**;

— пользователь нажал клавиши быстрого доступа к кнопке или индикатору; например, если свойство **Caption** индикатора записано как «&Полужирный» и

символ ‘П’ подчеркнут, то нажатие пользователем комбинации клавиш Alt+П вызовет событие **OnClick** в этом индикаторе;

— приложение установило в **True** свойство **Checked** радиокнопки RadioButton;

— приложение изменило свойство **Checked** индикатора CheckBox;

— вызван метод **Click** элемента меню;

— для формы событие **OnClick** наступает, если пользователь щелкнул на пустом месте формы или на недоступном компоненте.

Событие нажатия и отпускания левой кнопки мыши объявлено как свойство

__property System::Classes::TNotifyEvent OnClick.

Описание типа **TNotifyEvent** представляет собой процедурную ссылку

typedef void __fastcall (__closure *TNotifyEvent) (System::TObject* Sender).

Событие нажатия на левую кнопку мыши **OnMouseDown** имеет вид

__property TMouseEvent OnMouseDown,

а событие отпускания левой кнопки мыши задается свойством

__property TMouseEvent OnMouseUp.

В указанных свойствах используется тип **TMouseEvent**, объявленный как

typedef void __fastcall (__closure *TMouseEvent)(System::TObject* Sender, System::Uitypes::TMouseButton Button, System::Classes::TShiftState Shift, int X, int Y).

Событие перемещения мыши привязано к свойству

__property TMouseMoveEvent OnMouseMove,

где тип **TMouseMoveEvent** имеет вид:

typedef void __fastcall(__closure*TMouseMoveEvent) (System::TObject*Sender, System::Classes::TShiftState Shift, int X, int Y).

Событие изменения размера компонента

__property System::Classes::TNotifyEvent OnResize

использует процедурный тип **TNotifyEvent.**

Рассмотрим теперь параметры, используемые в процедурных типах.

Во все обработчики событий, связанных с манипуляциями мыши (как и во все другие обработчики), передается параметр **Sender** типа указателя на **TObject**. Этот параметр содержит указатель на компонент, в котором произошло событие. Он не используется, если обработчик события создается для одного компонента. Однако часто один обработчик применяется для нескольких компонентов. При этом какие-то операции могут быть общими для любых источников события, а какие-то могут требовать индивидуальных действий. В последнем случае параметр **Sender** можно использовать для распознавания источника события. После распознавания источника события необходимо выполнить приведение указателя на **TObject** к нужному компонентному классу.

Операции, связанные с обработкой параметра **Sender**, рассмотрены в параграфе 6.4.

Помимо параметра **Sender** в обработчики событий мыши передаются параметры, позволяющие распознать нажатую кнопку, нажатые при этом вспомогательные клавиши, а также определить координаты курсора мыши. Например, заголовок обработчика события **OnMouseDown** для поля редактирования может иметь следующий вид:

void __fastcall TForm1::Edit1MouseDown (TObject* Sender, TMouseButton Button, TShiftState Shift, int X, int Y).

Помимо уже рассмотренного нами параметра **Sender** в обработчик события передаются параметры **Button, Shift**, **X**, **Y**.

Параметр **Button** типа **TMouseButton** определяет нажатую в этот момент кнопку мыши. Тип **TMouseButton** является перечислимым типом и определяется примерно следующим образом:

enum TMouseButton { mbLeft, mbRight, mbMiddle }.

Значение **mbLeft** соответствует нажатию левой кнопки мыши, значение **mbRight** — правой, а значение **mbMiddle** — средней. Например, если мы хотим, чтобы обработчик реагировал на нажатие только левой кнопки, то должны написать

if (Button != mbLeft) return;.

Теперь, если значение **Button** не равно **mbLeft**, т. е. нажата не левая кнопка, то выполнение обработчика завершится.

Параметр **Shift** типа **TShiftState** определяет, какие вспомогательные клавиши на клавиатуре нажаты в момент нажатия кнопки мыши. Тип **TShiftState** является множественным типом, определенным следующим образом:

typedef System::Set<System_Classes__1, System_Classes__1::ssShift, System_Classes__1::ssHorizontal> TShiftState,

где **System_Classes__1** является перечислимым типом:

enum System_Classes__1 : unsigned char { ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble, ssTouch, ssPen, ssCommand, ssHorizontal }.

Таким образом, значениями множественного типа **TShiftState** являются любые подмножества множнства **{ ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble, ssTouch, ssPen, ssCommand, ssHorizontal }**.

Элементы этого множества соответствуют нажатию клавиш клавиатуры Shift **(ssShift),** Alt **(ssAlt),** Ctrl **(ssCtrl),** а также кнопок мыши: левой (**ssLeft**), правой (**ssRight**), средней (**ssMiddle**). Информация о нажатых кнопках в параметрах **Button** и **Shift** имеет различный смысл. Параметр **Button** соответствует кнопке, нажимаемой в данный момент, а параметр **Shift** содержит информацию о том, какие кнопки были нажаты (и не отпущены), включая те, которые были нажаты ранее. Например, если пользователь нажмет левую кнопку мыши, а затем, не отпуская ее, нажмет правую, то после первого нажатия множество **Shift** будет равно **[ssLeft]**, а после второго нажатия — **[ssLeft, ssRight]**. Если до этого

пользователь нажал и не отпустил какие-то вспомогательные клавиши, то информация о них также будет присутствовать в множестве **Shift**.

Поскольку **Shift** является множеством, проверять наличие в нем тех или иных элементов необходимо методом **Contains**. Например, если мы хотим прореагировать на событие, заключающееся в нажатии левой кнопки мыши при нажатой клавише Alt, то можно использовать оператор

if ((Button == mbLeft) && (Shift.Contains (ssAlt))) ....

В приведенном операторе первое условие **(Button == mbLeft)** можно заменить эквивалентным ему условием, проверяющим ему условием, проверяющим параметр **Shift**:

if ((Shift.Contains (ssLeft)) && (Shift.Contains (ssAlt))) ....

Заметим, что при обратном порядке нажатия клавиш: сначала левой кнопки мыши, а затем — клавиши Alt, приведенные проверки будут не корректны.

Аналогичные параметры **Button** и **Shift** передаются и в обработчик события **OnMouseUp**. Отличие заключается в том, что параметр **Button** хранит информацию об отпускаемой в данный момент кнопке. Параметр **Shift** передается также обработчику события **OnMouseMove**, следовательно, и в этом обработчике можно получить информацию о нажатых клавишах и кнопках.

Во все обработчики событий, связанных с мышью, передаются также координаты курсора **X** и **Y**. Эти параметры определяют координаты курсора в клиентской области компонента. Благодаря этому можно обеспечить различную реакцию в зависимости от того, в какой части клиентской области расположен курсор.

Обилие событий, связанных с мышью, а также фактическое дублирование некоторых из них требуют четкого представления о последовательности отдельных событий, наступающих при том или ином действии пользователя. Для понимания происходящих процессов рассмотри простейшее действие пользователя: переключение с помощью мыши фокуса с одного визуального компонента на другой. Последовательность событий в этом случае приведена в таблице 12.4.

Таблица 12.4

Последовательность событий мыши при переключении фокуса

Номер шага	Действие пользователя	Событие
1	Перемещение курсора мыши в пределах первого компонента	Множество событий **OnMouseMove** в первом компоненте
2	Перемещение курсора мыши в пределах формы	Множество событий **OnMouseMove** в форме
3	Перемещение курсора мыши в пределах второго компонента	Множество событий **OnMouseMove** во втором компоненте
4	Нажатие кнопки мыши в пределах второго компонента	Событие **OnExit** в первом компоненте Событие **OnEnter** во втором компоненте Событие **OnMouseDown** во втором компоненте

Продолжение табл. 12.4

Номер шага	Действие пользователя	Событие
5	Отпускание кнопки мыши в пределах второго компонента	Событие **OnClick** во втором компоненте Событие **OnMouseUp** во втором компоненте

Приведенные в таблице 12.4 события **OnEnter** (попадание в фокус) и **OnExit** (уход из фокуса) принадлежат классу **TWinControl** и рассмотрены в параграфе 12.8.

Как видно из таблицы 12.4, события **OnMouseMove** происходят постоянно в процессе перемещения курсора мыши и даже при его дрожании, неизбежном, если пользователь не снимет руки с мыши. Это надо учитывать и пользоваться этим событием очень осторожно, поскольку оно, в отличие от других, происходит многократно.

Каждое действие пользователя, связанное с нажатием или отпусканием кнопки мыши, приводит к серии последовательно наступающих событий. В обработчиках событий можно распознать, какая кнопка мыши нажата и в какой точке компонента находится в данный момент курсор мыши.

Контрольные вопросы

1. В каких случаях наступает событие **OnClick**?
2. В каких случаях наступает событие **OnMouseDown**?
3. В каких случаях наступает событие **OnMouseUp**?
4. В каких случаях наступает событие **OnMouseMove**?
5. В каких случаях наступает событие **OnResize**?
6. Какое назначение параметров **Sender**, **Button**, **Shift**, **X**, **Y** в обработчиках событий мыши?

12.7.3. Технология перемещения визуальных компонентов

В настоящем параграфе рассмотрим события класса **TControl**, которые реализуют технологию перемещения визуальных компонентов и их элементов.

Технология Drag and Drop (Перетащи и отпусти) позволяет перемещать визуальные компоненты с помощью мыши. Например, в проводнике Windows мы с легкостью можем захватить курсором файл, перетащить его и оставить в каком-либо каталоге. Или, как второй пример, можно рассмотреть панель задач Windows. Мы можем перенести на нее какую-либо иконку, и она останется закреплена на ней. Вот эти действия и представляют собой приложение технологии Drag and Drop. Рассмотрим теперь, как она работает.

Естественно, что почти у всех визуальных компонентов имеются необходимые для реализации рассматриваемой технологии события. Рассмотрим эти события, которые представлены именно в том порядке, в котором они будут генерироваться во время операции перетаскивания.

Событие **OnStartDrag** наступает, когда пользователь нажал левую кнопку мыши над объектом и, не отпуская ее, начал смещать курсор мыши, т. е. начал перетаскивание. Свойство **OnStartDrag** объявлено в классе **TControl**, хра-

нит ссылку на обработчика сообщения о начале операции перетаскивания. Объявление имеет вид:

__property TStartDragEvent OnStartDrag,

где тип **TStartDragEvent** определяет ссылку на обработчик события

typedef void __fastcall (__closure *TStartDragEvent)
(System::TObject* Sender, TDragObject* &DragObject)

(использование ссылок на функцию обработки сообщений рассмотрено в п. 10.4.4).

Обработчик события **OnStartDrag** позволяет описать какие-либо специальные действия, необходимые перед началом перетаскивания. Параметр **Sender** является тем компонентом, который должен перетаскиваться или который содержит объект будущего перетаскивания. Переменная **DragObject** по умолчанию является пустой, т. е. имеет значение **NULL**. Это означает, что переноситься будет сам компонент или его объект. При этом автоматически создается объект типа **TDragControlObject**, с которым среда разработки C++Builder осуществляет весь процесс перетаскивания. Но обработчик может создавать и новый объект перетаскивания, в котором определен какой-нибудь особый вид курсора и т. п.

Событие **OnDragOver** генерируется объектом-приемником. Событие происходит в момент, когда перетаскиваемый объект пересекает границу данного компонента и оказывается внутри его контура. Заканчивается событие, когда объект покидает компонент, пересекая его границу. Свойство **OnDragOver** объявлено в разделе **protected** класса **TControl**

__property TDragOverEvent OnDragOver,

где тип **TDragOverEvent** является процедурной ссылкой на обработчика события

typedef void __fastcall (__closure *TDragOverEvent)
(System::TObject* Sender, System::TObject* Source, int X, int Y, System::Uitypes::TDragState State, bool &Accept).

Обработчик события **OnDragOver** используется для того, чтобы дать сигнал о готовности компонента принять перетаскиваемый объект в том случае, если пользователь отпустит его над данным компонентом. Событие имеет шесть параметров.

Если компонент готов принять объект, то в обработчике надо задать значение параметра **Accept** равным **True**. По умолчанию значение **Accept** равно **True**, так что его можно не задавать. Обработчик может быть и пустым, что будет означать готовность компонента принять любой объект. Обработчик нужен в любом случае (даже пустой), иначе приложение не получит сообщения о приеме компонента.

Во время перетаскивания над компонентом объекта форма курсора мыши может изменяться, сигнализируя пользователю о готовности компонента принять объект. Для обеспечения этого необходимо во время проектирования задать соответствующее значение свойству компонента **DragCursor**. Параметр **Source** обработчика события определяет перетаскиваемый объект, а параметр **Sender** — сам компонент. Параметры **X** и **Y** дают координаты точки экрана

в пикселах. Параметр **State** типа **TDragState** (перечислимый тип данных) определяет состояние перетаскиваемого объекта по отношению к другим объектам. Значения этого параметра приведены в таблице:

Значение	Описание
dsDragEnter	Курсор мыши входит в пределы компонента
dsDragMove	Курсор мыши перемещается в пределах компонента
dsDragLeave	Курсор мыши выходит за пределы компонента

Событие **OnDragDrop** генерирует элемент-приемник, возникает оно при отпускании мыши. Свойство **OnDragDrop** объявлено в разделе **protected** класса **TControl**

__property TDragDropEvent OnDragDrop,

где тип **TDragDropEvent** представляет собой процедурную ссылку на обработчика события

typedef void __fastcall (__closure *TDragDropEvent)(System::TObject* Sender, System::TObject* Source, int X, int Y).

Если параметр **Accept**, полученный в событии **OnDragOver**, имеет значение **True**, то элемент-источник успешно ляжет на элемент-приемник. Событие имеет четыре параметра: **Sender**, **Source**, **X**, **Y** — имеют такое же, как и в событии **OnDragOver**, назначение.

Событие **OnEndDrag** создает объект-источник. Вне зависимости от того, приняла ли переменная **Accept** значение **True** (можно передать элемент) или осталась ли она **False** (запрет на передачу элементов) — сгенерируется это событие.

Свойство **OnEndDrag** объявлено в разделе **protected** класса **TControl**

__property TEndDragEvent OnEndDrag,

где тип **TEndDragEvent** является процедурной ссылкой на обработчика события

typedef void __fastcall (__closure *TEndDragEvent) (System::TObject* Sender, System::TObject* Target, int X, int Y).

Параметры **Sender**, **Target**, **X**, **Y** полностью аналогичны описанным выше. В данном случае параметр **Target** — объект-приемник.

Рассмотрим теперь пример. На экранной форме имеется два списка класса **TListBox**. Нашей задачей является перенос строк из списка 1 в список 2. Экранная форма задачи представлена на рисунке 12.8.

Мы не будем рассматривать действия по построению экранной формы и расположенных на ней списков. Отметим только один важный момент: настройку свойств списков, имеющих отношение к нашей задаче. Во-первых, свойство **DragKind** имеет два значения: **dkDrag** и **dkLock**. Первое значение разрешает использование технологии Drag and Drop, а второе — запрещает. Необходимо для каждого списка установить значение **dkDrag** свойству **DragKind**. Во-вторых, свойство **DragMode**, имеющее два значения (**dmManual**

или **dmAutomatic**), необходимо установить в **dmManual** для каждого списка. Значение **dmAutomatic** позволяет перетаскивать компонент пользователю. Значение **dmManual** разрешает перетаскивать компонент только после выполнения метода **BeginDrag**. В нашем случае передача строк из первого списка во второй будет выполняться в ручном режиме.

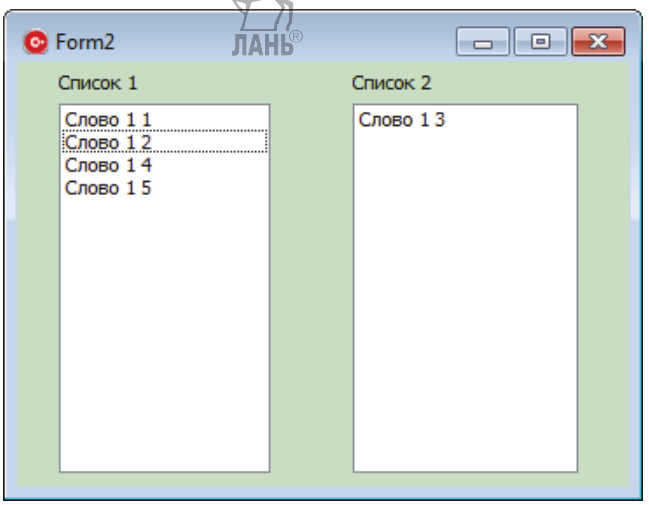


Рис. 12.8

Экранная форма демонстрации перемещения строк из списка 1 в список 2

Рассмотрим теперь реализацию обработчиков событий, которая позволит решить поставленную задачу.

Для того чтобы в ручном режиме начать перетаскивание, следует вызвать метод **BeginDrag** данного элемента управления. В классе **TControl** (раздел **public**) описание метода **BeginDrag** имеет вид:

void __fastcall BeginDrag(bool Immediate, int Threshold=0xffffffff).

Если установить **Immediate** (непосредственный) в **True**, перетаскивание начинается немедленно. В противном случае перетаскивание не начнется до тех пор, пока пользователь не перетащит мышь на указанное в **Threshold** (порог) количество пикселей. Если **Threshold** равен –1, используется значение по умолчанию.

Инициализацию события **OnStartDrag** для первого списка выполним в обработчике события движения мыши:

```
void _fastcall TForm2::ListBox1MouseMove(TObject *Sender, TShiftState Shift, int
X, int Y)
{
    if ((ListBox1–>ItemIndex != -1) && Shift.Contains(ssLeft))
      ListBox1–>BeginDrag(False,8);
}.
```

В приведенном коде выполняется, если нажата левая кнопка мыши (ssLeft) и выделена строка (текущий индекс строки не равен –1), то вызывается событие начала операции перетаскивания и вызывается обработчик события **OnStartDrag** для первого списка:

```
void __fastcall TForm2::ListBox1StartDrag(TObject *Sender, TDragObject
*&DragObject)
{
  intFItemIndex=ListBox1–>ItemIndex;
},
```

где целочисленная переменная intFItemIndex должна быть объявлена в разделе **private** класса **TForm2**, в ней запоминаем номер выделенной строки первого списка.

Следующим обработчиком будет обработчик события **OnDragOver** во втором списке, который используется для того, чтобы дать сигнал о готовности компонента принять перетаскиваемый объект в том случае, если пользователь отпустит его над данным компонентом.

```
void __fastcall TForm2::ListBox2DragOver(TObject *Sender, TObject *Source,
                                         int X, int Y, TDragState State, bool &Accept)
{
     Accept=(intVItemIndex != -1) && (Sender != Source);
}.
```

Далее реализуем обработчик события **OnDragDrop** для второго списка:

```
void __fastcall TForm2::ListBox2DragDrop(TObject *Sender, TObject *Source, int
X, int Y)
{
        String s;
        s=ListBox1–>Items–>Strings[intVItemIndex];
        ListBox2–>Items–>Add(s);
        ListBox1–>Items–>Delete(intFItemIndex);
},
```

в котором при отпускании мыши над вторым списком происходит перенос выделенной строки первого списка во второй список.

Наконец, необходимо реализовать обработчик события **OnEndDrag** и тем самым завершить операцию Drag and Drop:

```
void __fastcall TForm2::ListBox1EndDrag(TObject *Sender, TObject *Target, int X, int Y)
{
    intFItemIndex=-1;
}.
```

Примечание. Обработчики событий приведены в том порядке, в котором возникают соответствующие события.

Контрольные вопросы

1. Когда происходит событие **OnStartDrag**?
2. Когда происходит событие **OnDragOver**?
3. Когда происходит событие **OnDragDrop**?
4. Когда происходит событие **OnEndDrag**?
5. Какое назначение параметров обработчиков событий технологии Drag and Drop?

12.8. Класс TWinControl

Класс **TWinControl** обеспечивает создание и использование компонентов, которые относятся к группе оконных элементов управления. Оконными элементами являются формы, диалоги, а также большинство стандартных элементов управления, таких как кнопки, поля редактирования, средства изменения состояния. Оконные элементы управления обеспечивают взаимодействие программы с пользователем и операционной системой: прием и передачу фокуса, отклик на действия мышью и ввод с клавиатуры и т. д.

Класс **TWinControl** инкапсулирует оконные элементы управления с дескрипторами. Некоторые производные от **TWinControl** компоненты (**TEdit**, **TListBox** и **TComboBox**) инкапсулируют стандартные элементы управления Windows — поля редактирования, простые и комбинированные списки и т. д. Поэтому нам не придется манипулировать с ними посредством стандартных функций Windows API, а пользоваться свойствами и методами, предоставляемыми самими компонентами.

12.8.1. Свойства и методы

Производные от **TWinControl** компоненты обладают тремя основными характеристиками: они имеют оконные дескрипторы, способны принимать фокус ввода и могут являться родителями других элементов управления. Поэтому многие свойства **TWinControl** предназначены для изменения фокуса, обслуживания событий клавиатуры и отображения потомков компоненты:

Доступ к оконному элементу управления. Каждый оконный элемент управления имеет системный дескриптор окна, который представляет собой некоторую константу целого типа индекс в массиве ссылок на структуру окна (см. п. 5.1).

Значение кода оконного элемента управления программисту недоступно, однако имеется возможность получить и использовать системный дескриптор окна с помощью свойства

__property HWND Handle,

где тип **HWND** обеспечивает косвенный доступ к содержимому окна.

Для создания дескриптора окна для элемента управления используется метод

virtual void __fastcall CreateHandle(void).

При создании оконного элемента управления вызывается метод

virtual void __fastcall CreateParams(TCreateParams &Params),

который заполняет структуру **Params** необходимыми значениями. Описание структуры **TCreateParams** имеет вид:

```
struct DECLSPEC_DRECORD TCreateParams
{
    public:
                System::WideChar *Caption;
                unsigned Style;
                unsigned ExStyle;
```

```
            int X;
            int Y;
        int Width;
            int Height;
            HWND WndParent;
            void *Param;
            tagWNDCLASSW WindowClass;
            System::StaticArray<System::WideChar, 64> WinClassName;
};.
```

Операционная система создает дескриптор окна только вместе с самим окном. Поэтому метод **CreateHandle** только создает окно, а для присваивания свойству **Handle** значения дескриптора окна вызывает метод **virtual void fastcall CreateWnd(void)**.

Рис. 12.9 (начало)
Виды рамки и обрезов рамки визуального компонента

Рис. 12.9 (продолжение)
Виды рамки и обрезов рамки визуального компонента

Рис. 12.9 (окончание)
Виды рамки и обрезов рамки визуального компонента

Управление фокусом. Фокусом называют состояние активности окна. В каждый момент времени только одно окно может находиться в фокусе. Для передачи фокуса между элементами управления на одной форме часто используется клавиша <Tab>. Порядок перемещения фокуса между элементами определяется свойством

__property TTabOrder TabOrder;

где тип **TTabOrder** определен как целое число.

В первую очередь фокус передается компоненту с минимальным значением свойства **TabOrder**. Далее — по возрастанию значения. При переносе компонента на форму это значение задается автоматически в соответствии с числом компонентов на форме.

Компонент можно заставить не откликаться на клавишу <Tab>. Для этого свойству

__property bool TabStop

необходимо присвоить значение **False**.

Для передачи фокуса выбранному элементу управления применяется метод

virtual void __fastcall SetFocus(void).

Чтобы узнать, имеет ли элемент управления фокус в настоящее время, используется метод

bool __fastcall Focused(void).

Обрамление оконного элемента управления. Все оконные элементы имеют рамку по контуру (впрочем, она может быть не видна). Ее параметры задаются группой свойств.

Первое свойство

__property TBorderWidth BorderWidth

определяет ширину рамки. Тип **TBorderWidth** является синонимом целого типа данных.

Рамка формируется тремя зонами: поле рамки, внешней откос (bevel outer) и внутренней откос (bevel inner).

Следующие два свойства:

__property TBevelCut BevelInner;
__property TBevelCut BevelOuter;

определяют, соответственно, параметры внутренней и внешней зон рамки, где тип **TBevelCut** имеет вид:

enum TBevelCut { bvNone, bvLowered, bvRaised, bvSpace }.

Толщина откоса задается свойством

__property TBevelWidth BevelWidth,

где тип **TBevelWidth** является синонимом целого типа данных.

Результаты

__property TBevelKind BevelKind;
enum TBevelKind { bkNone, bkTile, bkSoft, bkFlat }

определяет стиль рамки;

__property TBevelWidth BevelWidth;

(тип **TBevelWidth** определен как **typedef int TBevelWidth**) задает размер рамки.

При знакомстве с перечисленными свойствами мы имеем типичную ситуацию: «лучше один раз увидеть», поэтому на рисунке 12.8 приведены результаты эксперимента со свойствами рамки компонента **TPanel** при значениях **BorderWidth**=15, **BevelWidth**=10. Значения параметров **BevelInner**, **BevelOuter** и **BorderStyle** приведены на каждом изображении визуального компонента **TPanel**.

Свойство

__property TBrush Brush

определяет параметры кисти (цвет и заполнение), которой рисуется фон элемента.

Элемент управления может содержать другие элементы управления. Для организации их применяется индексированный список указателей, представляющих свойство

__property TControl Controls[Index: Integer];.

Общее число дочерних элементов управления содержится в свойстве

__property Integer ControlCount;.

Внешний вид оконного элемента определяется свойством

__property Boolean Ctl3D;.

При значении **True** элемент управления имеет трехмерный вид. Иначе элемент выглядит плоским.

Для вызова темы контекстной помощи для конкретного элемента управления предназначено свойство

__property THelpContext HelpContext;,

где **THelpContext** является синонимом типа данных **lnteger**. Значение свойства должно соответствовать номеру темы в файле помощи с расширением **.hlp**. Это свойство позволяет организовать контекстный поиск в справочной службе для отдельных компонент.

В классе **TWinControl** добавлена возможность использования редакторов способа ввода (Input Method Editor, IME). Такие редакторы позволяют приспособить стандартную раскладку клавиатуры для символьных языков для ввода нестандартных символов (иероглифов и т. д.). Редакторы IME представляют собой специально устанавливаемое в операционной системе программное обеспечение. Имя такого редактора задается в свойстве **ImeName**. Режим работы редактора определяется свойством **ImeMode**.

Контрольные вопросы

1. Поясните назначение класса **TWinControl**.
2. Объясните назначение понятия «дескриптор окна».
3. Почему дескриптор окна доступен программисту только косвенно?
4. Какие свойства и методы класса **TWinControl** отвечают за внешний вид визуального компонента?
5. Какие свойства класса **TControl** отвечают за связь визуального компонента с родительским компонентом (владельцем)?
6. Какие возможности класс **TWinControl** унаследовал от класса **TObject**?
7. Какие возможности класс **TWinControl** унаследовал от класса **TComponent**?
8. Какие возможности класс **TWinControl** унаследовал от класса **TControl**?

12.8.2. События

В класс **TWinControl** добавлено несколько методов-обработчиков событий, обеспечивающих реакцию на ввод с клавиатуры, получение и потерю фокуса. **TWinControl** имеет события, вызываемые взаимодействием с клавиатурой, изменением фокуса.

Событие **OnKeyDown** наступает при нажатии пользователем любой клавиши. Можно распознать нажатые клавиши, включая функциональные (и кнопки мыши), но нельзя распознать символ нажатой клавиши[46].

Свойство **OnKeyDown** объявлено в разделе **protected** класса **TWinControl**

__property TKeyEvent OnKeyDown.

[46] Причины невозможности распознавания символа рассмотрены далее.

Событие **OnKeyUp** наступает при отпускании пользователем любой клавиши. Можно распознать нажатые клавиши, включая функциональные и кнопки мыши, но нельзя распознать символ отпускаемой клавиши.

Свойство **OnKeyUp** объявлено в разделе **protected** класса **TWinControl**

__property TKeyEvent OnKeyUp.

В свойствах **OnKeyDown OnKeyUp** используется процедурный тип, который имеет объявление

typedef void __fastcall (__closure *TKeyEvent)(System::TObject*Sender, System::Word &Key,
System::Classes::TShiftState Shift).

Рассмотрим параметры процедурного типа **TKeyEvent**.

Параметр **Sender** — компонент, в котором произошло событие (см. п. 12.7.2).

Параметр **Key** определяет нажатую в момент события клавишу клавиатуры. Для не алфавитно-цифровых клавиш используются виртуальные коды API Windows. Приведем несколько строк таблицы виртуальных кодов, соответствующих наиболее распространенным клавишам (табл. 12.5).

Таблица 12.5

Некоторые виртуальные коды

Клавиша	Десятичное число	Шестнадцатеричное число	Символическое имя	Сравнение по символу
F1	112	0×70	**VK_F1**	
Enter	13	0×0D	**VK_RETURN**	
Shift	16	0×10	**VK_SHIFT**	
Ctrl	17	0×11	**VK_CONTROL**	
Alt	18	0×12	**VK_MENU**	
Esc	27	0×1B	**VK_ESCAPE**	
0)	48	0×30		**0**
1 !	49	0×31		**1**
n N т Т	78	0×4E		**N**
y Y н Н	89	0×59		**Y**

Параметр **Key** является целым числом, определяющим клавишу, а не символ. Например, один и тот же код соответствует прописным и строчным символам как латиницы, так и кириллицы «Y», «y», «Н», «н». Аналогично (см. табл. 12.5) символам «0», «)» соответствует один код клавиши. Распознать символ с помощью параметра **Key** невозможно.

Проверять нажатую клавишу можно, сравнивая параметр **Key** с целым десятичным числом, приведенным во втором столбце таблицы 12.5. Например, реакцию на нажатие пользователем клавиши Enter можно оформить оператором: **if (Key == 13) ...;**.

Можно сравнивать **Key** и с шестнадцатеричным представлением кода, приведенным в третьем столбце таблицы 12.5. Например, приведенный выше оператор можно записать в виде: **if (Key == 0×0D) ...;**.

Для клавиш, которым не соответствуют символы, введены также именованные константы, которые облегчают написание программы, поскольку не требуют помнить числовые значения кодов клавиш. Приведенный выше пример запишется так: **if (Key == VK_RETURN) ...;**.

Для символьных и цифровых клавиш можно производить проверку, сравнивая с десятичным или шестнадцатеричным кодом, но это не очень удобно, так как трудно помнить коды различных символов. Другой путь — воспользоваться тем, что коды латинских символов в верхнем регистре совпадают с виртуальными кодами, используемыми в параметре **Key**. Поэтому, например, если мы хотим распознать клавишу, соответствующую символу «Y», то можно записать: **if (Key == 'Y') ...;**.

Поскольку при работе с параметром **Key** мы можем использовать только латинские символы в верхнем регистре, необходимо помнить, что если мы захотим записать латинский символ «y» или соответствующие этой клавиши символы кириллицы «Н» или «н», то оператор не сработает. Однако оператор **if (Key == 'Y') ...** будет действовать на все символы, относящиеся к указанной клавише: «Y», «y», «Н», «н». Иногда это удобно, а иногда нет. Например, если мы задаем пользователю какой-то вопрос, на который он должен ответить положительно (**Y**) или отрицательно, то подобный оператор избавляет пользователя от необходимости следить, в каком регистре он вводит символ и какой язык — английский или русский — включен в данный момент. Ему достаточно при положительном ответе просто нажать клавишу с символом «Y». Однако если пользователь привык к символам кириллицы, то могут возникнуть неприятности, так как клавиша «Н» может ассоциироваться с отрицательным ответом.

В завершение рассмотрения параметров процедурного типа **TKeyEvent** приведем пример распознавания комбинации клавиш. Пусть мы хотим распознать комбинацию клавиш Alt+X. Для этого можно воспользоваться оператором

if ((Key == 'X') && Shift.Contains(ssAlt)) ...;.

В этом примере используется параметр **Shift**, хранящий сведения об одновременно нажатых клавишах. Параметр **Shift** мы рассмотрели в контексте обработки событий мыши в п. 12.7.2.

Переходим к рассмотрению события **OnKeyPress**.

Событие **OnKeyPress** наступает при нажатии пользователем клавиши символа. Можно распознать только нажатую клавишу символа, различить символ в верхнем и нижнем регистре, различить символы кириллицы и латинские, но нельзя распознать функциональные клавиши и кнопки.

Свойство **OnKeyPress** объявлено в разделе **protected** класса **TWinControl**

__property TKeyPressEvent OnKeyPress,

где описание типа **TKeyPressEvent** имеет вид

typedef void __fastcall (__closure *TKeyPressEvent)(System::TObject*Sender, System::WideChar&Key).

В процедурном типе имеется параметр **Key**, определяющий нажатую клавишу символа. Но обратим внимание, что тип этого параметра не целое число, как в предыдущих случаях, а символ. В данном случае в обработчик передается не виртуальный код клавиши, а символ, по которому можно определить, какая это буква: прописная или строчная, русская или латинская.

Пусть, например, мы задали пользователю вопрос, на который он должен ответить символами «Д» или «д» (да) или символами «Н» или «н» (нет). Тогда можно распознать положительный ответ в обработчике события **OnKeyPress** оператором: **if ((Key == ‘Д’) || (Key == ‘д’)) ...**.

Приведенный оператор реагирует только на положительный ответ пользователя и не реагирует, соответственно, на отрицательный или ошибочный ответ. Реакцию на все возможные ответы обеспечивает оператор **switch**:

```
switch (Key)
{
     case ‘Д’:
     case ‘д’: ...; break;
     case ‘Н’:
     case ‘н’: ...; break;
    default: Beep();
}.
```

Здесь предусмотрена реакция на положительный и отрицательный ответы, а также звуковой сигнал при ошибочном ответе.

Обратим внимание на приведенный ранее тип обработчика: параметр **Key** в нем передается по ссылке. Это позволяет в обработчике изменять значение **Key**, изменяя, соответственно, его стандартную обработку в компоненте, поскольку наш обработчик события срабатывает раньше стандартного обработчика компонента. Пусть, например, мы хотим иметь на форме окно редактирования класса **TEdit**, в котором пользователь должен вводить только целые числа без знака. В этои случае мы можем обеспечить безошибочный ввод, подменяя все недопустимые символы нулевым с помощью следующего кода:

```
Set<Char,0,255> Dig;
if (!(Dig << ‘0’ << ‘1’ << ‘2’ << ‘3’ << ‘4’ << ‘5’ << ‘6’ << ‘7’ << ‘8’ << ‘9’ <<
‘\b’).Contains(Key))
{ Key=0; Beep();}.
```

В приведенном примере формируется множество допустимых символов

Dig={‘0’, ‘1’,‘2’,‘3’,‘4’,‘5’,‘6’,‘7’,‘8’,‘9’,‘\b’},

и выполняется операция проверки непринадлежности символа **Key** множеству **Dig**.

Символ **BackSpace** (**‘\b’**) добавлен в число допустимых, чтобы упростить пользователю редактирование вводимого текста.

При нажатии пользователем любой клавиши, кроме клавиш с цифрами и клавиши **BackSpace**, символы подменяются нулевым символом и просто не появляются в окне редактирования. При нажатии пользователем ошибочной клавиши вызываемая функция **Beep()** воспроизводит звуковой сигнал.

Событие **OnEnter** наступает в момент получения визуальным компонентом фокуса в результате манипуляции мышью, нажатия клавиши табуляции или программной передачи фокуса.

Свойство **OnEnter** объявлено в разделе **protected** класса **TWinControl**

__property System::Classes::TNotifyEvent OnEnter.

Событие **OnExit** наступает в момент потери визуальным компонентом фокуса в результате манипуляции мышью, нажатия клавиши табуляции или программной передачи фокуса.

__property System::Classes::TNotifyEvent OnExit.

Контрольные вопросы

1. Когда наступает событие **OnKeyDown**?
2. Когда наступает событие **OnKeyUp**?
3. Какую информацию передают параметры процедурного типа **TKeyEvent**: **Sender**, **Key**, **Shift**?
4. Когда наступает событие **OnKeyPress**?
5. Какую информацию передают параметры процедурного типа **TKeyPressEvent**: **Sender**, **Key**?

12.9. Класс TCustomControl

Стандартные компоненты, как производные от **TWinControl** (например, **TEdit** и **TListbox**), уже имеют способности собственного отображения, предоставленные инкапсулированными элементами управления Windows. А как создать оконную компоненту, которая отображает себя в виде, соответствующем оригинальным требованиям пользователя? Решение именно этой задачи и обеспечивает **TCustomControl**.

Будучи производным от класса **TWinControl**, **TCustomControl** является оконным элементом управления и, следовательно, может принять фокус ввода. Разработанные компоненты могут быть произведены от **TCustomControl**. Класс **TCustomControl** содержит свойство **Canvas** (Канва), которое предоставляет возможность произвольного рисования на выбранной прямоугольной области.

__property TCanvas Canvas;.

По существу, производные от **TCustomControl** компоненты предоставляют в наше распоряжение виртуальный метод

virtual void __fastcall Paint(void);,

переопределение которого позволит рисовать компоненты так, как мы пожелаем.

Класс **TCustomControl** предназначен для создания на его основе нестандартных оконных элементов управления.

Возможности этого класса унаследовали классы **TPanel**, **TGroupBox**, **TStringGrid** и т. д.

Контрольные вопросы

1. Каково назначение класса **TCustomControl**?
2. Какие возможности класс **TCustomControl** унаследовал от класса **TObject**?
3. Какие возможности класс **TCustomControl** унаследовал от класса **TComponent**?
4. Какие возможности класс **TCustomControl** унаследовал от класса **TControl**?
5. Какие возможности класс **TCustomControl** унаследовал от класса **TWinControl**?

12.10. Класс TGraphicControl

Класс **TGraphicControl** предназначен для создания на его основе визуальных компонентов, не получающих фокус в процессе выполнения приложения. Так как непосредственным предком класса является класс **TControl**, то потомки **TGraphicControl** умеют реагировать на управляющие воздействия мышью.

Производные от **TGraphicControl** классы используются в тех ситуациях, когда мы хотим изобразить на форме текст или графику, не обращаясь к функциональным возможностям обычных оконных элементов управления. Отметим следующие достоинства такого подхода. Во-первых, **TGraphicControl** не пользуется системными ресурсами Windows, так как не требует оконного дескриптора. Во-вторых, метод рисования **TGraphicControl** исполняются немного быстрее за счет того, что перерисовка компоненты не связана с диспетчеризацией сообщений Windows, а реализуется процессом рисования, заложенным в родителе данного элемента.

Производные **TGraphicControl** имеют обработчики событий, вызываемые манипуляциями с мышью.

Наглядный пример элемента управления, которому не нужно получать фокус, — это компонент **TLabel**, предназначенный для отображения текста, или компонент **TImage**, предназначенный для визуализации изображений.

Для визуализации элементов управления на основе этого класса используется канва, инкапсулированная в классе **TCanvas**.

Доступ к канве осуществляется через свойство

__property Vcl::Graphics::TCanvas* Canvas.

Отрисовка элемента управления осуществляется методом

void __fastcall WMPaint(Winapi::Messages::TWMPaint &Message);

после получения сообщения **WM_PAINT**.

Контрольные вопросы

1. Каково назначение класса **TGraphicControl**?
2. Какие возможности класс **TGraphicControl** унаследовал от класса **TObject**?

3. Какие возможности класс **TGraphicControl** унаследовал от класса **TComponent**?

4. Какие возможности класс **TGraphicControl** унаследовал от класса **TControl**?

12.11. Класс TApplication

В каждом приложении операционной системы Windows автоматически создается объект **Application** класса **TApplication** — приложение. Этот компонент отсутствует в палитре библиотеки, вероятно, только потому, что он всегда один в приложении. Объект **Application** создается автоматически при запуске приложения и управляет происходящими процессами: создание и уничтожение объектов, обработка сообщений, обслуживание исключительных ситуаций.

Объект **Application** отвечает за создание компонентов:

— экранные формы класса **TForm**;

— дисплей **Screen** класса **TScreen**;

— принтер **Printer** класса **TPrinter**;

— интерфейс обмена данными с другими приложениями **Clipboard** класса **TClipboard**;

— информационные структуры инициализации элементов приложения **IniFile** класса **TIniFile** и др.

Все создаваемые объекты являются глобальными, и на их использование рассчитаны многие методы других компонентов.

Класс **TApplication** находится на ветви иерархии наследования компонентов **TComponent**, **TPersistent**, **TObject**, поэтому свойства и методы перечисленных компонентов становятся доступными в объекте **Application**. В настоящем параграфе мы рассмотрим только собственные свойства и методы класса **TApplication**.

К объекту **Application** нужно обращаться в двух случаях:

1. Получение информации о приложении.

2. Управление процессом выполнения приложения и его формами.

12.11.1. Свойства и методы

Рассмотрим свойства, с помощью которых можно получить информацию о приложении.

Свойство

__property System::UnicodeString Title

хранит имя приложения, которое по умолчанию совпадает с именем проекта. Имя приложения можно определить на странице «Application setting» (доступ через Proect | Option). Значение свойства можно изменять в процессе исполнения приложения.

Свойство

__property System::UnicodeString ExeName

содержит имя выполняемого файла с полным путем к нему. Это свойство удобно использовать, чтобы определить каталог, из которого запущено приложение

и который может содержать другие файлы (настройки, документы, базы данных и др.), связанные с приложением. Вызов функции **ExtractFilePath(Application–>ExeName)** возвращает этот каталог. Обычно свойство **ExeName** тождественно результату вызова функции **ParamStr(0)**, возвращающей нулевой параметр командной строки — имя файла с путем.

Булево свойство

__property bool Active

характеризует активность приложения. Оно истинно, если форма приложения находится в фокусе.

Свойство

__property TForm* MainForm

определяет главную форму приложения.

Булево свойство

__property bool ShowMainForm

определяет, должна ли главная форма быть видимой в момент запуска приложения на выполнение. По умолчанию оно равно истине, что обеспечивает видимость главной формы в момент начала работы приложения.

Свойство

__property System::UnicodeString HelpFile

указывает файл справки, который используется в приложении. По умолчанию оно равно опции Help File, задаваемой во время проектирования на странице Application окна опций проекта (меню Project | Option). Свойство можно изменять программно, назначая в зависимости от режима работы приложения тот или иной файл справки.

Свойство

__property System::UnicodeString Hint

объявлено в области видимости **public**, содержит текст подсказки того визуального компонента или раздела меню приложения, над которым в данный момент перемещается курсор мыши. Смена этого свойства происходит в момент события **OnHint**. Во время перемещения курсора над поверхностью визуального компонента текст **Hint** подсказки указанного визуального компонента переносится в свойство **Hint** объекта **Application**. Таким образом, свойство **Hint** объекта **Application** можно использовать для отображения текста подсказки в полях, характеризующих текущий режим приложения.

В общем случае значение свойства **Hint** состоит из двух частей, разделенных символом вертикальной черты «|». Левая часть значения отображается в ярлычке, если пользователь задержит курсор мыши над данным компонентом (это может быть любой компонент, включая разделы меню). Обычно краткое пояснение компонента. Например, такой подсказкой, как правило, снабжаются быстрые кнопки типа **TSpeedButton**. Правая часть значения свойства **Hint** содержит текст, отображаемый в какой-то выделенной для этого части окна, например, в строке состояния. Это обычно развернутое пояснение. Например,

свойство **Hint** для быстрой кнопки доступа к разделу меню сохранения файла может иметь вид: «Сохранить|Сохранение текущего документа в файле». Как частный случай, в свойстве **Hint** может быть задана только первая часть подсказки без символа '|'.

Свойство

__property bool ShowHint

объявлено в области видимости **public**. Свойство **ShowHint** определяет режимы показа подсказки. По умолчанию значение свойства равно **True**.

Для того, чтобы левая часть подсказки появлялась в ярлычке, когда пользователь задержит курсор мыши над данным компонентом, нужно выполнить следующие действия:

1. Указать тексты свойства **Hint** для всех компонентов, которые мы хотим снабдить подсказкой.

2. Установить свойства **ShowHint** (показать подсказку) этих компонентов в **True** или установить в **True** свойство **ParentShowHint** (**ShowHint** родителя) и установить в **True** свойство **ShowHint** контейнера, содержащего данные компоненты, или формы.

Естественно, что мы можем устанавливать значения свойства в **True** или **False** программно, включая и отключая подсказки в различных режимах работы приложения. Если значение свойства **ShowHint** равно **True**, то ярлычок будет всплывать. Если мы не задали значение свойства компонента **Hint**, но установили в **True** свойство **ShowHint** или установили в **True** свойство **ParentShowHint**, а в родительском компоненте задали **ShowHint** равным **True**, то в ярлычке будет отображаться текст **Hint** из родительского компонента. Правда, все описанное выше справедливо при значении свойства **ShowHint** приложения **Application,** равном **True**. Если установить в **Application** значение свойства **ShowHint** в **False**, то ярлычки не будут появляться независимо от значений **ShowHint** в любых компонентах.

Для того чтобы вторая часть сообщения, записанного в **Hint**, отображалась в строке состояния в моменты, когда курсор мыши проходит над компонентом, надо использовать обработку события **OnHint**. Это событие именно приложения **Application**, а не того компонента, над которым проходит курсор мыши. Если обработчик этого события определен, то в момент прохождения курсора над компонентом, в котором задано свойство **Hint**, вторая часть сообщения компонента заносится в свойство **Hint** объекта Application. Если свойство **Hint** компонента содержит только первую часть, то в свойство **Hint** объекта **Application** заносится эта первая часть. Причем все это делается независимо от состояния свойства компонента **ShowHint**. Чтобы отображать эти сообщения в строке сотояния, надо определить и написать обработчик события приложения **OnHint**, как показано в приведенном ниже примере. Третий способ использования свойства **Hint** компонента заключается в непосредственном отображении текста, заключенного в нем сообщения в какой-то метке или панели с помощью функций **GetShortHint** и **GetLongHint**, первая из которых возвращает первую часть сообщения, а вторая — вторую (если второй части нет, то возвращается

первая часть). Например, обработка события **OnHint** компонента **Edit1** может иметь вид:

```
void __fastcall TForm2::ApplicationEvents1Hint(TObject *Sender)
{
        Edit2–>Text=GetLongHint(Application–>Hint);
}.
```

В результате в поле **Edit2–>Text** будет появляться вторая часть подсказки **Edit1–>Hint** «Поле редактирования | Поле для редактирования шифра детали», заданной в свойстве компонента **Edit1**.

Теперь остановимся на некоторых методах объекта **Application**. Методы

void __fastcall Initialize(void) — инициализация приложения,
void __fastcall Run(void) — запуск цикла обработки сообщений,

включаются в проект автоматически.

Вызов методов **Initialize** и **Run** выполняется только один раз, и место вызова можно увидеть в главной функции приложения. Там же осуществляется вызов метода

void __fastcall CreateForm(Classes::TComponentClass InstanceClass, void *Reference)

для всех автоматически создаваемых экранных форм приложения. Первый параметр, **InstanceClass**, определяет класс создаваемой формы, а **Reference** — ссылка на создаваемый объект. Например,

Application–>CreateForm(__classid(TForm2), &Form2),

где **__classid** — операция образования имени класса.

Метод

void __fastcall Terminate(void)

завершает выполнение приложения. Если приложение завершается при активности главной формы, то вместо метода **Application–>Terminate()** можно использовать метод

void __fastcall Close(void),

который наследуется классом **TForm** от класса **TCustomForm**.

Метод

void __fastcall Minimize(void)

сворачивает приложение, помещая его пиктограмму в полосу задач Windows.

В классе **TApplication** имеется очень полезный метод

int __fastcall MessageBox(const System::WideChar * Text, const System::WideChar * Caption, int Flags=0x0)[47]**.**

Метод **MessageBox** отображает диалоговое окно с заданными кнопками, сообщением и заголовком и позволяет проанализировать ответ пользователя.

[47] Одноименная функци API рассмотрена в сокращенном виде в параграфе 5.3.

Эта функция инкапсулирует одноименную функцию API Windows. Рассмотрим метод **MessageBox** подробно.

Параметр **Text** представляет собой выводимый в окно текст сообщения, параметр **Caption** задает текст заголовка окна. Имеется отличие в использовании этих параметров: значение параметра **Text** может переноситься на следующую строку, а значение параметра **Caption** не переносится и может привести к появлению очень широкого окна.

Параметр **Flags** может иметь несколько значений, каждое из которых определяет вид и поведение диалогового окна. Этот параметр может комбинироваться операцией сложения по одному флагу из следующих групп.

Флаги кнопок, отображаемых в диалоговом окне

Флаг	Значение (в скобках указаны надписи в русифицированных версиях Windows)
MB_ABORTRETRYIGNORE	Кнопки Abort (Стоп), Retry (Повтор) и Ignore (Пропустить)
MB_OK	Кнопка OK. Этот флаг принят по умолчанию
MB_OKCANCEL	Кнопки OK и Cancel (Отмена)
MB_RETRYCANCEL	Кнопки Retry (Повтор) и Cancel (Отмена)
MB_YESNO	Кнопки Yes (Да) и No (Нет)
MB_YESNOCANCEL	Кнопки Yes (Да), No (Нет) и Cancel (Отмена)

Флаги пиктограмм в диалоговом окне

Флаг	Пиктограмма
MB_ICONEXCLAMATION, MB_ICONWARNING	Восклицательный знак (замечание, предупреждение)
MB_ICONINFORMATION, MB_ICONASTERISK	Буква i в круге (подтверждение)
MB_ICONQUESTION	Знак вопроса (ожидание ответа)
MB_ICONSTOP, MB_ICONERROR, MB_ICONHAND	Знак креста на красном круге (запрет, ошибка)

Флаги, указывающие кнопку по умолчанию (которая в первый момент находится в фокусе)

Флаг	Кнопка
MB_DEFBUTTON1	Первая кнопка. Это принято по умолчанию
MB_DEFBUTTON2	Вторая кнопка
MB_DEFBUTTON3	Третья кнопка
MB_DEFBUTTON4	Четвертая кнопка

Флаги модальности

Флаг	Пояснение
MB_APPLMODAL	Пользователь должен ответить на вопрос, прежде чем сможет продолжить работу с приложением. Но он может перейти в окна другого приложения. Он может также работать со всплывающими окнами данного приложения. Этот флаг принят по умолчанию

Продолжение табл.

Флаг	Пояснение
MB_SYSTEMMODAL	То же самое, что MB_APPLMODAL, но окно диалога всегда отображается поверх других окон, даже если пользователь перешел к другим приложениям. Используется для предупреждения о серьезных ошибках, требующих немедленного вмешательства

Дополнительные флаги (могут задаваться оба флага)

Флаг	Пояснение
MB_HELP	Добавляет в окно кнопку Help (Справка), щелчок на которой или нажатие клавиши F1 генерирует событие Help
MB_TOPMOST	Помещает окно всегда сверху

Возможны еще некоторые флаги, определяющие характер поведения окна при работе в сети нескольких пользователей, позволяющие отображать тексты справа налево (для восточных языков) и т. п.

Функция возвращает нуль, если не хватает памяти для создания диалогового окна. Если функция выполнена успешно, то возвращаемая величина свидетельствует о следующем:

Значение	Численное значение	Пояснение
IDOK	1	Выбрана кнопка OK
IDCANCEL	2	Выбрана кнопка Cancel (Отмена) или нажата клавиша Esc
IDABORT	3	Выбрана кнопка Abort (Стоп)
IDRETRY	4	Выбрана кнопка Retry (Повтор)
IDIGNORE	5	Выбрана кнопка Igmore (Пропустить)
IDYES	6	Выбрана кнопка Yes (Да)
IDNO	7	Выбрана кнопка No (Нет)

Рассмотрим пример. Ниже приведен текст выхода из диалога ввода данных, предусматривающий проверку введенных данных. В случае правильности введенных данных осуществляется выход с сохранением. При неправильно введенных данных происходит выход без сохранения данных, возможен также возврат к диалогу ввода данных для их редактирования.

```
 int Res;
 if (введенные данные верны)
{
Res=Application–>MessageBoxW(String(«Сохранить текущую запись?»).w_str(),
                             String(«Введенные данные верны!»).w_str(),
                             MB_YESNO+MB_ICONQUESTION);
        switch (Res)
             {
               case (IDYES): { выход из диалога с сохранением данных};
               case (IDNO)  : { выход из диалога без сохранения данных};
             }
 }
  else
```

```
  {
    Res=Application–>MessageBoxW(String(«Сохранить текущую за-
пись?»).w_str(),
                                 String(«Введенные данные содержат ошиб-
ку!»).w_str(), MB_YESNOCANCEL+MB_ICONQUESTION);
          switch (x)
              {
                case (IDYES): {выход из диалога с сохранением данных};
                case (IDNO)  : {выход из диалога без сохранения данных};
                case (IDCANCEL) : {возврат к диалогу ввода данных};
   }
 }.
```

Отображаемые этим кодом диалоговые окна приведены на рисунках 12.10, 12.11.

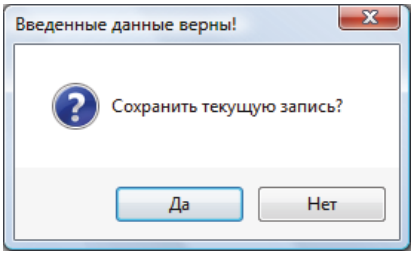


Рис. 12.10

Вид окна диалога с пользователем при правильно введенных данных

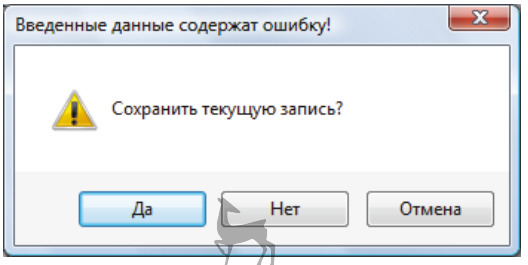


Рис. 12.11

Вид окна диалога с пользователем при неправильно введенных данных

Контрольные вопросы

1. Когда создается объект **Application**?
2. Назовите предков класса **TApplication**.
3. Какая технология используется для вывода подсказки компонента?
4. Какими приемами можно завершить приложение?
5. Какие свойства предназначены для получения информации о приложении?
6. Сформулируйте назначение метода **MessageBox**.
7. Какие возможности класс **TApplication** унаследовал от класса **TObject**?
8. Какие возможности класс **TApplication** унаследовал от класса **TComponent**?

12.11.2. Класс TApplicationEvents

В классе TApplication определено множество событий, которые очень полезны для управления приложением. Начиная с C++Builder 5, в библиотеку визуальных компонентов включен невизуальный компонент **TApplicationEvents**, существенно облегчивший задачу обработки событий. Этот компонент перехватывает события объекта **Application** и, следовательно, обработчики этих событий теперь можно писать как обработчики событий невизуального компонента **TApplicationEvents**. На каждую форму приложения можно поместить компонент класса **TApplicationEvents**. События объекта **Application** будут передаваться каждому такому компоненту. Если мы хотим, чтобы событие передавалось, прежде всего, какому-то одному из них, то нужно применить к нему метод **Activate**, который поставит его в начало очереди компонентов класса **TApplicationEvents**. Если же мы при этом не хотим, чтобы компоненты **TApplicationEvents** других экранных форм получали события, то можно применить к привилегированному компоненту метод **CancelDispatch**. Тогда после обработки события в данном компоненте другие компоненты **TApplicationEvents** вообще не будут реагировать на эти события.

Во многие обработчики событий компонента **TApplicationEvents** передается по ссылке параметр **Handled**. По умолчанию его значение равно **False**. Если мы обработали соответствующее событие и не хотим, чтобы оно далее обрабатывалось другими компонентами **TApplicationEvents**, надо в обработчике установить значение свойства **Handled** равным **True**. Если же мы оставим **Handled** равным **False**, то событие будут пытаться обрабатывать другие компоненты **TApplicationEvents** (если они есть). Если событие так и останется необработанным, то его будет пытаться обработать активный компонент, а если и он не обработает, то — активная форма. Предотвратить обработку события другими компонентами можно, используя описанный ранее метод **CancelDispatch**.

Все события класса **TApplicationEvents** объявлены в области видимости **published**.

Событие **OnActivate** происходит в момент, когда приложение становится активным. Это происходит в начале выполнения и в случаях, когда пользователь, перейдя к другим приложениям, вернулся в данное приложение. Событие **OnActivate** объявлено как свойство:

```
__property System::Classes::TNotifyEvent OnActivate.
```

Если это событие обработано в экземпляре класса **TApplicationEvents**, то предотвратить дальнейшую его обработку можно методом **CancelDispatch**.

Событие **OnDeactivate** происходит в момент, когда пользователь переключается на другое приложение. Событие **OnDeactivate** объявлено как свойство:

```
__property System::Classes::TNotifyEvent OnDeactivate.
```

Если это событие обработано в **ApplicationEvents**, то предотвратить дальнейшую его обработку можно методом **CancelDispatch**.

Событие **OnException** задано свойством

__property TExceptionEvent OnException,

описание типа **TExceptionEvent** имеет вид:

typedef void __fastcall (__closure *TExceptionEvent)
(System::TObject* Sender, System::Sysutils::Exception* E).

Событие **OnException** возникает, когда в приложении сгенерировано исключение, которое нигде не перехвачено. В обработчик события передается параметр **Sender** — источник исключения, и параметр **E** типа **Exception** — объект исключения. Параметр **E** помогает определить тип исключения. Например, проверка

if (E–>ClassNameIs(«EZeroDivide»))

позволяет узнать, явилось ли деление на ноль причиной исключительной ситуации.

В обработчике события **OnException** можно предусмотреть нестандартную обработку исключений на уровне приложения, например, русифицировать стандартные сообщения об исключениях и дать пользователю какие-то рекомендации. При этом нужно иметь в виду, что введение собственного обработчика события **OnException** отключит стандартную реакцию приложения на исключительные ситуации.

Событие **OnHint** возникает в момент, когда курсор мыши начинает перемещаться над компонентом или разделом меню, в котором определено свойство **Hint**. Объявление обработчика события имеет вид

__property System::Classes::TNotifyEvent OnHint.

Технология обработки события **OnHint** рассмотрена при описании свойства **Hint** в предыдущем параграфе.

Событие **OnMessage** возникает, когда приложение получает сообщение Windows (но не переданное функцией SendMessage API Windows). Обработчик события имеет описание

__property TMessageEvent OnMessage,

где тип **TMessageEvent** задан определением

typedef void __fastcall (__closure *TMessageEvent)(tagMSG &Msg, bool &Handled).

В обработчике события **OnMessage** можно предусмотреть нестандартную (отличную от определенной в классе **TApplication**) обработку сообщения. В обработчик события передается параметр **Msg** — полученное сообщение и по ссылке передается параметр **Handled**. Не нужно забывать, что в Windows передаются тысячи сообщений в секунду, так что обработчик **OnMessage** может серьезно снизить производительность приложения.

События **OnMinimize** и **OnRestore** возникают, соответственно, при сворачивании и восстановлении ранее свернутого приложения. Описание обработчиков событий имеет вид:

__property System::Classes::TNotifyEvent OnMinimize.

Событие **OnShowHint** возникает, когда приложение собирается отобразить ярлычок с текстом подсказки **Hint**. Описание обработчика события имеет вид:

__property TShowHintEvent OnShowHint.

Здесь используется определение типа

typedef void __fastcall (__closure *TShowHintEvent)(System::UnicodeString &HintStr,
bool &CanShow, Vcl::Controls::THintInfo &HintInfo).

В обработчик события передается по ссылке параметр **HintStr** — первая часть свойства **Hint** компонента, ярлычок которого должен отображаться. В обработчике этот текст можно изменить. Также по ссылке передается параметр **CanShow**. Если в обработчике установить его равным **False**, то ярлычок отображаться не будет. Третий параметр, передаваемый по ссылке, — **Hintlnfo**. Это структура, поля которой содержат информацию о ярлычке: его координаты, цвет, задержки появления и т. п. В частности, имеется поле **HintControl** — компонент, сообщение которого должен отображаться в ярлычке, и поле **HintStr** — отображаемое сообщение. По умолчанию **Hintlnfo**. **HintStr** — первая часть свойства **Hint** компонента. Но в обработчике это значение можно изменить.

Контрольные вопросы

1. С какой целью введен класс **TApplicationEvents**?
2. Какое назначение параметра **Handled** в обработчиках событий класса **TApplicationEvents**?
3. В какой области видимости объявлены события класса **TApplicationEvents**?
4. Когда возникает событие **OnActivate**?
5. Когда возникает событие **OnDeactivate**?
6. Когда возникает событие **OnException**?
7. Когда возникает событие **OnHint**?
8. Когда возникает событие **OnMessage**?
9. Когда возникает событие **OnMinimize**?
10. Когда возникает событие **OnRestore**?
11. Когда возникает событие **OnShowHint**?

12.12. Класс TScreen

Компонент **TScreen** представляет, как можно понять из названия, поведение экрана, или, иначе, рабочего стола компьютера. Поскольку экземпляр данного класса только один (он создается системой при запуске приложения), то большинство методов и свойств имеют информационный характер и недоступны для записи.

Класс **TScreen** наследуется непосредственно от **TComponent**. Это означает, что он является хранилищем компонентов. Естественно, что такими ком-

понентами являются экранные формы класса **TForm**. Необходимо помнить, что мы никогда, ни в каком случае не должны сами создавать объект класса **TScreen**. Для доступа к свойствам и методам класса в каждом VCL-приложении есть глобальная переменная **Screen**. Аналогично, мы никогда не должны разрушать глобальный объект **Screen** — он автоматически будет разрушен при завершении работы приложения.

Большинство свойств класса **TScreen** являются свойствами только для чтения. Класс **TScreen** представляет состояние экрана, на котором выполняется приложение и предоставляет разработчику информацию:

— о размерах и разрешении экрана;
— о формах и модулях данных инстанцированных приложением;
— об активной форме и об активном элементе управления на ней;
— о курсорах и шрифтах, доступных приложению;
— о мониторах, на которых может быть отображена форма приложения, включая их размеры.

Мы можем в любом приложении использовать, например, такие свойства объекта **Screen**, как **int Height** — высота экрана, и **int Width** — ширина экрана. Это, в частности, может потребоваться, если мы задаем значение свойства форм такими, что размеры форм остаются постоянными. Например, если мы в процессе проектирования используем один тип монитора, а приложение в дальнейшем может работать с мониторами другого типа, то не исключено, что наша форма не поместится на экране или, наоборот, будет очень маленькой. Чтобы избежать таких неприятностей, можно автоматически масштабировать нашу форму, введя, например, в обработчик события **OnCreate** следующие строки:

```
Width=Screen–>Width/2;
Height=Screen–>Height/2;.
```

Этот код задает размеры экранной формы, равными половине соответствующих размеров экрана.

Разрешающую способность экрана можно узнать, воспользовавшись свойством **int PixelsPerInch**, указывающим количество пикселов экрана на дюйм в вертикальном направлении. Это справедливо именно для вертикального направления, поскольку во многих мониторах коэффициенты масштабирования по горизонтали и вертикали различаются.

Объект **Screen** имеет свойство **TForm*Forms [int Index]**, содержащее список форм, отображаемых в данный момент на экране, и свойство **int FormCount**, отражающее количество таких форм. Можно использовать эти свойства, например, для того, чтобы гарантировать, что на данном типе монитора размеры ни одной формы не превысят размеров экрана. Соответствующий код может выглядеть так:

```
for ( int i=0; i < Screen–>FormCount; i++)
                {
                    if (Screen–>Forms[i]–>Height > Screen–>Height)
                                Screen–>Forms[i]–>Height=Screen–>Height – 100;
                    if (Screen–>Forms[i]–>Width > Screen–>Width)
```

```
                Screen–>Forms[i]–>Width=Screen–>Width – 100;
}.
```

Размеры форм, превышающие размер экрана, урезаются этим кодом с запасом в 100 пикселов.

В приведенных примерах надо, конечно, предусмотреть, чтобы при изменении размеров формы адекватно изменялись размеры и расположение компонентов на ее поверхности.

Так же, как к формам, можно получить доступ и к модулям данных.

Свойство **System::Classes::TDataModule* DataModules [int Index]** содержит список всех существующих на текущий момент модулей данных приложения, а свойство **int DataModuleCount** указывает число таких модулей. Эти параметры можно использовать в совокупности для поиска по всем модулям данных приложения. Данную возможность предлагается изучить самостоятельно.

Свойство **Controls::TWinControl*ActiveControl** позволяет определить, какой оконный элемент управления в активной форме получает в настоящий момент фокус ввода. Данное свойство необходимо рассматривать вместе с событием **Classes::TNotifyEvent OnActiveControlChange**. Данное событие происходит, когда фокус ввода передается другому элементу управления. То есть если нам необходимо отследить переход фокуса с одного элемента управления на другой, то необходимо написать обработчик события **OnActiveControlChange.** Это реализуется следующим образом. В заголовочном файле в секцию **private** класса формы необходимо добавить объявление функции, которая будет обработчиком события **OnActiveControlChange**

```
void fastcall ActiveControlChange(TObject *Sender).
```

В обработчике события **OnCreate** формы мы назначаем данную функцию обработчиком события **OnActiveControlChange:**

```
Screen–>OnActiveControlChange=ActiveControlChange;.
```

Сама функция **ActiveControlChange** реализуется следующим образом:

```
void __fastcall TMainForm::ActiveControlChange(TObject *Sender)
{
    TWinControl *WinControl=Screen–>ActiveControl;
    if (WinControl)  Caption=WinControl–>Name;
}.
```

Подобным же образом осуществляется работа со свойством **ActiveForm** и событием **OnActiveFormChange**. Свойство **TForm*ActiveForm** определяет, какая из форм приложения в настоящий момент активна, т. е. имеет фокус. Если само приложение в настоящий момент неактивно, то значение свойства определяет, какая форма станет активной при активации приложения. Обратим внимание, что **ActiveForm —** свойство только для чтения, и для того, чтобы сделать форму активной, можно воспользоваться методом **SetFocus**. При переходе фокуса с одной формы на другую возникает событие **OnActiveFormChange.**

Еще одно полезное свойство объекта **Screen** — это **System::Classes::TStrings*Fonts** (шрифты). Это свойство содержит список шрифтов, доступных на данном компьютере (свойство только для чтения). Его можно использовать в приложении, чтобы проверять, имеется ли на компьютере тот или иной шрифт, используемый в приложении. Если нет, то можно или дать пользователю соответствующее предупреждение, или сменить шрифт в приложении на один из доступных, или дать пользователю возможность самому выбрать соответствующий шрифт. Например, можно поместить в приложении компонент списка **TComboBox** и при событии формы **OnCreate** загрузить его доступными шрифтами с помощью операторов:

```
ComboBox1–>Items=Screen–>Fonts;
ComboBox1–>ItemIndex=0;.
```

Тогда в нужный момент пользователь может выбрать подходящий шрифт из списка, а для того, чтобы этот шрифт использовался, например, для текста в компоненте **RichEdit1**, в обработчик события **OnClick** или **OnChange** списка необходимо вставить следующие операторы:

```
RichEdit1–>SelAttributes–>Name = ComboBox1–>Items–>Strings[ComboBox1–
>ItemIndex];
RichEdit1–>SetFocus();
```

Если мы хотим использовать выбранный шрифт для всех компонентов формы, в которых свойство **ParentFont** установлено в **True**, то оператор должен иметь вид:

```
Font–>Name=ComboBox1–>Items–>Strings[ComboBox1–>ItemIndex];
```

Свойство **Cursor** объекта **Screen** определяет вид курсора. Если это свойство равно **crDefault**, то вид курсора при перемещении над компонентами определяется установленными в них свойствами **Cursor**. Но если свойство **Cursor** объекта **Screen** отлично от **crDefault**, то соответствующие свойства компонентов отменяются и курсор имеет глобальный вид, заданный в **Screen**. Для иллюстрации проведем эксперимент:

```
Screen–>Cursor=crDefault;    // Screen–>Cursor=crArrow;
Edit1–>Cursor =crNone;       // Курсор отсутствует
Memo1–>Cursor =crSizeNS;     // Стрелка вверх-вниз
Button1–>Cursor=crCross;     // Крест
Panel1–>Cursor =crIBeam;     // Вертикальная линия
Edit2–>Cursor  =crSizeNESW;  // Наклонная стрелка вверх-вниз.
```

В результате исполнения рассмотренного фрагмента кода курсор при попадании на поверхность компонентов **Edit1**, **Memo1**, **Button1**, **Panel1**, **Edit2** будет менять вид. Если первую строку примера заменим на приведенный в комментарии оператор **Screen–>Cursor=crArrow;**, то вид курсора в тех же ситуациях изменяться не будет.

Необходимо помнить, что любое изменение вила курсора должно выполняться только на время решения нашей задачи. Затем вид курсора нужно восстановить. Например, изменение курсора на форме «песочные часы» во

время выполнения каких-то длинных операций можно оформить следующим образом:

```
TCursor Save_Cursor=Screen–>Cursor;
Screen–>Cursor=crHourGlass;
try
  {
     // выполнение требуемых длинных операций
   }
catch (...)
  {
     Screen–>Cursor=Save_Cursor; // восстановление курсора
     throw;
  }
Screen–>Cursor=Save_Cursor;
```

или

```
TCursor Save_Cursor=Screen–>Cursor;
Screen–>Cursor=crHourGlass;
try
  {
     // выполнение требуемых длинных операций
   }
__finally
  {
     Screen–>Cursor=Save_Cursor; // восстановление курсора
  }.
```

При успешном или аварийном окончании длинных операций курсор в любом случае возвращается в значение по умолчанию.

Контрольные вопросы

1. Каково назначение класса **TScreen**?
2. Почему создается только один объект класса **TScreen**?
3. Какие возможности класс **TScreen** унаследовал от класса **TObject**?
4. Какие возможности класс **TScreen** унаследовал от класса **TComponent**?

12.13. Класс TClipboard

12.13.1. Общее описание

Операции «Копировать», «Вставить», «Вырезать» есть практически в любом полноценном приложении. Эти операции осуществляются с помощью буфера обмена.

Буфер обмена является ресурсом коллективного использования, поэтому необходимы средства управления использованием памяти буфера обмена. Буфер обмена представляет собой область памяти, в которую приложение может поместить информацию и из которой может забрать информацию (рис. 12.12).

Возникает вопрос: как разобраться в том, что было записано в буфер обмена? Для этого нужны некоторые договоренности.

Сначала проведем аналогию с файловым обменом. Здесь мы сталкиваемся с теми же проблемами.

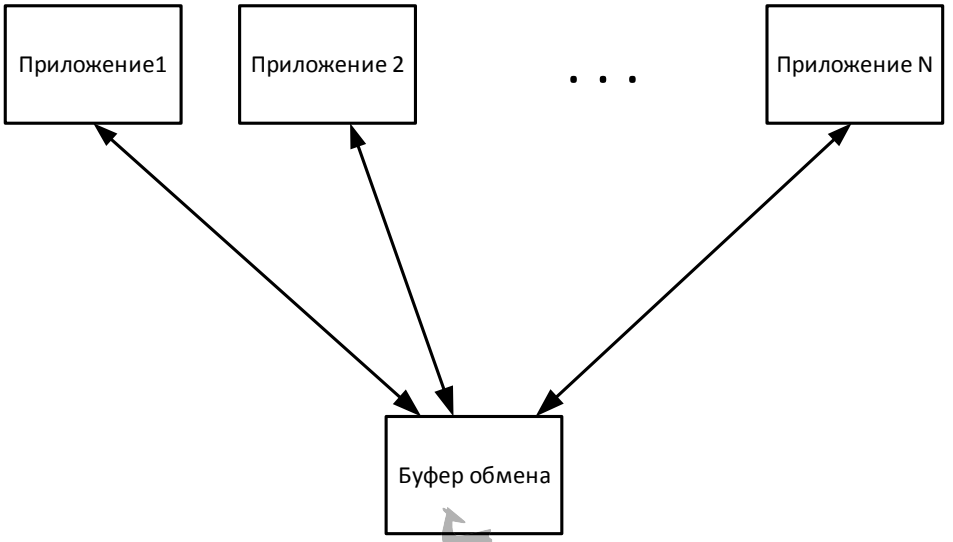


Рис. 12.12
Взаимодействие приложений с буфером обмена

Во-первых, мы имеем два типа файлов: текстовые и двоичные. Для использования файлов нам достаточно ввести переменную вида **FILE*F**. Затем после открытия файла мы можем использовать операции ввода-вывода **fscanf**, **fprintf**. В этом случае структура файла становится известной: текстовый файл. При использовании двоичных файлов нам необходимо использовать операции **fread**, **fwrite**. Таким образом, структура (формат) файла закладывается используемыми операциями.

Вернемся к буферу обмена. Наша задача состоит в том, чтобы реализовать подобную идею использования форматов данных для правильного понимания содержимого буфера обмена.

Операционная система Windows поддерживает достаточно большое количество стандартных форматов данных для работы с буфером обмена. Приведем в качестве примера названия некоторых из них:

— **CF_TEXT** — текстовый формат, устроен точно так же, как и в текстовых файлах;

— **CF_BITMAP** — формат для обмена данными растровой графики;

— **CF_KOMPONENT** — формат для обмена компонентами;

— **CF_PALETTE —** формат для обмена данными о палитре и т. д.

Естественно, что с каждым названием связано встроенное в систему программирования описание формата.

Прежде чем приступить к подробному описанию методов работы с буфером обмена, рассмотрим одну интересную возможность: отложенная запись данных в буфер обмена.

Чтобы понять, что такое отложенная запись, разберемся вначале с обычной записью данных.

Вопрос: где хранятся данные, записанные в буфере обмена? Очевидно, в памяти. Но в какой? Эта память должна быть доступна для всех приложений. Кроме того, содержимое памяти не должно исчезать при завершении работы отдельных приложений. Например, мы можем запустить Paintbrush, скопировать из него фрагмент изображения в буфер обмена, затем завершить работу приложения Paintbrush и запустить Word for Windows. Несмотря на то, что приложение Paintbrush не запущено, буфер обмена по-прежнему содержит фрагмент изображения, который можно вставить в документ, редактируемый с помощью Word for Windows.

Всем перечисленным выше требованиям удовлетворяет глобальная память, принадлежащая операционной системе Windows. Такая память должна имеет атрибут **GMEM_DDESHARE**.

Следовательно, когда приложение записывает данные в буфер обмена, размер оперативной памяти, доступной другим приложениям, уменьшается. Причем уменьшается достаточно сильно, если выполняется запись графического изображения, так как данные сохраняются одновременно в нескольких форматах. Для каждого формата изображению может потребоваться несколько десятков мегабайт памяти.

Когда какое-либо приложение вставляет данные из буфера обмена, оно использует только один из имеющихся там форматов. Остальные форматы данных могут так и не потребоваться.

Механизм отложенной записи данных предназначен для экономии оперативной памяти. В момент, когда пользователь копирует данные в буфер обмена, при отложенной записи никакой передачи данных не происходит, хотя буфер обмена отмечается как содержащий эти данные в одном или нескольких форматах. Когда какому-либо приложению потребуются данные из буфера обмена, приложение, записавшее в буфер обмена эти данные, получает сообщение о необходимости выполнения фактической записи данных, что и осуществляется.

Таким образом, данные сохраняются в глобальной памяти с атрибутом **GMEM_DDESHARE** только тогда, когда в этом появилась необходимость, и только в том формате, который нужен приложению, читающему содержимое буфера обмена. Если приложение, имеющее отложенную запись, завершает свою работу, оно должно записать данные в буфер обмена во всех возможных форматах, иначе данные будут утеряны.

Для работы с буфером обмена в C++ Builder существует класс **TClipboard**. Иерархия наследования имеет вид **TClipboard→TPersistent→TObject**. В программе автоматически создается экземпляр **Clipboard** класса **TClipboard**, поэтому этот объект можно использовать без предварительной инициализации, достаточно подключить модуль Clipbrd.hpp. Создавать потомков класса **TClipboard** не имеет смысла, так как к существующим форматам данных мы всегда сможем добавить свой отличный от существующих формат данных.

Теперь для работы с выбранным форматом данных нужно указывать имя этого формата данных. К счастью, для стандартных форматов данных эта работа уже автоматизирована. Дело в том, что все визуальные компоненты уже имеют в своем составе специальные методы для работы с буфером обмена. При введении собственных форматов данных указание имени формата обязательно. Используемые при этом приемы рассмотрены в настоящем параграфе.

У класса **TClipboard** имеется ряд свойств и методов, которые можно использовать в своих приложениях. Рассмотрим приемы использования этих методов и свойств.

Как уже говорилось, в программе автоматически создается объект **Clipboard**. В ранних версиях системы программирования вся работа с буфером обмена осуществлялась через имя **Clipboard**. В последних версиях ситуация изменилась. Для обращения к буферу обмена введена функция

extern DELPHI_PACKAGE TClipboard* __fastcall Clipboard(void),

которая возвращает адрес буфера обмена. По мнению автора, функция **Clipboard()** введена в соответствии с принципами инкапсуляции с целью обеспечения дополнительного контроля корректности использования буфера обмена. При этом имя объекта **Clipboard** стало недоступным.

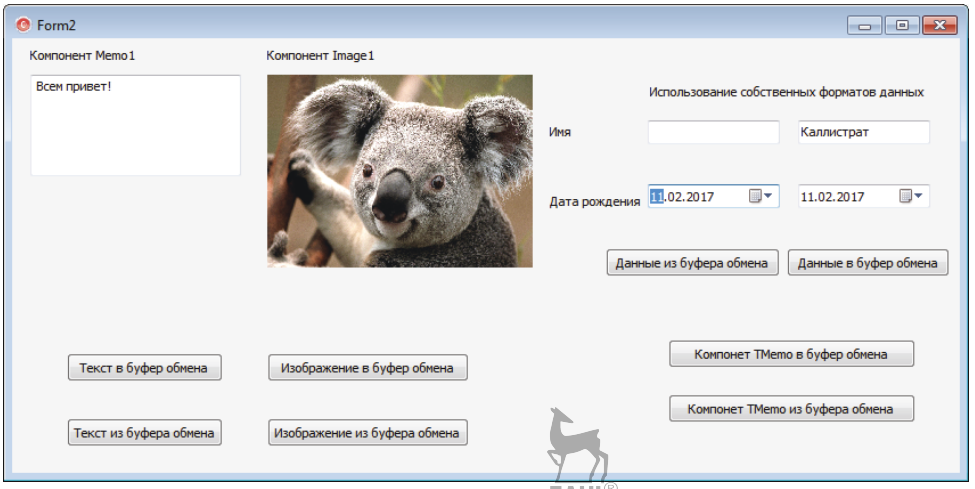


Рис. 12.13

Экранная форма для проведения экспериментов с буфером обмена

Для хранения форматов данных в буфере обмена предусмотрено векторное свойство

__property System::Word Formats[int Index].

Для изучения операций с буфером обмена разработано приложение, вид экранной формы которого приведен на рисунке 12.13. В нижней части экранной формы расположены четыре пары кнопок, с помощью которых будем вызывать операции буфера обмена.

Детали реализации приложения для изучения операций с буфером обмена (рис. 12.13) рассмотрены в последующих параграфах. Поэтому при чтении п. 12.12.2, 12.12.3, 12.12.4, 12.12.5 будет полезно возвращаться к рисунку 12.13.

Контрольные вопросы

1. Что называется форматом буфера обмена?
2. Какой атрибут имеет память для хранения данных в буфере обмена?
3. В чем заключается прием отложенной записи?
4. Как создается объект **Clipboard**?
5. Какое свойство предназначено для хранения описаний форматов данных?
6. Какие возможности класс **TClipboard** наследовал от класса **TObject**?

12.13.2. Обработка текстовой информации

Обработка текстовой информации может осуществляться несколькими приемами.

В основе первого приема лежит использование свойства **__property System::UnicodeString AsText**, которое моделирует содержание текстовой информации буфера обмена.

Для занесения в буфер обмена текстовой информации необходимо выполнить присваивание вида **Clipboard()–>AsText=«Всем привет!»**. Занесем текст из из поля класса **TMemo** в буфер обмена

Clipboard()–>AsText=Memo1–>Text.

Корректное копирование текстовой информации из буфера обмена выполняется следующим образом:

```
if (Clipboard()–>HasFormat(CF_TEXT))
{
   Memo1–>Text=Clipboard()–>AsText;
}.
```

Вначале мы проверяем, записана ли в буфер обмена информация в текстовом формате. Для этого используется метод класса **TClipboard bool __fastcall HasFormat(System::Word Format)**, который возвращает истину, если информация в буфере обмена соответствует параметру функции.

Затем, при выполнении условия проверки, из буфера обмена извлекается текст в поле **Memo1**.

В классе **TClipboard** определены два метода:

void __fastcall SetTextBuf(System::WideChar * Buffer) и
int __fastcall GetTextBuf(System::WideChar * Buffer, int BufSize).

Разместим в буфере обмена текстовую информацию:

Clipboard()–>SetTextBuf(L»Всем привет!»),

затем извлечем текст из буфера обмена в поле класса **TMemo**:

```
if (Clipboard()–>HasFormat(CF_TEXT))
{
```

```
int intVCB_Size=Clipboard()–>AsText.Length()+1;
String * Buffer=new String(intVCB_Size);
Clipboard()–>GetTextBuf(Buffer–>w_str(),intVCB_Size);
Memo1–>Text=Buffer–>w_str();
delete Buffer;  Buffer=NULL;
}.
```

Текст из поля класса **TMemo** можно поместить в буфер обмена следующим образом:

Clipboard()–>SetTextBuf(Memo1–>Lines–>Strings[0].w_str()).

Контрольные вопросы

1. Какое свойство класса **TClipboard** предназначено для обмена текстовыми данными с буфером обмена?
2. Какие методы предназначены для обмена текстовыми данными с буфером обмена?
3. Какую проверку нужно выполнять для корректного извлечения текстовой информации из буфера обмена?
4. Какие средства имеются у текстовых компонентов для обмена данными с буфером обмена?

12.13.3. Обработка графической информации

Обработка графической информации осуществляется с помощью метода класса **TClipboard**

virtual void __fastcall Assign(System::Classes::TPersistent* Source).

Метод осуществляет копирование содержимого одного объекта (**Source**) в другой: объект, вызвавший метод **Assign.** Например,

```
Clipboard()–>Assign(Image1–>Picture);
Image1–>Picture–>Assign(Clipboard()).
```

Метод **Assign** унаследован от класса **TPersistent** и переопределен для обработки графической информации.

Контрольные вопросы

1. Какой метод предусмотрен в классе **TClipboard** для обработки графической информации?

12.13.4. Копирование компонента в буфер обмена

С помощью буфера обмена можно также копировать компоненты. Для этой цели в классе **TClipboard** предусмотрены методы

void __fastcall SetComponent(System::Classes::TComponent* Component) и System::Classes::TComponent* __fastcall GetComponent(System: :Classes::TComponent*Owner, System::Classes::TComponent*Parent)

Отметим, что при копировании компонентов необходимо создать и зарегистрировать метакласс (класс классов).

Пример копирования и восстановления компонента Memo1 приведен ниже.

```
void __fastcall TForm2::btnFMemo1ToClipboardClick(TObject *Sender)
{
   TMetaClass * MetaClass;
   MetaClass=__classid(TMemo);
   RegisterClasses(&MetaClass,0);
   Clipboard()–>SetComponent(Memo1);
}
void __fastcall TForm2::btnFTMemoFromClipboardClick(TObject *Sender)
{
  TMetaClass * MetaClass;
  MetaClass=__classid(TMemo);
  RegisterClasses(&MetaClass,0);
  if (Clipboard()–>HasFormat(CF_COMPONENT))
  {
        Clipboard()–>GetComponent(Owner,this);
        Memo1–>Parent=this;
        Memo1–>Top=10;
        Memo1–>Left=250;
  }
}.
```

Проведенные эксперименты с копированием компонента TMemo показали, что на экранной форме создается новая копия объекта **Memo1**, а старая — становится недоступной.

Контрольные вопросы

1. Какие методы предусмотрены в классе **TClipboard** для обмена компонентами?

12.13.5. Копирование произвольных данных в буфер обмена

Рассмотренные примеры показывают, что каждый из классов **TMemo**, **TImage** и т. п. знает, как поместить свои данные в буфер обмена. Указанные знания базируются на форматах данных, соответственно, **CF_TEXT**, **CF_BITMAP** и др. Для копирования произвольных данных в буфер обмена нам понадобится создание нового формата. Работу с новым форматом данных мы осуществим следующей последовательностью действий:

а) создадим специальный класс для обмена информацией с буфером обмена;

б) выделим область памяти для хранения информации о формате обмена;

в) сообщим буферу обмена название выделенной области памяти для хранения формата (зарегистрируем формат);

г) обмен данными с буфером обмена придется осуществлять самостоятельно, так как интеллекта буфера обмена не хватает для автоматизации этого процесса.

Рассмотрим перечисленные действия на примере. Пусть данные для обмена содержат сведения о персональных данных, оформленные в виде струк-

туры «Персональные данные» с полями: имя персоны и дата рождения персоны:

```
struct TPersonData
{
    String strFName;
    TDateTime dtFBirthDate;
}.
```

Введем специальный класс для обмена данными с буфером обмена:

```
class TClipboardExchange
{
  public:
              LPCTSTR strFFormatName;
              Word wrdFFormatIndex;
              TPersonData * pdFPersonData;
              void ToClipboard();
              void FromClipboard();
},
```

где **strFFormatName** — имя нашего формата, **wrdFFormatIndex** — в это поле Clipboard запишет место нашего формата в векторном свойстве **Clipboard–> Formats**, **pdFPersonData** — имя структуры «Персональные данные», **ToClipboard()** и **FromClipboard()** — имена методов для записи персональных данных в буфер обмена и чтения их из буфера обмена.

Рассмотрим теперь реализацию методов класса **TClipboardExchange**:

```
void TClipboardExchange::ToClipboard()
{
  //очищаем буфер обмена
  EmptyClipboard();
  //выделяем память под область буфера обмена
  HGLOBAL hPersonData= GlobalAlloc(GMEM_DDESHARE, sizeof(TPersonData));
  // получаем адрес первого байта области буфера обмена и блокируем эту область
  // от обращения других программ
  TPersonData * PersonData=(TPersonData*) GlobalLock(hPersonData);
  //записываем данные в выделенную память
  PersonData–>strFName=ClipboardExchange–>pdFPersonData–>strFName;
  PersonData–>dtFBirthDate=ClipboardExchange–>pdFPersonData–>dtFBirthDate;
  //помещаем данные в буфер обмена
  SetClipboardData(ClipboardExchange–>wrdFFormatIndex, hPersonData);
  // разблокируем область буфера обмена
  GlobalUnlock(hPersonData);
}
void TClipboardExchange::FromClipboard()
{
 // извлекаем данные из буфера, получаем индекс таблицы информации буфера
 // обмена о блоках памяти
  HANDLE hClipboardData=
                        GetClipboardData(ClipboardExchange–>wrdFFormatIndex);
```

```
 // получаем указатель на блок памяти и блокируем эту область от обращения
 // других программ
  TPersonData* PersonData=(TPersonData*)GlobalLock( hClipboardData );
   //забираем данные из блока буфера обмена
  ClipboardExchange–>pdFPersonData–>strFName=PersonData–>strFName;
  ClipboardExchange–>pdFPersonData–>dtFBirthDate=PersonData–>dtFBirthDate;
   // разблокируем блок памяти буфера обмена
   GlobalUnlock( hClipboardData );
}.
```

Наконец, рассмотрим методы обработки сообщений о нажатии кнопок «Данные в буфер обмена» и «Данные из буфера обмена» (рис. 12.13).

```
void __fastcall TForm2::btnFDataToClipboardClick(TObject *Sender)
{
 if (OpenClipboard(this–>Handle))
  {
   ClipboardExchange–>pdFPersonData–>strFName=edtFNameToClipboard–>Text;
   ClipboardExchange–>pdFPersonData–>dtFBirthDate=
                                          dtpFBirthDateToClipboard–>DateTime;
   ClipboardExchange–>ToClipboard();
   CloseClipboard();
  }
}
void __fastcall TForm2::btnFDataFromClipboardClick(TObject *Sender)
{
 if (OpenClipboard(this–>Handle))
   {
     ClipboardExchange–>FromClipboard();
     edtFNameFromClipboard–>Text=ClipboardExchange–>pdFPersonData–>strFName;
     dtpFBirthDateFromClipboard–>DateTime=
                              ClipboardExchange–>pdFPersonData–>dtFBirthDate;
     CloseClipboard();
     }
}.
```

Контрольные вопросы

1. Какую последовательность действий необходимо выполнить для обмена произвольными данными с буфером обмена?
2. Прокомментируйте приведенный пример обмена произвольными данными с буфером обмена.

12.14. Класс TIniFile

Ini-файл (*англ.* Initialization file) — это файл конфигурации, который содержит данные настроек для Windows и приложений. Основной способ использования Ini-файла — это сохранение состояния приложения во время завершения работы (обработка события OnClose) и восстановление сохраненного состояния приложения во время очередного запуска (обработка события OnCreate).

Ini-файлы — это обычные текстовые файлы, которые можно редактировать и просматривать при помощи любого текстового редактора. Структурно Ini-файл содержит последовательность секций, каждая секция имеет следующий формат:

```
[Наименование секции]
Наменование параметра1=Значение1
Наменование параметра2=Значение2
        ...
```

где в качестве значения может быть последовательность символов, целое число, логическое, вещественное, дата, время и др. Например, в одном распространенном приложении имеется Ini-файл следующего содержания (приводится в сокращении):

```
[Dropbox]
BrandCode=DRQX
InstallReturn=-1
OfferReasons=-1
   ...
 [GoogleChrome]
BrandCode=VSNG
InstallReturn=-1
OfferReasons=2
   ...
[GoogleToolbar]
BrandCode=VSNH
InstallReturn=-1
  ...
[RemoteOffer]
InstallReturn=-1
OfferResult=-1
Url=
  ...
```

Между строками и справа от описателя может быть вставлен комментарий, который начинается с символа «**;**» и продолжается до конца строки.

Для использования Ini-файлов в C++Builder необходимо подключить пространство имен с помощью директивы **#include <System.IniFiles.hpp>**.

Все средства обработки Ini-файла инкапсулированы в классе **TIniFile**. Иерархия наследования имеет вид **TIniFile** → **TCustomIniFile** → **TObject**. Таким образом, класс **TIniFile** является достаточно простым.

Конструктор **TIniFile** объявлен как

```
inline __fastcall TIniFile(const System::UnicodeString FileName) :
                   TCustomIniFile(FileName).
```

При создании экземпляра класса **TIniFile** необходимо указывать имя Ini-файла. Например,

```
TIniFile * Ini=new TIniFile("c:/options.ini").
```

Рекомендуется размещать Ini-файл в папке приложения:

TIniFile *IniFile=new TIniFile(ExtractFilePath(Application->ExeName) + «exp.ini»);.

В последнем примере считывается путь к exe-файлу, добавляется к нему строка **«exp.ini»** (**exp** от «эксперимент»), полученное имя становится именем Ini-файла. Под этим именем файл будет находиться в папке приложения. Как обычно, если указанный параметром конструктора файл найден, то он открывается, если не найден, то создается и открывается.

Имя Ini-файла можно узнать с помощью свойства

__property System::UnicodeString FileName.

Запись значения параметра в Ini-файл выполняется с помощью методов:

```
virtual void __fastcall WriteString(const System::UnicodeString Section,
        const System::UnicodeString Ident, const System::UnicodeString Value);
virtual void __fastcall WriteInteger(const System::UnicodeString Section,
        const System::UnicodeString Ident, int Value);
virtual void __fastcall WriteBool(const System::UnicodeString Section,
        const System::UnicodeString Ident, bool Value);
virtual void __fastcall WriteFloat(const System::UnicodeString Section,
        const System::UnicodeString Name, double Value);
virtual void __fastcall WriteDate(const System::UnicodeString Section,
        const System::UnicodeString Name, System::TDateTime Value);
virtual void __fastcall WriteDateTime(const System::UnicodeString Section,
        const System::UnicodeString Name, System::TDateTime Value);
virtual void __fastcall WriteTime(const System::UnicodeString Section,
        const System::UnicodeString Name, System::TDateTime Value);.
```

Во всех методах первым параметром является наименование секции, вторым — наименование параметра (ключа), третьим — значение параметра.

Приведем соответствующие методы чтения значений параметров из Ini-файла:

```
virtual System::UnicodeString __fastcall ReadString(const System::UnicodeString Section,
        const System::UnicodeString Ident, const System::UnicodeString Default);
virtual int __fastcall ReadInteger(const System::UnicodeString Section,
        const System::UnicodeString Ident, int Default);
virtual bool __fastcall ReadBool(const System::UnicodeString Section,
        const System::UnicodeString Ident, bool Default);
virtual double __fastcall ReadFloat(const System::UnicodeString Section,
        const System::UnicodeString Name, double Default);
virtual System::TDateTime __fastcall ReadDate(const System::UnicodeString Section,
        const System::UnicodeString Name, System::TDateTimeDefault);
virtual System::TDateTime __fastcall ReadDateTime(const System::UnicodeString Section,
        const System::UnicodeString Name, System::TDateTime Default);
virtual System::TDateTime __fastcall ReadTime(const System::UnicodeString Section,
        const System::UnicodeString Name, System::TDateTime Default);.
```

Первый и второй параметры перечисленных методов обозначают, соответственно, наименование секции и ключа. Третий параметр определяет значение по умолчанию, которое возвратит функция в случае неуспешного чтения значения параметра из Ini-файла. В случае успешного чтения значения параметра функция возвращает прочитанное значение.

Осталось рассмотреть примеры использования конфигурационного файла. Объявление конфигурационного файла имеет вид: **TIniFile * IniFile**. Это объявление можно поместить рядом с объявлением экранной формы.

В конструктор экранной формы **Form2** помещаем оператор создания экземпляра класса **TIniFile**:

```
__fastcall TForm2::TForm2(TComponent* Owner) : TForm(Owner)
{
  IniFile=new TIniFile(ExtractFilePath(Application–>ExeName)+»exp.ini»);
}.
```

Обработчик события завершения работы с экранной формой приложения будет иметь вид:

```
void __fastcall TForm2::FormClose(TObject *Sender, TCloseAction &Action)
{
        IniFile–>WriteString(L»Service.vol.1»,L»Caption»,»Конфигурация»);
        IniFile–>WriteBool(L»Service.vol.1»,L»ErrorControl»,True);
        IniFile–>WriteInteger(L»Service.vol.1»,L»Mode»,5);
        IniFile–>WriteFloat(L»Service.vol.2»,L»Accuracy»,5.3);
        TDateTime CurrentDateTime=Now();
        TDateTime CurrentDate=Date();
        TDateTime CurrentTime=Time();
        IniFile–>WriteDateTime(L»Service.vol.2»,L»Дата+время»,CurrentDateTime);
        IniFile–>WriteDate(L»Service.vol.2»,L»Дата»,CurrentDate);
        IniFile–>WriteTime(L»Service.vol.2»,L»Время»,CurrentTime);
}.
```

После выполнения метода **FormClose** мы сможем посмотреть содержимое файла **«exp.ini»**:

```
[Service.vol.1]
Caption=Конфигурация
ErrorControl=1
Mode=5
[Service.vol.2]
Accuracy=5,3
Дата+время=19.03.2017 10:59:15
Дата=19.03.2017
Время=10:59:15.
```

Загрузка и создание главной экранной формы нашего приложения будет сопровождаться вызовом обработчика:

```
void __fastcall TForm2::FormCreate(TObject *Sender)
{
  Memo1–>Lines–>Clear();
```

```
  Memo1–>Lines–>Add(IniFile–>ReadString(L»Service.vol.1»,L»Caption»,»«));
  bool b=IniFile–>ReadBool(L»Service.vol.1»,L»ErrorControl»,False);
  if (b) Memo1–>Lines–>Add(«True»); else Memo1–>Lines–>Add(«False»);
  Memo1–>Lines–>Add(IntToStr(IniFile–
>ReadInteger(L»Service.vol.1»,L»Mode»,5)));
  Memo1–>Lines–>Add(FloatToStr(IniFile–
>ReadFloat(L»Service.vol.2»,L»Accuracy»,0)));
  TDateTime CurrentDateTime=Now();
  Memo1–>Lines–>Add(DateTimeToStr(IniFile–>
              ReadDateTime(L»Service.vol.2»,L»Дата+время»,CurrentDateTime)));
  TDateTime CurrentDate=Date();
  Memo1–>Lines–>Add(DateToStr(IniFile–
>ReadDate(L»Service.vol.2»,L»Дата»,CurrentDate)));
  TDateTime CurrentTime=Time();
  Memo1–>Lines–>Add(TimeToStr(IniFile–
>ReadTime(L»Service.vol.2»,L»Дата»,CurrentTime)));
}.
```

Содержимое поля **Memo1** при начальном состоянии приложения будет иметь вид:

```
Конфигурация
True
5
5,3
18.03.2017 10:21:00
18.03.2017
10:21:07.
```

Контрольные вопросы

1. Каково назначение файла конфигурации Ini-файла?
2. Какой формат представления данных используется в Ini-файле?
3. Как задается имя Ini-файла в программе?
4. Значения каких типов данных мы можем размещать в Ini-файле?
5. Какие возможности класс **TIniFile** унаследовал от класса **TObject**?

12.15. Класс TForm

Основным элементом любого Windows-приложения является экранная форма. Форма является контейнером, в котором размещаются другие визуальные и невизуальные компоненты. Создание приложений — это процесс проектирования экранных форм, размещение на них компонентов, настройка их и добавление кода. Терминологически форма — это просто другое наименование окна. RAD Studio сильно упрощает процесс создания, установки и настройки свойств форм. Хотя поэкспериментировать с ними — заманчивая идея, но при создании серьезных приложений уместен определенный консерватизм. Одним из преимуществ Windows является единый принцип построения интерфейса и сходное поведение большинства программ. Существует следующая рекомендация: если мы не уверены, какой стиль построения формы использовать, то нуж-

но найти в другом приложении Windows форму, схожую с той, которую мы хотим использовать, и затем желательно воспроизвести ее внешний вид, поведение и функциональные возможности. Пользователь нашей программы, конечно, это не заметит, однако его реакция на несоблюдение данной рекомендации будет однозначно негативной.

Для проектирования форм приложения в библиотеке визуальных компонентов предусмотрен класс **TForm**. В настоящем параграфе рассмотрены свойства, методы и события класса **TForm**.

12.15.1. Класс TForm в иерархии наследования

На рисунке 12.14 представлена иерархия наследования для класса **TForm**. Рассмотрим свойства, методы и события класса **TForm**, получаемые в результате наследования. Рассмотрение начнем с корня иерархии, продвигаясь по непосредственным потомкам к классу **TForm**.

В корне иерархии находится класс **TObject**, от которого наследуются свойства и методы, придающие классу **TForm** черты класса и возможность построения соответствующего объекта. Содержание класса **TObject** подробно рассмотрено в параграфе 6.2.

За классом **TObject** следует класс **TPersistent**, который позволяет взаимодействовать со средой проектирования и выполнять нестандартные действия. Класс **TPersistent** рассмотрен в параграфе 12.2.

Присутствие в иерархии наследования класса **TComponent** (см. п. 12.3) придает классу **TForm** свойства «быть компонентом», «являться компонентом». Так, например, экземпляры класса **TForm** могут быть владельцами других окон: панелей, полей редактирования текстов, полей обработки древовидных структур и т. п. С другой стороны, в классе **TScreen** имеется свойство Forms, которое позволяет экрану быть владельцем расположенных на экране форм.

Далее предком класса **TForm** является класс **TControl** (см. п. 12.4), наследуемые свойства и методы которого позволяют отображать экземпляры класса **TForm** на экране стандартным способом.

Класс **TWinControl** (п. 12.5) обеспечивает класс **TForm** возможностью взаимодействия с пользователем и операционной системой.

Класс **TScrollingWinControl** ранее не рассматривался (предлагается изучить самостоятельно), но из названия класса нетрудно установить назначение: управление скроллингом.

Класс **TCustomForm** является базовым классом для класса **TForm**. Класс **TCustomForm** упорядочивает наследование свойств и событий всей иерархии наследования, изменяя их область видимости и предоставляя пользователю класса **TForm** необходимые для эффективного использования класса и скрывая вспомогательные, но существенные для функционирования предков средства.

У многих визуальных компонентов имеется непосредственный предок, имеющий имя вида **TCustom**XXXX. В этом классе свойства и методы получают такие области видимости, при которых создаются наилучшие условия для построения соответствующего визуального компонента. При этом свойства и методы, имеющие локальное для предка значение, скрываются, а наиболее значимые

продолжают свою «деятельность» в визуальных компонентах. В инспекторе объекта отображаются не все свойства объекта (только те, что попадают в секцию **published**).

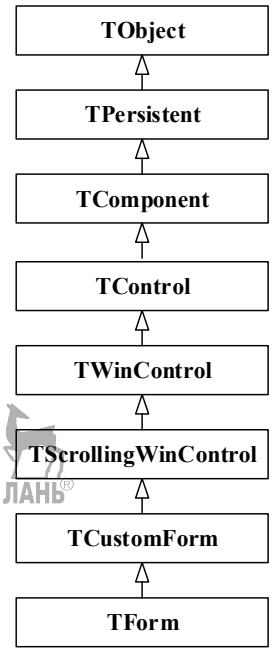


Рис. 12.14
Иерархия наследования класса **TForm**

Заметим, что развитая система наследования придает классу **TForm** богатый набор возможностей по проектированию и управлению программой.

В следующих параграфах рассматриваются свойства, методы и события класса **TForm**. Рассмотрение сопровождается информацией о месте объявления указанного средства, областях видимости, изменении областей видимости в потомках. Например, свойство **ClientHeight** объявлено в области **public** класса **TControl**, затем переведено в область **protected** в классе **TCustomForm** и, наконец, в классе **TForm** переведены в область видимости **published**. Рекомендуется объяснить причины указанных изменений области видимости каждого свойства. Для указанного свойства **ClientHeight** (высота клиентской области формы) рассуждения могут быть следующими. Класс **TControl** предназначен для придания компонентам свойства визуальности. Свойство **ClientHeight** (высота клиентской области компонента) может использоваться как в потомках компонента, так и в клиентских компонентах, поэтому свойству **ClientHeight** в классе **TControl** была назначена область видимости **public**. Класс **TCustomForm** является специализированным классом для создания на его основе экранных форм, поэтому свойство **ClientHeight** (высота клиентской области экранной формы) переведено в классе **TCustomForm** в область видимости **protected**. Тем самым обеспечиваются единые правила использования свойства **ClientHeight** во всех экранных формах, которые строятся как потомки класса **TCus-**

tomForm. Клиентам класса **TCustomForm** это свойство не будет доступно. Наконец, в классе **TForm** свойство переведено в область видимости **published** (опубликовано). Это можно объяснить желанием обеспечить доступность свойства **ClientHeight** как во время проектирования экранной формы, так и во время исполнения использующей экранную форму программы. Кроме того, область видимости **published** обеспечивает доступность свойства **ClientHeight** как в потомках экранной формы, так и в компонентах-клиентах экранной формы.

Контрольные вопросы

1. Какова иерархия наследования класса TForm?
2. Исходя из иерархии наследования, опишите, какие возможности получает класс TForm по наследству?

12.15.2. Свойства и методы класса TForm

Свойства формы. Свойства и методы формы часто встречаются у других визуальных компонентов, так как многие из них вводятся в предках класса **TForm** и затем передаются по другим ветвям наследования.

Рассмотрим основные свойства формы в алфавитном порядке.

Свойство **Active** введено в классе **TCustomForm** в области видимости **public**[48]

__property bool Active.

Свойство определяет наличие фокуса на форме.

Свойство **ActiveControl** введено в классе **TCustomForm** в области видимости **public**

__property Vcl::Controls::TWinControl* ActiveControl

и переведено в область видимости **published** в классе **TForm**. Свойство **ActiveControl** определяет, какой компонент, помещенный в форму, в данный момент является активным (имеет фокус).

Свойство **Align** введено в классе **TControl** (см. п. 12.15.2) в области видимости **protected**

__property TAlign Align

и переведено в область видимости **published** в классе **TForm**. Тип **TAlign** определен как

enum TAlign : unsigned char { alNone, alTop, alBottom, alLeft, alRight, alClient, alCustom }.

Свойство **Align** определяет, остается ли компонент неизменным при изменении размеров содержащей его формы, панели, другого компонента, или он изменяется, занимая всю доступную площадь, ее верхнюю, нижнюю, левую или правую часть. Для лучшего понимания действия каждого значения рекомендуем поэкспериментировать со значениями свойства **Align**.

[48] Ввиду того, что свойство **Active** находится в области видимости **public,** оно может использоваться только во время исполнения программы.

Свойство **AutoScroll** введено в классе **TScrollingWinControl** в области видимости **protected**

__property bool AutoScroll

и переведено в область видимости **published** в классе **TForm**. Свойство определяет, будут ли автоматически появляться полосы прокрутки формы, если в ней не будут помещаться компоненты (т. е. чтобы их увидеть, надо будет форму «прокрутить»). Если значение этого свойства **True**, то полосы прокрутки будут появляться при необходимости, а если **False** — то не будут.

Свойство **AutoSize** введено в классе **TControl** в области видимости **protected**

__property bool AutoSize

и переведено в область видимости **published** в классе **TForm**. Свойство определяет, будет ли форма принимать прежние размеры при ее растяжке. Если значение свойства **True**, то форма автоматически принимает прежние размеры, как бы мы ее не растягивали. При значении свойства равном **False** форма «позволяет» себя растягивать.

Свойство **BorderStyle** введено в классе **TCustomForm** в области видимости **public**

__property TFormBorderStyle BorderStyle,

где тип **TFormBorderStyle** определен как перечислимый тип данных

enum TFormBorderStyle : unsigned char { bsNone, bsSingle, bsSizeable, bsDialog, bsToolWindow, bsSizeToolWin }.

В классе **TForm** свойство переведено в область видимости **published.**

Свойство **BorderStyle** задает появление и поведение границ формы; определяет, можно ли мышью менять размеры формы, когда приложение находится в режиме исполнения. Свойство **BorderStyle** может принимать следующие значения:

- **bsDialog** — неизменяемые границы стандартного диалогового окна;
- **bsSingle** — неизменяемые границы, одиночная линия;
- **bsNone** — неизменяемые границы, нет видимой границы;
- **bsSizeabie** — стандартная изменяемая граница;
- **bsToolWindow** — подобно **bsSingle**, но с уменьшенным заголовком;
- **bsSizeToolWin** — подобно **bsSizeable**, но с уменьшенным заголовком.

Использование того или иного значения свойства **BorderStyle** рекомендуем изучить экспериментально.

Свойство **BorderWidth** введено в классе **TWinControl** в области видимости **protected**

__property TBorderWidth BorderWidth,

где **TBorderWidth** является синонимом целого типа данных.

Свойство **BorderWidth** задает в пикселях величину отступа координатной сетки формы от границ окна формы, т. е. фактически размеры формы мож-

но изменить за счет изменения координатной сетки, задав ее отступ от границ окна. По умолчанию величина отступа равна нулю, т. е. форма, занимает все пространство окна.

Свойство **Canvas** (см. п. 12.6) объявлено в разделе **public** класса **TCustomForm**

__property Vcl::Graphics::TCanvas* Canvas.

Свойство **Canvas** обеспечивает программисту возможность рисования в форме.

Свойство **Caption** — название формы, помещаемое в заголовке, появилось в классе **TControl** (см. п. 12.7) в области видимости **protected** с возможностью передачи его потомкам, затем в классе **TCustomForm** было переведено в область видимости **public** и, наконец, в классе **TForm** получило атрибут видимости **published** и появилось в инспекторе объектов. По умолчанию первая форма проекта автоматически получает заголовок **Form1**, вторая — **Form2** и т. д.

Свойства **ClientHeight**, **ClientWidth** (см. п. 12.7.1) введены в области видимости **public** класса **TControl**, переведены в область **protected** в классе **TCustomForm** и, наконец, в классе **TForm** переведены в область видимости **published**. Напомним, что свойства **ClientHeight**, **ClientWidth** хранят сведения, соответственно, о высоте и ширине клиентской области формы в пикселах. При изменении значений этих свойств автоматически изменяются значения свойств формы **Height** и **Width**.

Свойство **Color** определяется в разделе **protected** класса **TControl**

__property System::Uitypes::TColor Color,

где **TColor** задает диапазон целых чисел, каждое число из диапазона соответствует некоторой константе, определяющей цвет вида

static const System::Uitypes::TColor clGreen=System::Uitypes::TColor(32768).

В классе **TCustomForm** свойство **Color** переведено в область видимости **public**, а в классе **TForm** — в область видимости **published.**

Color — это свойство задает цвет поверхности формы. Цвет можно выбрать из раскрывающегося списка, который появится, если нажать кнопку в поле этого свойства. В этом списке следует установить стрелками или мышью строку подсветки на нужный цвет и щелкнуть мышью или нажать клавишу <Enter>. Кроме того, цвет можно выбрать из палитры цветов, которая появится, если дважды щелкнуть кнопкой мыши в поле.

Свойство **Constraints** (ограничения) определено в классе **TControl** в области видимости **public**

__property TSizeConstraints* Constraints,

где **TSizeConstraints** — имя класса, в котором присутствуют четыре опубликованных свойства

__property TConstraintSize MaxHeight — максимальная высота;
__property TConstraintSize MaxWidth — максимальная ширина;

__property TConstraintSize MinHeight — минимальная высота;
__property TConstraintSize MinWidth — минимальная ширина.

Тип **TConstraintSize** является синонимом целого.

В перечисленных свойствах указываются ограничительные величины на размеры компонента (в данном случае — формы). Это свойство рекомендуется не изменять, так как его значения связаны со свойствами **Align** (выравнивание объекта относительно контейнера его содержащего) и **Anchors** (якоря), которые задают привязку компонента к родителю. Компонент поддерживает свое текущее месторасположение относительно угла родительского окна, даже тогда, когда родительское окно изменяет размеры. В этом случае компонент удерживает свою позицию по отношению к тому углу, к которому он привязан. Якоря и задают эту привязку.

В классе **TForm** свойство **Constraints** переведено в раздел **published**.

Свойство **Constraints** устанавливает пределы автоматического изменения размеров формы.

Свойство **Ctl3D** введено в области видимости **protected** класса TWinControl

__property bool Ctl3D,

затем переведено в область видимости **published** в классе **TScrollingWinControl**, далее оказалось в области видимости **protected** в классе **TCustomForm** и, наконец, переведено в область видимости **published** в классе **TForm**.

Свойство **Ctl3D** придает форме объемный (3D) вид.

Свойство **Cursor** (см. п. 12.12) введено и опубликовано в классе **TControl**

__property System::Uitypes::TCursor Cursor.

Форма курсора будет действовать над областью всего компонента. Выбор курсора аналогичен выбору цвета, но при двойном щелчке в окошечке будут появляться следующие по списку формы курсора.

Свойство **DockSite** включено в классе **TWinControl** в области видимости **public**

__property bool DockSite.

Свойство **DockSite** задает способность компонента стать участком стыковки для других компонентов. Применяется для различных панелей, объектов группирования и т. д. При включении (**True**) позволяет использовать этот объект как базу для стыковки других объектов на форме при перетаскивании. Обеспечивает режим «прилипания».

Свойства **DragKind**, **DragMode** (рассмотрены в п. 12.7.3) определяют возможности формы при операциях перетаскивания элементов. Определены в классе **TControl** в области видимости **protected**, переведены в классе **TForm** в область видимости **published**.

Свойство **Enabled** (см. п. 12.7.1) определяет возможность реакции формы на события мыши, клавиатуры и таймеров. Объявлено в классе **TControl** в области видимости **public**, переведено в классе **TForm** в область видимости **published**.

Свойство **Font** объявлено в классе **TControl** в области видимости **protected**

__property Vcl::Graphics::TFont* Font,

далее оказалось в области видимости **public** в классе **TCustomForm**, наконец, в классе **TForm** свойство **Font** опубликовано.

Свойство **Font** определяет шрифт надписей и текстов компонентов, имеет множество подсвойств. Кроме того, в компонентах имеется свойство **ParentFont**. Если это свойство установлено в **True**, то шрифт данного компонента берется из свойства **Font** его родительского компонента — панели или формы, на которой расположен компонент. Использование свойств **ParentFont** и **ParentColor** помогает обеспечить единообразие отображения компонентов в окне приложения.

Свойство **FormStyle** введено в области видимости **protected** класса **TCustomForm** и опубликовано в классе **TForm**. Свойство **FormStyle** используется для управления многооконными приложениями. Рекомендуем эту тему изучить самостоятельно.

Свойства **Height** и **Width** — размеры формы (см. п. 12.7.1) объявлены в классе **TControl** в области видимости **published**, в классе **TCustomForm** свойства переведены в область видимости **protected**, в классе **TForm** опубликованы. Используются для изменения размеров компонентов при изменении размеров окна приложения.

Свойство **Hint** объявлено в классе **TControl** в области видимости **published**

__property System::UnicodeString Hint.

Свойство **Hint** содержит текст подсказки, появляющейся при наведении на форму указателя мыши. Во время работы приложения значение свойства передается в одноименное свойство экземпляра класса **TApplication**. Правила работы со свойством **Hint** компонента и класса **TApplication** рассмотрены в параграфе 12.12.

Свойства **Left** и **Top** — координаты экранной формы введены в классе TControl (рассмотрены в п. 12.7).

Свойство **Name** — имя экранной формы как компонента введено и опубликовано в классе **TComponent** (см. п. 12.5).

Свойство **Position** определяет положение и размеры формы. Свойство

__property TPosition Position

введено в области видимости **protected** класса **TCustomForm**, в классе **TForm** свойство опубликовано.

Определение класса **TPosition** имеет вид:

enum TPosition : unsigned char { poDesigned, poDefault, poDefaultPosOnly, poDefaultSizeOnly, poScreenCenter, poDesktopCenter, poMainFormCenter, poOwnerFormCenter }.

Значениями свойства **Position** могут быть:

— **poDesigned** — принять те значения положения и размеров, которые были определены во время разработки приложения.

— **poDefault** — принять положение и размеры, определенные средой Windows по умолчанию;

— **poDefaultPosOnly** — принять только положение, определенное средой Windows по умолчанию;

— **poDefaultSizeOnly** — принять только размеры, определенные средой Windows по умолчанию;

— **poScreenCenter** — разместить окно по центру экрана (при этом размеры будут равны половине ширины и высоты экрана в пикселах, а для дочерней формы — половине соответствующих размеров клиентского окна) и т. д.

Свойство **ShowHints** — разрешение на отображении подсказки. Правила работы со свойством **ShowHints** рассмотрены в параграфе 12.12.

Свойство **Visible** — видимость экранной формы (см. п. 12.7). Введено в области видимости **public** класса **TControl**, переведено в область видимости **protected** в классе **TCustomForm**, в классе **TForm** свойство опубликовано.

Остается добавить, что почти невозможно кратко, но исчерпывающе рассказать обо всех свойствах экранной формы. Необходимо самостоятельно попробовать все доступные значения и понять их действия. Не бойтесь экспериментировать: читая только справочную информацию, невозможно полностью оценить интерфейс и возможности визуальных компонентов.

Рассмотрим теперь наиболее полезные методы формы.

Метод **void __fastcall Close(void)** объявлен в классе **TCustomForm** в области видимости **public**. В результате выполнения метода форма закрывается, а если закрывается главная форма, то закрывается и приложение.

Метод **void __fastcall Hide(void)** объявлен в классе **TCustomForm** в области видимости **public**. Вызов метода устанавливает свойству **Visible** значение **False** и форма становится невидимой.

Метод **void __fastcall Release(void)** разрушает форму, при этом освобождается память, занятая формой. Метод введен в области видимости **public** класса **TCustomForm**.

Метод virtual **void __fastcall SetFocus(void)** впервые объявлен в классе **TWinControl** в области видимости **public** (см. п. 12.8.1). Затем метод переопределяется в области видимости **public** класса **TCustomForm**. Вызов метода делает форму активной: свойства **Visible** и **Enabled** устанавливаются в **True**.

Метод **void __fastcall Show*(void)*** объявлен в классе **TCustomForm** в области видимости **public**. Вызов метода устанавливает свойству **Visible** значение **True**, и форма становится видимой и помещается поверх всех форм на экране.

Контрольные вопросы

1. Каково назначение свойства **Active**?
2. Каково назначение свойства **ActiveControl**?
3. Каково назначение свойства **Align**?
4. Каково назначение свойства **AutoScroll**?
5. Каково назначение свойства **AutoSize**?
6. Каково назначение свойства **BorderStyle**?
7. Каково назначение свойства **BorderWidth**?

8. Каково назначение свойства **Canvas**?
9. Каково назначение свойства **Caption**?
10. Каково назначение свойства **ClientHeight**?
11. Каково назначение свойства **ClientWidth**?
12. Каково назначение свойства **Constraints**?
13. Каково назначение свойства **Ctl3D**?
14. Каково назначение свойства **Cursor**?
15. Каково назначение свойства **DockSite**?
16. Каково назначение свойства **DragKind**?
17. Каково назначение свойства **DragMode**?
18. Каково назначение свойства **Enabled**?
19. Каково назначение свойства **Font**?
20. Каково назначение свойства **FormStyle**?
21. Каково назначение свойства **Height**?
22. Каково назначение свойства **Width**?
23. Каково назначение свойства **Hint**?
24. Каково назначение свойства **Left**?
25. Каково назначение свойства **Top**?
26. Каково назначение свойства **Name**?
27. Каково назначение свойства **Position**?
28. Каково назначение свойства **ShowHints**?
29. Каково назначение свойства **Visible**?
30. Каково назначение метода **Close**?
31. Каково назначение метода **Hide**?
32. Каково назначение метода **Release**?
33. Каково назначение метода **SetFocus**?
34. Каково назначение метода **Show**?

12.15.3. События класса TForm

Событие **OnActivate** возникает, когда форма активизируется. Событие введено в классе **TCustomForm** в области видимости **protected**

__property System::Classes::TNotifyEvent OnActivate.

В классе **TCustomForm** событие **OnActivate** переведено в область видимости **protected**, в классе **TForm** событие опубликовано.

Событие **OnClick** возникает при щелчке мышью в форме. Событие введено в классе **TControl** в области видимости **protected**

__property System::Classes::TNotifyEvent OnClick.

В классе **TForm** событие **OnClick** опубликовано.

Событие **OnClose** возникает, когда форма становитя ненужной. Событие введено в классе **TCustomForm** в области видимости **protected**

__property TCloseEvent OnClose,

где **TCloseEvent** определяет процедурный тип

typedef void __fastcall (__closure *TCloseEvent)(System::TObject* Sender, System::Uitypes::TCloseAction &Action).

Здесь параметр ***Sender**, как обычно, передает адрес вызвавшего событие объекта, а второй параметр (передается по адресу) имеет тип **TCloseAction**, определенный следующим образом:

enum class TCloseAction: unsigned char {caNone, caHide, caFree, caMinimize},

Значения типа **TCloseAction** определяют способы закрытия формы:

— **caNone —** не делать ничего;
— **caHide —** сделать форму невидимой (скрыть);
— **caFree —** уничтожить форму, вызвав ее деструктор;
— **caMinimize —** свернуть форму в значек.

Перед вызовом **OnClose** в параметр **Action** засылается значение **caHide**. Исключение составляют дочерние формы, для которых засылается значение **caMinimize** или **caNone** в зависимости от возможности их сворачивания.

Если обработчик установил значение **caNone**, форма не закрывается. При закрытии главной формы приложения любое отличное от **caNone** значение вызывает завершение приложения. Для формы, не являющейся главной, значения **caHide** и **caMinimize** позволяют, соответственно, скрыть и свернуть ее в значок. Выбор значения **caFree** влечет за собой вызов деструктора путем обращения к методу Release(). При уничтожении формы инициализируется событие **OnDestroy**.

В классе **TForm** событие **OnClose** опубликовано.

Событие **OnCreate** возникает при создании формы. Описание события в области видимости **protected** класса имеет **TCustomForm** вид

__property System::Classes::TNotifyEvent OnCreate.

В классе **TForm** событие **OnCreate** опубликовано.

Событие **OnDeactivate** возникает, когда форма перестает быть активной. Событие введено в классе **TCustomForm** в области видимости **protected**

__property System::Classes::TNotifyEvent OnDeactivate.

В классе **TForm** событие **OnCreate** опубликовано.

Событие **OnKeyDown** возникает, когда пользователь нажимает некоторую клавишу. Свойство **OnKeyDown** объявлено в разделе **protected** класса **TWinControl** (см. п. 12.8.2). В классе **TForm** событие **OnKeyDown** опубликовано.

Событие **OnKeyPress** возникает, когда пользователь нажимает некоторую (только одну) клавишу, кроме функциональной. Свойство **OnKeyPress** объявлено в разделе **protected** класса **TWinControl** (см. п. 12.8.2). В классе **TForm** событие **OnKeyPress** опубликовано.

Событие **OnKeyUp** возникает, когда пользователь отпускает клавишу, которая до этого была нажата. Это событие возникает и тогда, когда до этого были нажаты и комбинации клавиш с <Shift>, <Alt> и <Ctrl>, а также функциональные

клавиши. Свойство **OnKeyUp** объявлено в разделе **protected** класса **TWinControl** (см. п. 12.8.2). В классе **TForm** событие **OnKeyUp** опубликовано.

Событие **OnMouseDown** возникает при нажатии любой кнопки мыши, а также при комбинации клавиш <Shift>, <Ctrl> и <Alt> и кнопок мыши. Свойство **OnMouseDown** объявлено в разделе **protected** класса **TControl** (см. п. 12.7.2). В классе **TForm** событие **OnMouseDown** опубликовано.

Событие **OnMouseMove** возникает, когда пользователь двигает указатель мыши, пока он находится над объектом (в нашем случае — над формой). Свойство **OnMouseMove** объявлено в разделе **protected** класса **TControl** (см. п. 12.7.2). В классе **TForm** событие **OnMouseMove** опубликовано.

Событие **OnMouseUp** возникает, когда пользователь отпускает кнопку мыши, которая была нажата на компоненте. Свойство **OnMouseUp** объявлено в разделе **protected** класса **TControl** (см. п. 12.7.2). В классе **TForm** событие **OnMouseUp** опубликовано.

Событие **OnPaint** возникает, когда начинается прорисовка формы. Свойство объявлено в области видимости **protected** класса **TCustomForm**

__property System::Classes::TNotifyEvent OnPaint.

В классе **TForm** событие **OnPaint** опубликовано.

Событие **OnShow** возникает в момент, когда форма появляется на экране. Свойство объявлено в области видимости **protected** класса **TCustomForm**

__property System::Classes::TNotifyEvent OnShow.

В классе **TForm** событие **OnShow** опубликовано.

Контрольные вопросы

1. Когда возникает событие **OnActivate**?
2. Когда возникает событие **OnClick**?
3. Когда возникает событие **OnClose**?
4. Когда возникает событие **OnCreate**?
5. Когда возникает событие **OnDeactivate**?
6. Когда возникает событие **OnKeyDown**?
7. Когда возникает событие **OnKeyPress**?
8. Когда возникает событие **OnMouseDown**?
9. Когда возникает событие **OnMouseMove**?
10. Когда возникает событие **OnMouseUp**?
11. Когда возникает событие **OnPaint**?
12. Когда возникает событие **OnShow**?

Задачи

Приведенные в этой главе задачи образуют цикл заданий по созданию приложения.

Целью заданий является получение навыков разработки приложений на языке программирования C++Builder в системе программирования **RAD** (**R**apid **A**pplication **D**evelopment — быстрая разработка приложений) **Studio** for Microsoft Windows.

К выполнению каждого задания (кроме первого) следует приступать только по завершению всех предыдущих. Каждое очередное задание предусматривает развитие приложения, полученного на предыдущем шаге. Рассмотренные задания могут использоваться в качестве лабораторных работ по дисциплине «Объектно-ориентированное программирование».

Задание 31. Программирование векторной графики

31.1. Описание приложения. Приложение состоит из двух областей: рабочей области и области элементов управления (рис. 31).

В рабочей области приложения расположена геометрическая фигура. В последующих заданиях эта геометрическая фигура будет рассматриваться как траектория движения другой геометрической фигуры, поэтому здесь и далее рассматриваемая фигура называется траекторией.

В области элементов управления расположен «движок». Пользователь имеет возможности:

) с помощью «движка» «Размер траектории» изменить размер траектории;

) путем захвата и перемещения мышью края формы изменить размер формы;

) завершить работу приложения.

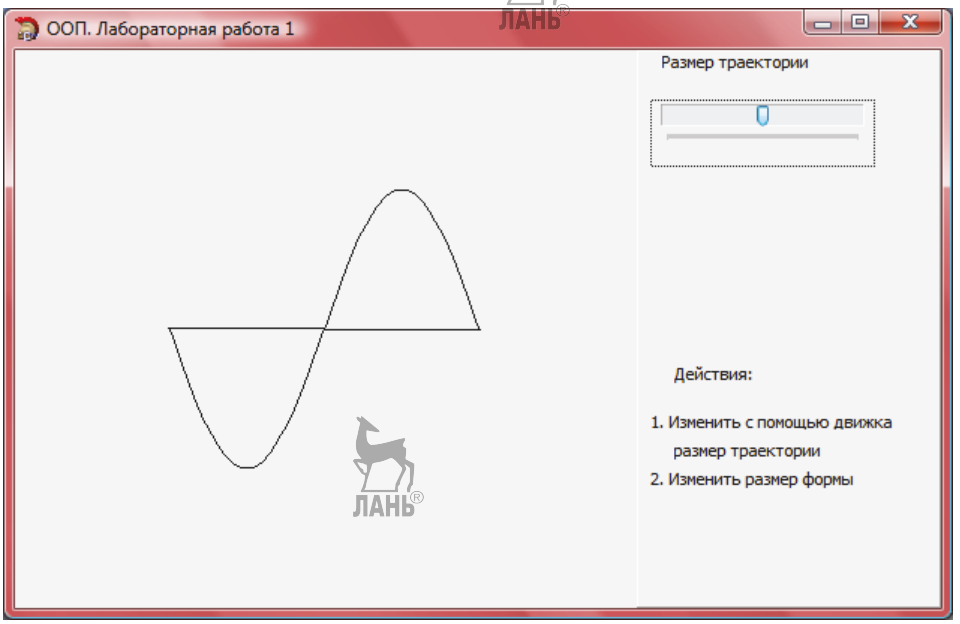


Рис. 31
Общий вид приложения

31.2. Задание. Разработать приложение на языке программирования C++ в системе программирования **RAD Studio for Microsoft Windows**.

Требования:

1. Геометрический центр траектории всегда должен совпадать с центром рабочей области формы.

2. Траектория не должна выходить за пределы рабочей области формы.

3. При уменьшении линейных размеров траектории масштаб не должен становиться отрицательным (траектория не должна «выворачиваться наизнанку»).

4. При изменении размеров формы масштаб прорисовки траектории должен пропорционально измениться.

5. Для моделирования траектории в программе предусмотреть отдельный класс «Траектория», размещенный в отдельном программном модуле.

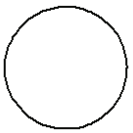
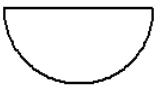
6. Необходимо использовать правила именования идентификаторов (см. Приложение 2).

7. В разделе **public** описания класса «Траектория» должны находиться только описания свойств (**property**) и методов.

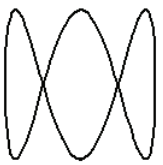
31.3. Варианты заданий приведены в таблице 31.1.

Таблица 31.1

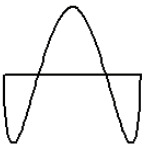
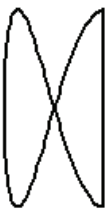
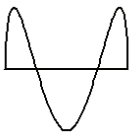
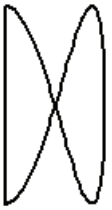
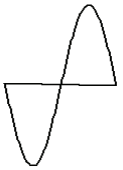
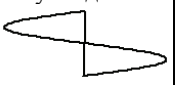
Варианты траекторий

№ варианта траектории	Вид траектории	Уравнение траектории	Примечание
	Круг 	$x=x_0+r*\cos(\varphi)$, $y=y_0-r*\sin(\varphi)$, где x — координата точки по оси x, y — координата точки по оси y, $\varphi \in [0-2\pi]$ — угол, r — радиус окружности, x_0 — координата центра круга, y_0 — координата центра круга	
1		Угол изменяется в диапазоне $\varphi \in [0-\pi]$	Траектория является полукругом
2		Угол изменяется в диапазоне $\varphi \in \left[\frac{\pi}{2}-\frac{3\pi}{2}\right]$	Траектория является полукругом
3		Угол изменяется в диапазоне $\varphi \in [\pi-2\pi]$	Траектория является полукругом
4		Угол изменяется в диапазоне $\varphi \in \left[\frac{3\pi}{2}-\frac{5\pi}{2}\right]$	Траектория является полукругом

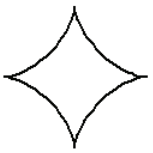
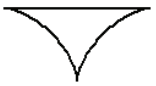
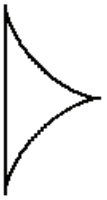
Продолжение табл. 31.1

№ варианта траектории	Вид траектории	Уравнение траектории	Примечание
4	Эллипс	$x=x_0+a*\cos(\varphi)$, $y=y_0-b*\sin(\varphi)$, где x — координата точки по оси x, y — координата точки по оси y, $\varphi \in [0-2\pi]$ — угол, a — размер горизонтальной полуоси, b — размер вертикальной полуоси, x_0 — координата центра эллипса, y_0 — координата центра эллипса	
5		Угол изменяется в диапазоне $\varphi \in [0-\pi]$	Траектория является полуэллипсом
6		Угол изменяется в диапазоне $\varphi \in \left[\frac{\pi}{2}-\frac{3\pi}{2}\right]$	Траектория является полуэллипсом
7		Угол изменяется в диапазоне $\varphi \in [0-2\pi]$	Траектория является полуэллипсом
8		Угол изменяется в диапазоне $\varphi \in \left[\frac{3\pi}{2}-\frac{5\pi}{2}\right]$	Траектория является полуэллипсом
	Фигура Лиссажу 	$x=x_0+r\cdot\sin(\varphi+\frac{\pi}{2})$, $y=y_0+r*\sin(k*\beta)$, где $\varphi \in [0-2\pi]$ — угол, $\beta=k*\varphi$, $k=3$, r — радиус фигуры Лиссажу, x_0 — координата центра фигуры, y_0 — координата центра фигуры	

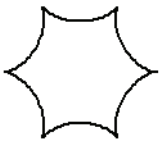
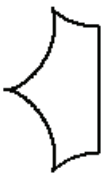
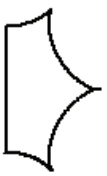
Продолжение табл. 31.1

№ варианта траектории	Вид траектории	Уравнение траектории	Примечание
9		Угол изменяется в диапазоне $\varphi \in [0 - \pi]$	Траектория является полуфигурой Лиссажу
10		Угол изменяется в диапазоне $\varphi \in \left[\frac{\pi}{2} - \frac{3\pi}{2}\right]$	Траектория является полуфигурой Лиссажу
11		Угол изменяется в диапазоне $\varphi \in [0 - 2\pi]$	Траектория является полуфигурой Лиссажу
12		Угол изменяется в диапазоне $\varphi \in \left[\frac{3\pi}{2} - \frac{5\pi}{2}\right]$	Траектория является полуфигурой Лиссажу
13	Синусоида 	$x = x_0 - \frac{a \cdot \varphi}{2\pi}$, $y = y_0 - b*\sin(\varphi)$, где a — период, b — амплитуда, $\varphi \in [0 - 2\pi]$ — угол, x_0 — координата правого конца фигуры, y_0 — координата горизонтальной оси	Траектория является синусоидой
14	Синусоида 	$x = x_0 - a*\sin(\varphi)$, $y = y_0 + \frac{b \cdot \varphi}{2\pi}$, где a — амплитуда, b — период, $\varphi \in [0 - 2\pi]$ — угол, x_0 — координата вертикальной оси, y_0 — координата верхнего конца фигуры	Траектория является синусоидой

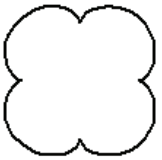
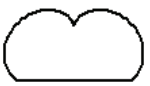
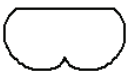
Продолжение табл. 31.1

№ варианта траектории	Вид траектории	Уравнение траектории	Примечание
	Гипоциклоида 	$x = x_0 + r \cdot (k-1)(\cos(\varphi) + \frac{\cos((k-1)\varphi)}{k-1})$, $y = y_0 + r \cdot (k-1)(\sin(\varphi) - \frac{\sin((k-1)\varphi)}{k-1})$, где $\varphi \in [0 - 2\pi]$ — угол, $k = \frac{R}{r} = 4$, R — радиус неподвижной окружности, r — радиус катящейся окружности, x_0 — координата центра фигуры, y_0 — координата центра фигуры	
15		Угол изменяется в диапазоне $\varphi \in [0 - \pi]$	Траектория является полуастроидой
16		Угол изменяется в диапазоне $\varphi \in \left[\frac{\pi}{2} - \frac{3\pi}{2}\right]$	Траектория является полуастроидой
17		Угол изменяется в диапазоне $\varphi \in [\pi - 2\pi]$	Траектория является полуастроидой
18		Угол изменяется в диапазоне $\varphi \in \left[\frac{3\pi}{2} - \frac{5\pi}{2}\right]$	Траектория является полуастроидой

Продолжение табл. 31.1

№ варианта траектории	Вид траектории	Уравнение траектории	Примечание
18	Гипоциклоида 	$x = x_0 + r \cdot (k-1)(\cos(\varphi) + \frac{\cos((k-1)\varphi)}{k-1})$, $y = y_0 + r \cdot (k-1)(\sin(\varphi) - \frac{\sin((k-1)\varphi)}{k-1})$, где $\varphi \in [0 - 2\pi]$ — угол, $k = \frac{R}{r} = 6$, R — радиус неподвижной окружности, r — радиус катящейся окружности, x_0 — координата центра фигуры, y_0 — координата центра фигуры	
19		Угол изменяется в диапазоне $\varphi \in [0 - \pi]$	Траектория является полугипоциклоидой
20		Угол изменяется в диапазоне $\varphi \in [\frac{\pi}{2} - \frac{3\pi}{2}]$	Траектория является полугипоциклоидой
21		Угол изменяется в диапазоне $\varphi \in [\pi - 2\pi]$	Траектория является полугипоциклоидой
22		Угол изменяется в диапазоне $\varphi \in [\frac{3\pi}{2} - \frac{5\pi}{2}]$	Траектория является полугипоциклоидой

Продолжение табл. 31.1

№ варианта траектории	Вид траектории	Уравнение траектории	Примечание
	Эпициклоида 	$x = x_0 + r \cdot (k+1)(\cos(\varphi) - \frac{\cos((k+1)\varphi)}{k+1})$, $y = y_0 + r \cdot (k+1)(\sin(\varphi) - \frac{\sin((k+1)\varphi)}{k+1})$, где $\varphi \in [0 - 2\pi]$ — угол, $k = \frac{R}{r} = 4$, R — радиус неподвижной окружности, r — радиус катящейся окружности, x_0 — координата центра фигуры, y_0 — координата центра фигуры	
23		Угол изменяется в диапазоне $\varphi \in [0 - \pi]$	Траектория является полуэпициклоидой
24		Угол изменяется в диапазоне $\varphi \in \left[\frac{\pi}{2} - \frac{3\pi}{2}\right]$	Траектория является полуэпициклоидой
25		Угол изменяется в диапазоне $\varphi \in [\pi - 2\pi]$	Траектория является полуэпициклоидой
26		Угол изменяется в диапазоне $\varphi \in \left[\frac{3\pi}{2} - \frac{5\pi}{2}\right]$	Траектория является полуэпициклоидой
	Эпициклоида	$x = x_0 + r \cdot (k+1)(\cos(\varphi) - \frac{\cos((k+1)\varphi)}{k+1})$, $y = y_0 + r \cdot (k+1)(\sin(\varphi) - \frac{\sin((k+1)\varphi)}{k+1})$, где $\varphi \in [0 - 2\pi]$ — угол, $k = \frac{R}{r} = 6$, R — радиус неподвижной окружности, r — радиус катящейся окружности, x_0 — координата центра фигуры, y_0 — координата центра фигуры	

Продолжение табл. 31.1

№ варианта траектории	Вид траектории	Уравнение траектории	Примечание
27		Угол изменяется в диапазоне $\varphi \in [0 - \pi]$	Траектория является полуэпициклоидой
28		Угол изменяется в диапазоне $\varphi \in \left[\frac{\pi}{2} - \frac{3\pi}{2}\right]$	Траектория является полуэпициклоидой
29		Угол изменяется в диапазоне $\varphi \in [\pi - 2\pi]$	Траектория является полуэпициклоидой
30		Угол изменяется в диапазоне $\varphi \in \left[\frac{3\pi}{2} - \frac{5\pi}{2}\right]$	Траектория является полуэпициклоидой

Примечание. В приведенных уравнениях все углы измеряются в радианах, а все линейные размеры — в пикселах.

Дополнительные задания для любознательных читателей.

1. Предусмотреть возможность поворота траектории по часовой стрелке и (или) против часовой стрелки. Управление поворотом должен осуществлять пользователь с помощью дополнительного «движка».

2. Предусмотреть возможность произвольного изменения положения центра траектории внутри рабочей области приложения. Траектория не должна выходить за пределы рабочей области формы. Управление перемещением центра траектории можно реализовать с помощью «движков» (по оси «X» и оси «Y») или с помощью устройства «мышь» путем захвата и перетаскивания.

3. Полагаясь на собственную фантазию, предусмотреть любые другие приемы манипуляции с приложением.

4. Построить приложение таким образом, чтобы было легко рисовать в нем траекторию произвольного вида (см. п. 9.2).

31.4. Методические указания

1. В соответствии с объектно-ориентированной технологией проектирования программных изделий рекомендуется вначале разработать диаграмму классов и только после этого приступать к написанию исходного текста программы.

2. Рекомендуемые компоненты:

— **TLabel** (раздел **Standard**) — моделирование надписей;

— **TTrackBar** (раздел **Win32**) — установка значения («движок»);

— **TPanel** (раздел **Standard**) — оформление прямоугольной области на экранной форме для размещения объединенных тематически компонентов, в нашем случае — области элементов управления;

— **TPaintBox** (раздел **System**) — оформление прямоугольной области на экранной форме, преимущественно для целей рисования, в нашем случае — рабочей области.

Компонент **TForm** используется автоматически.

3. Для моделирования траектории рекомендуется использовать метод базовой точки. Метод базовой точки удобно использовать в случае, когда необходимо обеспечить синхронное изменение положения и размеров изображения вслед за изменениями размеров рабочей области или перемещения рисунка по рабочей области. Изображение фигуры в данном случае всегда рисуется относительно некоторой точки, называемой базовой точкой. Масштабирование траектории осуществляется использованием координатной сетки (рисуется на бумаге при проектировании изображения фигуры). Поясним сказанное на примере (рис. 31.2).

Пусть на рисунке 31.2 кружок обозначает центр траектории с координатами (X_0, Y_0). Тогда координата точки 1 будет иметь значение ($X_0 - 4\Delta$, $Y_0 - 3\Delta$), где Δ — размер квадратной клетки. Координата точки 2 будет вычисляться как (X_0, Y_0+2,5Δ). Если теперь изменить значение Δ, то изменится масштаб рисования траектории. Размещение центра фигуры (X_0, Y_0) задает положение траектории в рабочей области моделирования.

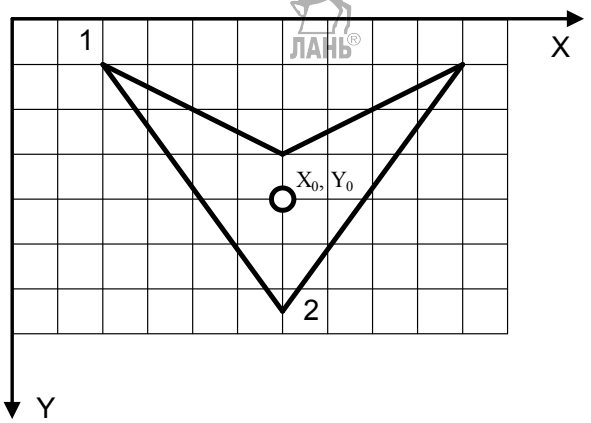
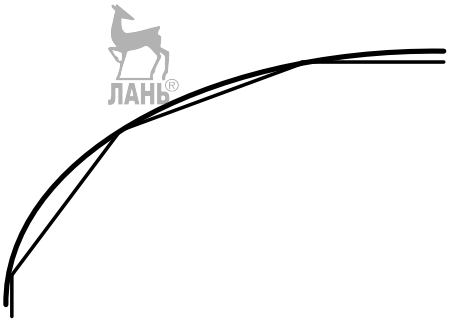


Рис. 31.2
Метод базовой точки

Замечание: расположение осей X, Y координатной сетки на рисунке 31.2 соответствует расположению осей X, Y в визуальных компонентах.

4. Любую кривую линию (траекторию) необходимо представить совокупностью отрезков вида



Размер и количество отрезков подбираются экспериментально таким образом, чтобы дискретность линии сделать на экране незаметной. Если размер отрезка сделать очень маленьким, то время рисования может увеличиться. Прорисовку линии в языке C++Builder удобно осуществлять методами **Polygon(M, k)** или **Polyline(M, k)** стандартного компонента **TCanvas**, где **M** — массив координат точек концов отрезков, **k** — количество используемых точек. В отличие от **Polyline(M, k)** метод **Polygon(M, k)** рисует замкнутую ломаную линию, соединяя дополнительным отрезком первую и последнюю точки в массиве **M**. Координаты точек в массиве **M** задаются стандартным типом данных **TPoint**, который определен в модуле Dialogs.hpp системы программирования и представляет собой пару целых чисел, изображающих координаты точки *X* и *Y*.

5. Рабочую область задачи удобно моделировать компонентом TPaintBox (раздел System), так как замечено, что на этом компоненте прорисовка идет быстрее. В этом случае центром прорисовки траектории будет центр видимой на экране области рассматриваемого компонента. Для несимметричных фигур выбрать центр прорисовки самостоятельно.

6. Для реализации пользовательских действий по масштабированию траектории рекомендуется использовать компонент TTrackBar (раздел Win32). Используются свойства компонента TTrackBar: Min — значение минимальной (левой) позиции, Max — значение максимальной (правой) позиции, Position — значение текущей позиции «движка». Все свойства имеют целый тип данных.

7. Перерисовка клиентской области экранной формы осуществляется в методе OnPaint класса TForm, при этом все действия по перерисовке траектории необходимо выполнять на канве класса TPaintBox. Событие перерисовки вызывать в программе методом Invalidate() класса TWinControl (класс TForm и класс TPaintBox являются потомками класса TWinControl). В этом случае обработка события OnPaint автоматически начинается с очистки клиентской облас-

ти формы. Во многих случаях перерисовка формы вызывается автоматически. Например, в последних версиях RAD Studio изменение положения «движка» приводит к возникновению события OnPaint и форма перерисовывается автоматически.

8. При изменении размеров экранной формы нужно синхронно изменять размеры компонентов TPaintBox и TPanel. Для этого удобно использовать событие OnResize класса TForm. Положение и размеры расположенных на экземпляре класса TPanel элементов управления также при желании можно поменять.

31.5. Рекомендуемая последовательность действий

Описание последовательности действий по выполнению задания 31.2 приводятся из соображений облегчить и ускорить освоение системы программирования RAD Studio. Любознательным читателям, стремящимся дойти до всего самостоятельно, этот раздел можно пропустить. В описании последующих заданий этот раздел отсутствует, однако повторное обращение к приведенному здесь описанию последовательности действий может оказаться полезным.

1. Загрузить систему программирования RAD Studio.
2. Создать новый проект как приложение с экранной формой (выбрать в меню File, New, VCL Form Application — C++Builder).
3. Создать папку для размещения приложения, название папки должно соответствовать выполняемой работе.
4. Сохранить проект и модули главной формы с приданием им названий, соответствующих их смыслу. Рекомендуемое правило именования: для имени проектного файла использовать префикс **p** (например, pOOP_Laboratory_Practice), для имен программных файлов использовать префикс **u** (например, uOOP_Trajectory).
5. Выделить экранную форму, выбрать в инспекторе объектов закладку Properties (свойства), изменить значение свойства **Name** компонента «экранная форма», используя правила именования идентификаторов (например, **frmMain**). Запускать программу до исправления всех ошибок.
6. Для моделирования рабочей области приложения разместить на экранной форме компонент **TPaintBox**. Выделить компонент **TPaintBox** и изменить значение свойства **Name**, используя правила именования идентификаторов (например, **pboxWorkSpace**). Запускать программу до исправления всех ошибок.
7. Разместить на экранной форме компонент **TPanel**. Отрегулировать свойства **Height** (высота) и **Width** (ширина) для размещения на панели компонента **TTrackBar**. Выделить компонент **TPanel** и изменить значение свойства **Name**, используя правила именования идентификаторов (например, **pnlFControlSpace**). Запускать программу до исправления всех ошибок.
8. Выделить компонент **TPanel** и разместить на нем компонент **TTrackBar** («движок»). С помощью мыши отрегулировать положение и размеры «движка». Выделить «движок». Изменить значение свойства **Name**, используя

правила именования идентификаторов (например, **tbarFTrajectorySize**). Установить значения свойств **Min**, **Max** и **Position**. Рекомендуется установить значение **Position** как среднее значение свойств **Min** и **Max** (средний размер траектории). Запускать программу до исправления всех ошибок.

9. Разместить на панели компонент **TLabel**. Выделить компонент **TLabel**. Изменить значение свойства **Name** (например, **lblFTrajectorySize**). Изменить значение свойства **Caption** — ввести надпись: «Размер траектории». С помощью мыши отрегулировать положение и размеры компонента. Запускать программу до исправления всех ошибок.

10. Создать новый программный модуль для моделирования траектории (выбрать в меню File, New, Unit-C++Builder) и сохранить его под понятным именем (например, **uTrajectory**).

11. Ввести в модуль **uTrajectory** описание класса «Траектория» (пусть его имя TTrajectory). Включить в него свойства (property): «Базовая точка», «Масштаб», и свойства, характеризующие выбранный вариант траектории. Использовать для описания базовой точки тип TPoint, а для описания масштаба — вещественный тип данных (double). Включить также в класс метод рисования траектории (пусть он называется Draw). Удобно в заголовок метода рисования траектории включить параметр типа TPaintBox — рабочая область, на канве которой осуществляется рисование траектории. При вызове метода Draw из компонента «экранная форма» будем передавать в качесстве фактического параметра рабочую область приложения. Запускать программу до исправления всех ошибок.

12. Включить в метод **Draw** действия по вычислению: а) параметров траектории, исходя из размеров рабочей области (параметр класса **TPaintBox**) и с учетом изменения масштаба рисования траектории; б) позицию центра траектории; в) массива точек для использования метода **Poligon**, вычисление координат начала и конца каждого отрезка выполнять с помощью параметрических уравнений кривых, приведенных в каждом варианте задания. По завершении вычислений вызвать метод **Poligon**. Запускать программу до исправления всех ошибок.

13. Выделить экранную форму и выбрать в инспекторе объектов закладку Events (события). Ввести метод обработки события **OnCreate** — создания формы. Для этого нужно найти в списке строку OnCreate и щелкнуть два раза мышью в правой части строки. В текст программы автоматически будет включен метод — обработчик данного события. Обработка события **OnCreate** должна содержать: создание объекта «Траектория», установку положения и размеров рабочей области (компонент **TPaintBox**) и панели элементов управления (компонент **TPanel**) в зависимости от текущих размеров клиентской области формы (свойства **ClientHeight** и **ClientWidth**), установку диапазона значений и начальной позиции «движка» (можно оставить без изменения). Запускать программу до исправления всех ошибок.

14. Аналогично шагу 13 выделить экранную форму и ввести метод обработки события **OnPaint**, свидетельствующего о том, что оформление формы на поверхности экрана стало недействительным и форму необходимо перерисо-

вать. Перерисовка самой формы выполняется автоматически, поэтому в метод обработки события **OnPaint** нужно включить только вызов метода рисования траектории класса «Траектория», передав ему в качестве параметра рабочую область приложения (компонент **TPaintBox**). После запуска программы на выполнение мы должны увидеть траекторию на поверхности рабочей области при условии, что все параметры рабочей области и траектории корректны. Запускать программу до исправления всех ошибок.

15. Аналогично шагу 13, выделить экранную форму и ввести метод обработки события **OnResize** — событие изменения размеров формы. Обработка события **OnResize** должна содержать: установку положения и размеров рабочей области (компонент **TPaintBox**) и панели элементов управления (компонент **TPanel**) в зависимости от новых размеров клиентской области формы (свойства **ClientHeight** и **ClientWidth**. При желании можно изменить положение и размер элементов управления, расположенных на компоненте **TPanel**, с учетом новых значений параметров панели. Запускать программу до исправления всех ошибок.

16. Выделить «движок» (компонент **TTrackBar**) и ввести метод обработки события **OnChange** — изменения позиции движка. Обработка события **OnChange** должна содержать вычисление масштаба рисования траектории (значение соответствующего свойства экземпляра класса «Траектория») с учетом нового значения свойства **Position**. Заметим, что в ранних версиях системы программирования при изменении позиции движка событие **OnPaint** не возникает, поэтому его нужно создать принудительно вызовом метода **Invalidate()**. Начиная с версии 2010 года событие недействительности изображения в этом случае создается автоматически. Запускать программу до исправления всех ошибок.

17. Окончательно протестировать сформулированные в задании 31.1 все возможности пользователя по управлению приложением.

Разработка программы закончена. Примите поздравление!

Задание 32. Программирование движения

32.1. Описание приложения. В рабочей области приложения расположена траектория. Пользователь имеет возможность изменять размер формы, размер траектории и завершать работу приложения (см. задание 31).

По траектории с постоянной скоростью перемещается точка. В области элементов управления расположен «движок» «Скорость движения точки», с помощью которого пользователь может изменить скорость движения точки по траектории.

32.2. Задание. Взять программу задания 31 и расширить ее следующими возможностями:

1. По траектории с постоянной скоростью движется точка.
2. С помощью «движка» можно изменить скорость движения точки по траектории.

Требования:

1. Скорость перемещения точки по траектории никоим образом не связана со скоростью прорисовки кадров, а определяется величиной шага смещения центра фигуры в методе базовой точки (чем больше шаг, тем выше скорость).

2. Для решения этой задачи в программе предусмотреть отдельный класс «Движущаяся по траектории точка», размещенный в отдельном программном модуле.

3. Необходимо использовать правила именования идентификаторов.

4. В разделе **public** описания класса «Движущаяся по траектории точка» должны находиться только описания свойств (**property**) и методов.

32.3. Варианты задания остаются такими же, как в задании 31.

Дополнительные задания для любознательных читателей.

1. Предусмотреть возможность выбора направления движения точки: по часовой стрелке или против часовой стрелки. Управление направления движения должен осуществлять пользователь, например, с помощью компонента **TCheckBox**.

2. Предусмотреть возможность независимого (зависимого) движения по траектории нескольких точек.

3. Построить родовой тип «Траектория с движущейся точкой», который позволит легко обрабатывать любой вариант траектории из задания 31. Использовать инструмент шаблонов, рассмотренный в параграфе 9.4.

32.4. Методические указания

1. В соответствии с объектно-ориентированной технологией проектирования программных изделий рекомендуется вначале разработать диаграмму классов и только после этого приступать к написанию исходного текста программы.

2. Класс «Движущаяся по траектории точка» является потомком класса «Траектория».

3. Движущуюся точку рекомендуется прорисовывать с помощью метода

void Ellipse(int X1, int Y1, int X2, int Y2)

компонента **TCanvas**, где (**X1, Y1), (X2, Y2)** — координаты, соответственно, левой верхней и правой нижней точек прямоугольника, в который вписывается эллипс.

4. Рекомендуемые дополнительные компоненты:

TTimer (раздел System) — датчик времени.

5. Управление движением осуществляется с помощью компонента **TTimer**, который позволяет задавать в приложении интервалы времени. Таймер — невизуальный компонент — может размещаться во время проектирования в любом месте экранной формы. Он имеет два свойства, позволяющие им управлять: **Interval** — интервал времени в миллисекундах, и **Enabled** — доступность. Свойство **Interval** задает период срабатывания таймера после

предыдущего срабатывания таймера. Когда таймер срабатывает, возникает событие **OnTimer**. Если **Interval** равен 0 или **Enabled** равно **False**, то таймер перестает работать. Чтобы запустить отсчет времени, надо или задать **Enabled** равным **True**, если установлено положительное значение интервала, или задать положительное значение **Interval**, если **Enabled** равно **True**. В процедуру обработки события **OnTimer** (конец интервала таймера) необходимо включить обработку одного шага движения точки по траектории.

6. При использовании таймера для перемещения точки по траектории с целью обеспечения благоприятного для восприятия эффекта мультипликации (непрерывного движения со скоростью прорисовки 25 кадров в секунду) интервал таймера рекомендуется принять равным 40 мс. Когда движение точки будет видно на экране, можно поэкспериментировать с величиной интервала таймера.

7. В обработчик события **OnTimer** рекомендуется включить следующие действия: а) продвижение точки по траектории на некоторый угол; б) вызов метода **Invalidate()**, который создает событие недействительности изображения формы с последующей отработкой события **OnPaint**. Прорисовку точки (эллипса) рекомендуется включать в обработчик события **OnPaint**. Событие недействительности изображения создается путем вызова метода **Invalidate()** класса **TWinControl**. Метод **Invalidate()** доступен как в компоненте **TForm**, так и в компоненте **TPaintBox**.

8. Управление скоростью движения точки целесообразно осуществлять увеличением или уменьшением расстояния, на которое перемещается точка за один шаг движения. Расстояние между двумя смежными точками вычислять как функцию угла, заданного между центром траектории и двумя рассматриваемыми на траектории точками. Таким образом, величина приращения угла за один такт таймера является синонимом понятия «скорость движения точки». Для моделирования текущего угла и угла приращения использовать вещественный тип данных, так как углы измеряются радианами.

9. В вариантах траектории имеется два участка: линейный и криволинейный. Естественно, что перемещение точки по каждому виду участков происходит по разным правилам. Если перемещение по криволинейному участку очевидно является функцией рассмотренного угла, то использование угла для расчета движения точки по линейному участку менее очевидно, но очень желательно.

Задание 33. Программирование анимации

33.1. Описание приложения. В рабочей области приложения расположена траектория. Пользователь имеет возможность изменять размер формы, размер траектории и завершать работу приложения (см. задание 31).

По траектории с постоянной скоростью перемещается геометрическая фигура. В области элементов управления расположен «движок» «Скорость движения фигуры», с помощью которого пользователь может изменить скорость движения фигуры по траектории.

33.2. Задание. Взять программу задания 32 и расширить ее следующими возможностями:

1. По траектории движется с постоянной скоростью фигура.
2. С помощью «движка» можно изменить скорость движения фигуры по траектории.

Требования:

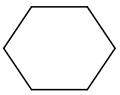
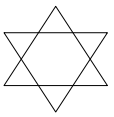
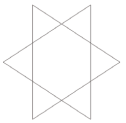
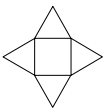
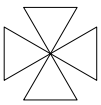
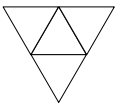
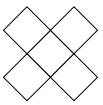
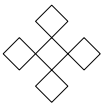
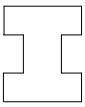
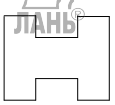
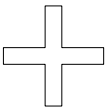
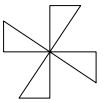
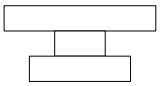
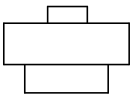
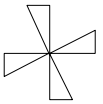
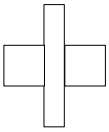
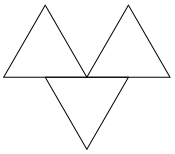
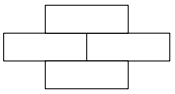
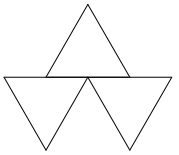
1. Для моделирования фигуры в программе предусмотреть отдельный класс «Фигура», размещенный в отдельном программном модуле.
2. В разделе **public** описания класса «Фигура» должны находиться только описания свойств (**property**) и методов.

33.3. Варианты заданий приведены в таблице 33.1.

ЛАНЬ®

Таблица 33.1

Варианты заданий

 1	 2	 3	 4
 5	 6	 7	 8
 9	 10	 11	 12
13	 14	 15	 16
 17	 18	 19	 20

Дополнительные задания для любознательных читателей.

1. Предусмотреть возможность изменения размера фигуры.
2. Опираясь на собственную фантазию, предусмотреть манипуляции с фигурой.
3. Предусмотреть возможность движения по траектории нескольких фигур.
4. Предусмотреть возможность рисования потенциально произвольной фигуры.

33.4. Методические указания.

1. В первую очередь рекомендуется доработать диаграмму классов из задания 32.
2. Для прорисовки фигуры рекомендуется использовать метод базовой точки, описание которого приведено в методических рекомендациях к заданию 31. В качестве базовой точки взять точку из задания 32, заменив рисование точки в виде эллипса рисованием фигуры.
3. Шаг движения фигуры осуществляется путем перемещения позиции базовой точки фигуры в другое место: погасить изображение фигуры, передвинуть базовую точку, высветить изображение фигуры. При использовании события **OnPaint** заботиться об удалении старого изображения фигуры не нужно, так как это действие выполняется автоматически при перерисовке всей экранной формы.
4. Для удобства дальнейшего развития приложения (см. задание 36) рекомендуется смоделировать абстрактную фигуру. Для этого вводится массив характерных точек варианта фигуры. Характерные точки выбираются из выбранного способа рисования фигуры с помощью отрезков прямых линий (см. главу 6). Необходимо выбрать положение каждой точки относительно координаты центра фигуры (0,0) исходя из среднего размера фигуры. Средний размер фигуры выбирается относительно размера рабочей области и размеров траектории. Неудачный выбор размера абстрактной фигуры можно будет исправить, когда фигуру увидим на экране.
5. Для рисования фактической фигуры в методе рисования вводится массив фактических точек фигуры, каждая точка которого рассчитывается с помощью массива абстрактной фигуры относительно текущего положения базовой точки фигуры и коэффициентов масштабирования.
6. Рисование линий выполняется с помощью свойства **Pen** (перо) класса **TCanvas**, описанного в параграфе 12.6.

Рисование фактической фигуры выполнять с помощью методов класса **TCanvas**:

virtual void __fastcall LineTo(int X, int Y),
virtual void __fastcall MoveTo(int X, int Y).

Метод **LineTo** передвигает перо в «опущенном состоянии» от текущего положения в точку (X, Y). Метод **MoveTo** передвигает перо в «поднятом состоянии» от текущего положения в точку (X, Y).

Для каждого варианта фигуры необходимо назначить начальную точку и спланировать последовательность рисования линий.

Задание 34. Программирование анимации с несколькими движениями

34.1. Описание приложения. В рабочей области приложения расположена траектория. Пользователь имеет возможность изменять размер формы и размер траектории (см. задание 31).

По траектории с постоянной скоростью перемещается геометрическая фигура. В области элементов управления расположен «движок» «Скорость движения фигуры», с помощью которого пользователь может изменить скорость движения фигуры по траектории (см. задание 33).

Дополнительно движущаяся фигура постоянно изменяет свой масштаб («дышит»). Пользователь имеет возможность изменить скорость «дыхания» фигуры.

34.2. Задание. Взять программу из задания 33 и расширить ее возможностью регулярного изменения масштаба изображения фигуры — «дыхания». Предусмотреть регулятор скорости масштабирования («дыхания»).

Требования:

1. В разделе **public** описания класса «Фигура» должны находиться только описания свойств (**property**) и методов.
2. Необходимо использовать правила именования идентификаторов.

34.3. Варианты заданий — нет.

Дополнительные задания для любознательных читателей.

1. Предусмотреть возможность вращения фигуры и регулировки скорости вращения фигуры.

34.4. Методические указания.

1. Управление анимацией осуществляется с помощью компонента **TTimer**. В процедуру обработки события **OnTimer (**конца интервала таймера) кроме вычисления новой позиции фигуры, необходимо добавить вычисление новых размеров фигуры.

2. Плавность «дыхания» фигуры можно обеспечить, используя для вычисления текущих размеров фигуры функцию $\mathrm{Sin}(\alpha)$ или функцию $\mathrm{Cos}(\alpha)$. Переменная α на каждом шаге анимации будет получать приращение, величина которого определяет скорость «дыхания» и может изменяться при управлении скоростью «дыхания». Единица измерения аргумента тригонометрических функций выражается в радианах.

3. Для управления несколькими движениями использовать единственный таймер.

Задание 35. Программирование цветового оформления экранной области

35.1. Описание приложения. В рабочей области приложения расположена траектория. Пользователь имеет возможность изменять размер формы и размер траектории (см. задание 31).

По траектории с постоянной скоростью перемещается геометрическая фигура. В области элементов управления расположен «движок» «Скорость движения фигуры», с помощью которого пользователь может изменить скорость движения фигуры по траектории (см. задание 33).

Движущаяся фигура постоянно изменяет свой масштаб («дышит»). Пользователь имеет возможность изменить скорость «дыхания» фигуры (см. задание 34).

Дополнительно пользователь имеет возможность:

а) изменить цвет рисования траектории;

б) изменить цвет рисования фигуры;

в) изменить цвет фона рабочей области.

35.2. Задание. Взять программу из задания 34 и расширить ее возможностями управления цветовым оформлением рабочей области приложения. Предусмотреть средства пользовательского выбора цветов рисования траектории, фигуры, закрашивания фона рабочей области приложения.

Требования:

1. Необходимо использовать правила именования идентификаторов.

Дополнительные задания для любознательных читателей.

1. Фигура переливается всеми цветами радуги.

35.3. Варианты заданий — нет.

35.4. Методические указания.

1. Рекомендуемые компоненты:

— **TButton** (раздел **Standard**) — кнопка выбора цвета;

— **TColorDialog** (раздел **Dialogs**) — включение диалога выбора цвета;

— **TPaintBox** (раздел **System**) с внутренним компонентом **TCanvas** — оформление области рисования.

2. Рисование осуществляется с помощью компонента **TCanvas**. В работе для этой цели используется экземпляр класса **TCanvas**, входящий в компонент **TPaintBox**.

Класс **TCanvas** (канва) является основой графической подсистемы C++Builder. Канва обеспечивает:

a) загрузку и хранение графических изображений;

b) создание новых и изменение хранимых изображений с помощью пера, кисти, шрифта;

c) рисование и закраску различных фигур, линий, текстов;

d) комбинирование различных изображений.

3. Цвет визуального объекта задается с помощью типа **TColor**. Применяется в свойствах **Color** многих компонентов.

Тип **TColor** определен в модуле стандартной поставки **Graphics.hpp**:

enum TColor {clMin=-0x7fffffff-1, clMax=0x7fffffff};.

Для удобства работы с значениями цветов в модуле **Graphics.hpp** введены соответствующие константы. Для примера приведем несколько определений:

```
static const TColor clBtnFace=-16777201;      // цвет поверхности кнопок
static const TColor clGreen=32768;            // зеленый цвет
static const TColor clOlive=32896;            // оливково-зеленый цвет
```

В таблице 35.1 приведены обозначения, числовые значения и расшифровки значения цвета наиболее часто используемых цветов.

Таблица 35.1

Таблица цветов

Константа	Числовое значение	Значение цвета
clBlack	0	Черный
clMaroon	128	Темно-бордовый
clGreen	32768	Зеленый
clOlive	32896	Оливково-зеленый
clNavi	8388608	Темно-синий
clPurple	8388736	Пурпурный
clTeal	8421376	Морской волны
clGray	8421504	Серый
clSilver	12632256	Серебряный
clRed	255	Красный
clLime	65280	Лимонно-зеленый
clBlue	16711680	Синий
clWhite	16777215	Белый

4. В составе канвы **TCanvas** имеется свойство **Brush** (кисть), которое является экземпляром компонента **TBrush**. Кисть используется для закрашивания поверхности, ограниченной замкнутой линией. Закрашивание выполняется автоматически при рисовании замкнутой линии, при этом используется цвет, заданный свойством **Color** кисти.

5. Рисование и цветовое оформление линий выполняется с помощью компонента **Pen** (перо), которое является свойством компонента **TCanvas**. Закрашивание заданным цветом осуществляется автоматически во время рисования линий. Здесь также используется свойство пера **Color**.

6. Для программирования пользовательского выбора цвета удобно воспользоваться компонентом **TColorDialog**, который необходимо разместить на экранной форме в процессе проектирования приложения. В составе этого компонента нам понадобятся свойство **Color** и метод **Execute()**. При вызове метода **Execute()** на экране воспроизводится диалог выбора цвета. Выбранный пользователем цвет запоминается в свойстве **Color** компонента **TColorDialog**. По окончании диалога результат пользовательского выбора цвета необходимо присвоить свойству **Color** кисти или пера.

7. Кисть заполняет цветом только замкнутые области рисунка. Поэтому для закрашивания рабочей области приложения необходимо задать значение свойства **Color** кисти, а затем нарисовать прямоугольник по контуру рабочей области с помощью метода канвы

virtual void __fastcall Rectangle(int X1, int Y1, int X2, int Y2).

Параметрами функции **Rectangle** являются точки (**X1, Y1**), (**X2, Y2**), соответственно, левой верхней и нижней правой точек прямоугольника.

8. Как и в предыдущих заданиях, канва рабочей области приложения передается параметром в методы рисования и окрашивания элементов нашего приложения: траектории, фигуры, рабочей области.

9. Делом чести является не оставлять измененные цветовые параметры, поэтому изменение цветовых параметров нужно выполнять следующими действиями:

а) запомнить существующее значение цвета;

б) изменить значение цвета на нужное;

в) выполнить необходимую прорисовку;

г) восстановить исходное значение цвета.

Задание 36. Программирование обмена текстовой информацией с приложением

36.1. Описание приложения. В рабочей области приложения расположена траектория. Пользователь имеет возможность изменять размер формы и размер траектории (см. задание 31).

По траектории с постоянной скоростью перемещается геометрическая фигура. В области элементов управления расположен «движок» «Скорость движения фигуры», с помощью которого пользователь может изменить скорость движения фигуры по траектории (см. задание 33).

Движущаяся фигура постоянно изменяет свой масштаб («дышит»). Пользователь имеет возможность изменить скорость «дыхания» фигуры (см. задание 34).

Имеются средства управления цветом рисования траектории, фигуры, фона рабочей области (см. задание 35).

Дополнительно пользователь имеет возможность редактировать фигуру с помощью текстового поля.

36.2. Задание. Взять программу из задания 35 и расширить ее средствами редактирования фигуры из текстового поля.

Требования:

1. При окончании редактирования фигуры в текстовом поле изображение фигуры должно соответственно изменяться.

2. Необходимо использовать правила именования идентификаторов.

Дополнительные задания для любознательных читателей.

1. В данной работе редактирование фигуры осуществляется без изменения количества вершин фигуры (строк в текстовом поле). Предусмотреть возможность изменения количества вершин фигуры (строк в текстовом поле). Появляется возможность использования фигур произвольного вида. Введение этой возможности связано с использованием динамических массивов, реализация которых может быть выполнена с помощью типа данных **vector** из стандартной библиотеки STL. Имеется еще одна возможность использования динамических массивов — класс **TStringList**.

2. Предусмотреть средства контроля правильности вводимых данных.

36.3. Варианты заданий. Отсутствуют.

36.4. Методические указания.

1. Для редактирования фигуры необходимо разработать формат текста, например, для фигуры «квадрат» формат текста может быть следующим:

Фигура квадрат
100, 100
200, 100
200, 200
100, 200

Первой строкой, как показано в этом примере, может быть имя фигуры, а в следующих строках координаты вершин квадрата (в пикселах).

2. Моделирование текстового поля в программе рекомендуется осуществлять визуальным компонентом **TMemo** (раздел **Standard**). Компонент **TMemo** содержит свойство **Lines** класса **TStrings**. Экземпляр класса **TStrings** представляет собой массив строк типа **String**. Методы и свойства класса **TStrings** позволяют манипулировать со списками строк:

— добавлять и удалять строки в указанных позициях;
— перестраивать и упорядочивать последовательности строк;
— получать доступ к конкретным строкам;
— читать и записывать списки строк в файлы и потоки;
— связывать с каждой строкой некоторый объект.

Нам понадобятся свойства класса **TStrings**: свойство **Count** целого типа — количество строк в массиве (свойство только для чтения), свойство **Strings**, с помощью которого мы получаем доступ к строкам, например, **Memo1–>Lines–>Strings[3]** — обеспечивает доступ к строке 4 поля **Memo1** (как обычно, нумерация строк начинается с нуля).

Для работы со строками используются методы:

int Add(const AnsiString S)	Добавляет строку в конец списка. Возвращает индекс добавленной строки
void Append(const AnsiString S)	Добавляет строку в конец списка
void Clear(void)=0	Очищает список. Этот метод имеется в компонентах **Lines** и **Strings**. Поэтому можно использовать **Memo1–>Lines–>Clear()** и **Memo1–>Lines–>Strings–>Clear()**

3. При распознавании строк текста удобно сначала вычислить позицию запятой в строке, например, для выделения позиции символа «,» в строке 0 можно использовать оператор присваивания вида

n=Memo1–>Lines–>Strings[0].Pos(«,»);

где **n** — переменная целого типа. Далее содержащую число строку можно выделить с помощью оператора вида **String s=Memo1–>Lines–> Strings[0].SubString(1, n–1)**. В функции **SubString()** первый параметр задает начальную позицию подстроки, а второй — количество символов в выделяемой подстроке.

4. Для преобразования строки в целое число используется метод **StrToInt(s)**, где *s* — строка в формате целого числа. Метод даст сбой, если формат

целого числа не выдержан: таким примером может быть строка « –100 », в которой содержатся лишние пробелы. Удалить слева и справа от числа пробелы можно спомощью функции **Trim(s)**, где *s* — исходная строка. Функция возвращает строку с удаленными слева и справа пробелами.

5. Рекомендуется использовать два вида координат: абстрактные и фактические (см. методические указания к заданию З3). Абстрактные координаты вычисляются на основе некоторых абстрактных размеров фигуры и не изменяются в процессе работы программы. Фактические координаты фигуры изменяются в процессе выполнения программы и вычисляются с помощью координатной сетки при каждом выводе фигуры в рабочую область с учетом текущих значений масштабов изображения. При запуске программы абстрактные координаты нужно вывести в поле редактирования фигуры.

6. При изменении пользователем координат точек необходимо распознать новые значения координат абстрактной фигуры и ввести их значения в массив абстрактной фигуры. При соблюдении методических рекомендаций задания З3, дальнейшая прорисовка новой конфигурации фигуры выполнится уже разработанными средствами.

7. Ввод изменений параметров фигуры рекомендуется инициализировать с помощью специально предусмотренной кнопки класса **TButton**.

8. Абстрактные и фактические координаты необходимо размещать в разных массивах точек, которые вводятся в классе «фигура». Вычисление значений каждого массива удобно размещать в отдельных методах.

Задание З7. Программирование взаимодействия программы с окружением

З7.1. Описание приложения. В рабочей области приложения расположена траектория. Пользователь имеет возможность изменять размер формы и размер траектории (см. задание З1).

По траектории с постоянной скоростью перемещается геометрическая фигура. В области элементов управления расположен «движок» «Скорость движения фигуры», с помощью которого пользователь может изменить скорость движения фигуры по траектории (см. задание З3).

Движущаяся фигура постоянно изменяет свой масштаб («дышит»). Пользователь имеет возможность изменить скорость «дыхания» фигуры (см. задание З4).

Пользователь может изменить цвет рисования траектории, цвет рисования фигуры, цвет фона рабочей области (см. задание З5).

Пользователь имеет возможность редактировать фигуру с помощью текстового поля (см. задание З6).

Дополнительно пользователь имеет возможность:

а) сохранять фигуру в буфере обмена;

б) восстанавливать фигуру из буфера обмена.

З7.2. Задание. Взять программу из задания З6 и расширить ее следующими средствами:

1) предусмотреть возможность сохранения фигуры в буфере обмена;
2) предусмотреть возможность восстановления фигуры из буфера обмена.

Требования:

1. Операции с буфером обмена выполнять в текстовом формате.
2. Необходимо использовать правила именования идентификаторов.

Дополнительные задания для любознательных читателей.

1. В данной работе предусматривается выполнение операций с буфером обмена в стандартном текстовом формате. Работа в этом режиме настолько автоматизирована, что стала неинтересной. Предусмотреть возможность регистрации в буфере обмена собственного формата и выполнения операций с буфером обмена в зарегистрированном формате.

37.3. Варианты заданий отсутствуют.

37.4. Методические указания. Буфер обмена является очень распространенным инструментом при работе в операционной системе Windows для обмена данными как между компонентами приложения, так для обмена данными между приложениями. Поэтому мы просто обязаны предоставить такую возможность пользователям создаваемых нами программ.

Операционная система Windows поддерживает около 25 стандартных форматов обмена информацией с буфером обмена, причем использование этих форматов в значительной мере автоматизировано. В буфер обмена можно помещать текст, объекты растровой графики и множество других программных компонентов.

1. Перед выполнением задания рекомендуется просмотреть параграф 12.12.
2. Для выполнения операций с буфером обмена рекомендуется использовать компонент **TClipboard**. Описание невизуального компонента **TClipboard** находится в пространстве имен **Clipbrd.hpp**.
3. Для доступа к компоненту **TClipboard** необходимо сделать доступным пространство имен **Clipbrd.hpp**. В отличие от многих других компонентов буфер обмена нельзя перенести из палитры компонентов на форму. Поэтому буфер обмена создается в программе обычным для объектов способом: после объявления объектной переменной нужно создать объект вызовом конструктора компонента.
4. Для работы с текстовым форматом имеется несколько приемов (см. п. 12.12). В нашем случае удобно использовать методы **SetTextBuf**, **GetTextBuf**. Можно также использовать метод **AsText**.

Задание 38. Сохранение и воостановление состояния приложения

38.1. Описание приложения. В рабочей области приложения расположена траектория. Пользователь имеет возможность изменять размер формы и размер траектории (см. задание 31).

По траектории с постоянной скоростью перемещается геометрическая фигура. В области элементов управления расположен «движок» «Скорость движения фигуры», с помощью которого пользователь может изменить скорость движения фигуры по траектории (см. задание 33).

Движущаяся фигура постоянно изменяет свой масштаб («дышит»). Пользователь имеет возможность изменить скорость «дыхания» фигуры (см. задание 34).

Пользователь может изменить цвет рисования траектории, цвет рисования фигуры, цвет фона рабочей области (см. задание 35).

Пользователь имеет возможность редактировать фигуру с помощью текстового поля (см. задание 36).

Пользователь имеет возможность выполнять операции с буфером обмена над фигурой (см. задание 37).

Дополнительно программа имеет возможность:

а) при завершении сохранять настройки интерфейса;

б) при запуске восстанавливать настройки интерфейса предыдущего сеанса.

38.2. Задание. Взять программу из задания 37 и расширить ее следующими средствами:

— организовать сохранение при завершении работы программы и восстановление при запуске программы настроек интерфейса приложения: размеров и положения формы приложения, размеров траектории и фигуры, скорости движения и «дыхания» фигуры, цвета траектории, фигуры и фона рабочей области.

Требования:

1. После выполнения операций сохранения и восстановления настроек интерфейса состояние экранной формы приложения должно оставаться идентичным.
2. При завершении приложения настройки интерфейса должны сохраняться автоматически.
3. При запуске приложения сохраненные настройки интерфейса должны восстанавливаться автоматически.
4. Необходимо использовать правила именования идентификаторов.

38.3. Варианты заданий отсутствуют.

38.4. Методические указания.

1. Перед выполнением задания рекомендуется просмотреть параграф 12.13.
2. Для сохранения и восстановления настроек интерфейса предлагается использовать компонент **TIniFile** (см. п. 12.13). **TIniFile** не является визуальным компонентом, и поэтому мы его не найдем в Палитре компонентов (Tool Palette). Описание типа данных **TIniFile** находится в пространстве имен **IniFiles.hpp**, возможность использования **TIniFile** достигается включением в программный модуль строки

```
#include <IniFiles.hpp>
```

3. Действия по сохранению настроек интерфейса рекомендуется включать в метод обработки события **OnClose** — событие закрытия формы. В **Ini**-файле рекомендуется создать три раздела: рабочая область, траектория и фигура.

4. Действия по восстановлению настроек интерфейса включать в метод обработки события формы **OnCreate** — событие создания формы.

Задание 39. Обработка сообщений Windows

39.1. Описание приложения. В рабочей области приложения расположена траектория. Пользователь имеет возможность изменять размер формы и размер траектории (см. задание 31).

По траектории с постоянной скоростью перемещается геометрическая фигура. В области элементов управления расположен «движок» «Скорость движения фигуры», с помощью которого пользователь может изменить скорость движения фигуры по траектории (см. задание 33).

Движущаяся фигура постоянно изменяет свой масштаб («дышит»). Пользователь имеет возможность изменить скорость «дыхания» фигуры (см. задание 34).

Пользователь может изменить цвет рисования траектории, цвет рисования фигуры, цвет фона рабочей области (см. задание 35).

Пользователь имеет возможность редактировать фигуру с помощью текстового поля (см. задание 36).

Пользователь имеет возможность выполнять операции с буфером обмена над фигурой (см. задание 37).

Программа имеет возможность сохранять и восстанавливать настройки интерфейса (см. задание 38).

Дополнительно программа отображает событие изменения содержимого буфера обмена.

39.2. Задание. Взять программу из задания 38 и расширить ее следующими средствами:

— предусмотреть возможность перехвата сообщения **WM_DRAWCLIPBOARD** (изменение содержимого буфера обмена).

Требования:

1. Если из буфера обмена восстанавливается фигура, то нужно отобразить новое содержимое буфера в поле редактирования фигуры и новую фигуру в окне прорисовки.
2. Необходимо использовать правила именования идентификаторов.

39.3. Варианты заданий. Отсутствуют.

39.4. Методические указания. Перед выполнением задания рекомендуется повторить материал главы 10 «Обработка событий и сообщений».

При занесении любой программой в буфер обмена каких-то данных генерируется сообщение Windows **WM_DRAWCLIPBOARD**, не имеющее параметров. Это сообщение поступает только в те окна, которые включены в цепочку (список) окон просмотра буфера обмена (рис. 39.1). Чтобы получать сообщения от буфера обмена, наша программа должна заявить операционной системе об этом, и, если вдруг настанет такой момент, когда мы больше не хотим получать сообщения от буфера обмена (например: закрытие приложения), мы должны отказаться вызовом стандартной функции.

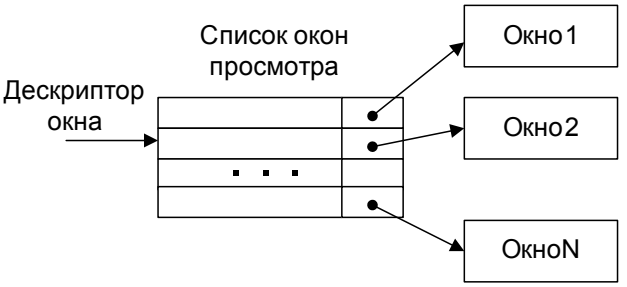


Рис. 39.1
Организация рассылки сообщения **WM_DRAWCLIPBOARD**

Включить свое окно в список окон просмотра буфера обмена можно функцией **SetClipboardViewer**:

HWND SetClipboardViewer(HWND hWndNewViewer).

Параметр **hWndNewViewer** является дескриптором окна, включаемого в цепочку. Функция возвращает дескриптор следующего окна в цепочке. Дело в том, что если мы включили свою форму в цепочку окон просмотра буфера обмена, то должны обеспечить обработку сообщений **WM_DRAWCLIPBOARD** и **WM_CHANGECBCHAIN**. Сообщение **WM_CHANGECBCHAIN** генерируется при изменениях в цепочке окон просмотра буфера. В конце обработчиков этих сообщений надо функцией **SendMessage** передать то же сообщение следующему окну цепочки, дескриптор которого вам вернула функция **SetClipboardViewer**.

Перед завершением приложения надо исключить свое окно из цепочки. Это делается функцией **ChangeClipboardChain**:

BOOL ChangeClipboardChain(IN HWND hWndRemove, IN HWND hWndNewNext).

Параметр **hWndRemove** — дескриптор удаленного окна, а параметр **hWndNewNext** — дескриптор следующего окна в цепочке.

Описание карты сообщений должно содержать информацию, по крайней мере, о двух сообщениях: **WM_DRAWCLIPBOARD** и **WM_CHANGECBCHAIN**. Оба сообщения не нуждаются в параметрах, поэтому в качестве имени структуры параметра можно использовать структуры **TWMNoParams**.

Вызов функции **SetClipboardViewer** целесообразно включить в конструктор формы или в обработчик события формы **OnCreate**. Вызов функции **ChangeClipboardChain** можно сделать в деструкторе формы или в обработчиках событий **OnDestroy** или **OnClose**.

Дескриптор экранной формы содержится в свойстве **Handle**.

Для организации процесса обработки сообщений использовать карту сообщений.

Задание 310. Обработка ошибок

310.1. Описание приложения. В рабочей области приложения расположена траектория. Пользователь имеет возможность изменять размер формы и размер траектории (см. задание 31).

По траектории с постоянной скоростью перемещается геометрическая фигура. В области элементов управления расположен «движок» «Скорость движения фигуры», с помощью которого пользователь может изменить скорость движения фигуры по траектории (см. задание 33).

Движущаяся фигура постоянно изменяет свой масштаб («дышит»). Пользователь имеет возможность изменить скорость «дыхания» фигуры (см. задание 34).

Пользователь может изменить цвет рисования траектории, цвет рисования фигуры, цвет фона рабочей области (см. задание 35).

Пользователь имеет возможность редактировать фигуру с помощью текстового поля (см. задание 36).

Пользователь имеет возможность выполнять операции с буфером обмена над фигурой (см. задание 37).

Программа имеет возможность сохранять и восстанавливать настройки интерфейса (см. задание 38).

Программа отображает событие изменения содержимого буфера обмена (см. задание 39).

Дополнительно имеются средства обработки исключительных ситуаций операций ввода текста.

310.2. Задание. Взять программу из задания 39 и расширить ее следующими средствами:

— с помощью класса **EConvertError** организовать контроль ввода данных редактирования фигуры из поля TMemo.

Требования:

Необходимо использовать правила именования идентификаторов.

310.3. Варианты заданий отсутствуют.

310.4. Методические указания.

1. Перед выполнением задания рекомендуется повторить материал главы 11 «Обработка ошибок».

2. В процессе преобразования вводимых данных редактирования фигуры используется операция **StrToInt()**, которая контролирует соответствие параметра операции формату целого числа. Если параметр операции не является целым числом, то возбуждается исключительная ситуация **EConvertError**. Обращаем внимание, что функция **StrToInt** чувствительна к пробелам, т. е. присутствие пробела в любом месте текстового поля числа классифицируется как ошибка.

3. Защитить от ошибок код ввода и преобразования данных можно с помощью конструкции try-catch с выдачей сообщения об ошибке.

4. Обычно об обнаруженных ошибках сообщают пользователю с помощью функции API **MessageBox**, описание которой приведено в п. 5.3.1.

ПРИЛОЖЕНИЕ 1. ИСТОРИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

История объектно-ориентированного программирования, по-видимому, еще не получила серьезного научного осмысления. Существует множество различных взглядов на развитие этого направления в программировании. Единственное, что не оспаривается: объектно-ориентированное программирование стало заметным этапом и, может быть, даже эпохой в развитии профессии. Мы рассмотрим два взгляда на историю объектно-ориентированного программирования. Источником информации послужил Интернет.

П1.1. Историческая справка 1[49]

Языки программирования. Пожалуй, наиболее важной вехой в истории программирования, сравнимой по значимости разве что с изобретением письменности, можно считать переход от машинных кодов (тарабарщины типа 0110110101111...) к понятным простому смертному языкам программирования (типа ALGOL, FORTRAN, PL/1, Pascal), а также к широкому использованию методов структурного программирования. Программы стали модульными, состоящими из подпрограмм. Появились библиотеки готовых подпрограмм, облегчающие многие задачи, но все равно программистам хватало трудностей, особенно при разработке пользовательского интерфейса.

Объектно-ориентированное программирование. Качественным шагом в развитии методов структурного программирования стало изобретение объектно-ориентированного программирования (языков SmallTalk, C++, Turbo Pascal и др.). Программы стали строиться не из чудовищных по размеру процедур и функций, перерабатывающих громоздкие структуры данных, а из сравнительно простых кирпичиков-объектов, в которых были упрятаны данные и подпрограммы их обработки. Гибкость объектов позволила очень просто приспосабливать их для собственных целей, прилагая для этого минимум усилий. Программисты обзавелись готовыми библиотеками объектов, но, как и раньше, создание пользовательского интерфейса требовало уйму времени и сил, особенно когда программа должна была работать под управлением популярной операционной системы Windows и иметь графический пользовательский интерфейс.

Визуальное программирование. С изобретением визуального программирования, первой ласточкой которого была среда разработки Visual Basic, создание графического пользовательского интерфейса стало под силу даже новичку. В среде Visual Basic можно было быстро создать приложение для операционной системы Windows, в котором были все присущие графическому пользовательскому интерфейсу элементы: окна, меню, кнопки, поля ввода и т. д.

[49] Источник: К. Сурков, А. Вальвачев, Д. Сурков, Ю. Четырько. Учебное пособие по программированию на языке Delphi.

Все эти элементы превратились в строительные блоки программы — компоненты-объекты, имеющие визуальное представление на стадии проектирования и во время работы.

Проектирование пользовательского интерфейса упростилось на порядок, однако для профессиональных программистов язык Basic оказался явно слабоват. Отсутствие в нем контроля типов данных и механизма их расширения оказалось камнем преткновения на пути создания серьезных программ. Создание нестандартных компонентов в среде Visual Basic было крайне затруднено (для этого приходилось прибегать к другим средствам разработки, в частности к языку C++). В общем, среда Visual Basic отлично подходила для создания прототипов приложений, но не для разработки коммерческих программных продуктов.

Среда программирования Delphi. Мечта программистов о среде программирования, в которой бы простота и удобство сочетались с мощью и гибкостью, стала реальностью с появлением среды Delphi. Она обеспечивала визуальное проектирование пользовательского интерфейса, имела развитый объектно-ориентированный язык Object Pascal (позже переименованный в Delphi) и уникальные по своей простоте и мощи средства доступа к базам данных. Язык Delphi по возможностям значительно превзошел язык Basic и даже в чем-то язык C++, но при этом он оказался весьма надежным и легким в изучении (особенно в сравнении с языком C++). В результате, среда Delphi позволила программистам легко создавать собственные компоненты и строить из них профессиональные программы. Среда оказалась настолько удачной, что по запросам любителей C++ была позже создана среда C++Builder — клон среды Delphi на основе языка C++ (с расширенным синтаксисом).

Среда Delphi стала, по сути, лучшим средством программирования для операционной системы Windows, но программистов ждало разочарование, если возникало желание перенести программу в другую операционную систему, в частности в операционную систему Unix.

Технология Java. Практически одновременно со средой программирования Delphi на свет появилась технология Java, включавшая три составляющих: одноименный язык программирования, очень похожий на язык C++, но более простой и безопасный; универсальный байт-код, в который компилировались программы на языке Java; интерпретатор (виртуальную машину) для выполнения байт-кода в любой операционной системе. Благодаря автоматическому управлению памятью — так называемой «сборке мусора» — резко повысилась надежность программ и скорость их разработки.

Поначалу на технологию Java возлагались большие надежды. Программные библиотеки для языка Java стали единым стандартом, поэтому написанные на нем программы оказались по-настоящему переносимыми. Однажды написанная и компилированная в байт-код программа могла работать на любой платформе без ограничений (единственное требование — наличие на этой платформе виртуальной машины Java).

Безграничная переносимость Java-программ родила идею сетевого компьютера и сетевых вычислений, суть которой в том, что все программы хранят-

ся в байт-коде на серверах сети Интернет. Когда подключенный к сети пользователь запускает программу, то она сначала загружается к нему на компьютер, а затем интерпретируется. Охваченные этой идеей крупные фирмы ринулись осваивать новый рынок Java-приложений. Для языка Java появились средства визуального программирования, такие как JBuilder и Visual Age for Java. Казалось бы, бери и используй их для разработки пользовательского интерфейса и серверных программ. Но практически пропускная способность сети Интернет в лучшем случае обеспечивала оперативную загрузку на клиентские компьютеры лишь небольших по размеру программ. Кроме того, созданный на языке Java пользовательский интерфейс хронически отставал от возможностей операционной системы Windows и раздражал своей медлительностью. Поэтому технологию Java стали применять, главным образом, для разработки серверных приложений. Однако и здесь цена переносимости программ оказалась очень высокой — представленные в байт-коде программы работали на порядок медленнее аналогичных программ, компилированных напрямую в команды процессора. Применение динамической компиляции, при которой программа перед выполнением преобразуется из байт-кода в команды процессора и попутно оптимизируется, улучшило положение дел, но скорость работы Java-приложений так и не смогла приблизиться к скорости работы традиционных приложений. Иными словами, переносимость программ шла в ущерб их производительности и удобству. Многие начали задумываться над целесообразностью такой переносимости программ вообще.

Тем временем назревала революция в области серверных платформ — небывалыми темпами росла популярность операционной системы Linux.

Среда программирования Kylix. В связи со стремительным распространением операционной системы Linux возникла необходимость в эффективных средствах создания программ для нее. Таким средством стала среда Kylix (произносится «киликс») — первая среда визуального программирования для операционной системы Linux. Среда Kylix явилась полным аналогом среды Delphi и была совместима с ней по языку программирования и библиотекам компонентов. Программу, созданную в среде Delphi, можно было без изменений компилировать в среде Kylix, и наоборот. Эта возможность достигалась за счет новой библиотеки компонентов, которая взаимодействовала с операционной системой не напрямую, а через промежуточный программный слой, скрывающий разницу в работе компонентов в той или иной операционной системе. Программисты получили возможность создавать программы сразу для двух самых популярных операционных систем: Windows и Linux. Фактически вместо принципа абсолютной переносимости программ была предложена идея разумной переносимости.

Постепенно пришло понимание того, что в эпоху Интернета способность программ к взаимодействию в сети не менее (а порой более!) важна, чем возможность их переноса на различные платформы. Такая способность была обеспечена за счет стандартизации протоколов обмена данными в сети Интернет и форматов этих данных. Развитие протоколов и стандартов Интернет привело к рождению технологии Web-сервисов, которая ставила своей задачей макси-

мально упростить создание программ, взаимодействующих по принципу клиент-сервер в глобальной сети. Поддержка технологии Web-сервисов была изящно встроена в системы Delphi и Kylix, в результате разработчики программ получили в руки еще один очень важный инструмент.

Технология .NET. Несмотря на трудности и уроки Java-технологии, программисты не желали отказываться от идеи создания полностью переносимых программ. Вместе с тем, их совершенно не устраивала необходимость платить производительностью и удобством программ за переносимость. Работы по разрешению этого противоречия привели к появлению на свет технологии под названием .NET (произносится «дот-нет»).

Технология .NET, по сути, явилась новой платформой, надстроенной над другими операционными системами, и этим походила на технологию Java. Однако у технологии .NET имелся ряд существенных концептуальных отличий. В частности, платформа .NET хотя и имела свой собственный новый язык программирования C# (произносится «си-шарп»), но не была привязана только к нему, позволяя писать программы на других языках. Кроме того, программы для платформы .NET компилировались не в байт-код, а в универсальный промежуточный язык, который сохранял семантику программы и был близок к ее исходному тексту (байт-код, напротив, близок к командам процессора). Программы на промежуточном языке вообще не интерпретировались, а всегда компилировались в команды процессора при запуске программы или при ее первоначальной установке на компьютер пользователя. Выполняемый код получался очень эффективным и оказывался сравнимым по быстродействию с выполняемым кодом, полученным прямой компиляцией с языка высокого уровня в команды процессора. Немаловажно и то, что на платформе .NET стало возможным использование любых (а не только стандартных) библиотек подпрограмм и компонентов, а также всех функций операционной системы. Все это обеспечило создание быстрых и удобных программ.

Поначалу технология .NET была доступна только для семейства операционных систем Windows, но со временем этот недостаток был устранен, и на свет появилась платформа Mono — клон технологии .NET для операционных систем Linux и Unix.

...и опять среда Delphi. Платформы .NET и Mono имеют большое будущее, поэтому фирма Borland адаптировала для них язык и среду программирования Delphi. В итоге разработчики получили уникальную возможность — применять один и тот же язык Delphi для создания профессиональных программ для любых операционных систем и платформ: Windows, Linux, .NET, Mono. Этим, кстати, язык Delphi выгодно отличается от модного ныне языка C#, который применяется лишь для программирования на платформах .NET и Mono.

У языка Delphi есть еще одно очень важное преимущество перед остальными коммерчески успешными языками — он великолепно подходит для обучения программированию. Поэтому авторы рекомендуют его в качестве первого языка для всех учеников и студентов, собирающихся стать профессиональными программистами.

П1.2. Историческая справка 2[50]

Из «древней истории». Пожалуй, первым компьютерным решением, воплотившим в себе объектный подход, стал программно-аппаратный графический планшет (Sketchpad: A Man-Machine Graphical Communications System), использовавший оборудование PDP фирмы DEC. Его в 1963 году разработал 25-летний Иван Сазерленд, помогавший в создании симуляторов вертолетов военному научному агентству DARPA, а затем занявшийся в Массачусетском технологическом институте докторской диссертацией и имевший удовольствие общения с Клодом Шенноном. С помощью светового пера и системы выпадающих меню пользователь планшета мог рисовать различные несложные изображения на аналоговом дисплее, перемещать их и точно располагать в определенных позициях экрана, а также хранить. В качестве понятия класса Сазерленд использовал определение «мастер», разделяя описание и реально существующий на экране объект (экземпляр «мастера»).

Сазерленд также придумал первый прототип нынешних систем виртуальной реальности, ухитрившись спроецировать изображение с двух мониторов непосредственно в глаза человека (как стереоскопический фильм), а было это в 1968 году. Специальные датчики контролировали положение головы и, соответственно, изменяли прокручиваемые перед глазами ролики. Этот человек защитил восемь патентов по компьютерной графике, занимался шагающими роботами в Калифорнийском университете, а затем работал вице-президентом компании Sun Microsystems.

Основоположниками объектного подхода в программировании считаются норвежцы Оле Джохан Дал и Кристен Нюгорт, авторы языка Симула. В 1952 году Нюгорт решил заняться моделированием крупных и сложных систем реального мира, в 1957 получил возможность компьютерной реализации своих замыслов на машине Ferranti MERCURY, но быстро осознал ограниченность машинных языков и Фортрана.

Ferranti MERCURY — версия, созданная Манчестерским университетом в содружестве с компанией Ferranti в ходе работ над компьютером Mark 2. Она отличалась поддержкой операций с плавающей точкой. Память компьютера составляла 1024 слова по 40 бит с временем доступа 10 мкс к 10-битовому слову. Устройство адресации могло работать с 10-, 20- и 40-битовыми словами. Долговременная память насчитывала 4096 40-битных слов и обрабатывала их со скоростью 17,28 мс. Для выявления сбоев выполнялась проверка четности. Программисты решали задачи с помощью языка Mercury Autocode.

Фирма Ferranti продала 19 таких машин, последнюю — в 1963 году.

История Симулы началась в 1962 году с проекта Simulation Language, предназначенного для программного моделирования метода Монте-Карло. Нюгорт, занимавший в то время должность директора по науке Норвежского компьютерного центра (NCC), приступил к созданию языка дискретного моделирования. Он привлек к сотрудничеству Оле Джохана Дала, коллегу по экспери-

[50] Источник: С. Бобровский. История объектно-ориентированного программирования.

ментальной группе Министерства обороны Норвегии, которую возглавлял Йан Гэрвик, основатель информатики в этой стране. Для работ NCC приобрел датский компьютер GIER за 280 тыс. долл.

Одновременно готовились две версии Симулы. Первая, Симула I, формировалась по контракту с подразделением Univac корпорации Sperry Rand для машины UNIVAC 1107 (обошедшейся NCC в 990 тыс. долл.). Американские заказчики желали видеть этот язык фортраноподобным, от чего, впрочем, авторы категорически отказались; в результате прародителем Симулы стал Алгол 60. Последний был выбран благодаря блочной архитектуре, хорошим средствам сокрытия данных, а также вследствие высокой популярности в европейских научных центрах. А работы над вторым вариантом спонсировал NCC.

В 1965 году авторам пришла в голову идея объединить данные с процедурами, их обрабатывающими. После успешного обсуждения возможностей Симулы I на саммите НАТО в 1966 году решено было продолжить его совершенствование. В язык вошли новые средства моделирования и имитации мультипроцессной работы. Авторы также придумали термины «класс» и «объект». Тогда же возникла и технология наследования — создатели Симулы ввели в язык возможность использования разными классами общих свойств путем указания названия класса в виде префикса. После публичного анонса новая технология вызвала интерес в Дании, Германии и СССР. В СССР в конце 1960-х появилась реализация Симулы для УРАЛа-16.

Новая версия языка была закончена в январе 1967 года. Он поддерживал проектирование «сверху вниз» с помощью виртуальных процедур и технологии статического и динамического связывания. В NCC была сформирована рабочая группа Simula Standards Group. Вскоре к ней подключился Якоб Палме, специалист Шведского института оборонных исследований. Он добавил в Симулу механизм сокрытия переменных. Первый законченный компилятор обновленной Симулы 67 увидел свет в 1969 году и работал на машинах Control Data. Затем последовали реализации для UNIVAC 1108 и IBM, но все они, к сожалению, стоили очень дорого, что, вероятно, помешало активному распространению этого прекрасного языка. Однако с его помощью были спроектированы, например, первые СБИС-чипы Intel.

Тем временем Алан Кей (человек, придумавший первый персональный компьютер; см. PC Week/RE, N 39/98, с. 25[51]) внимательно изучал идеи, заложенные в Планшет, Симулу и еще два оригинальных языка — LISP, применявшийся для задач искусственного интеллекта, и LOGO, предназначенный для обучения базовым понятиям программирования. В ходе ознакомления с этими языками Кей придумал новую концепцию разработки, в соответствии с которой набор последовательно выполняющихся инструкций мог быть заменен на многомерную среду взаимодействия объектов, общающихся друг с другом путем асинхронного обмена сообщениями. В результате появлялась возможность поддержки подобной среды не одним, а множеством компьютеров, объединен-

[51] Данное утверждение часто подвергается сомнению.

ных в сеть. Правда, для своего времени эта идея оказалась слишком революционной.

В конце 50-х годов Кей работал на Денверской военно-воздушной базе «Рандольф» (Randolph), где писал на машинном коде программы для ЭВМ Burroughs 220. Тогда он столкнулся с проблемой передачи сформированных на этой машине данных на компьютеры других баз. Стандартных форматов и ОС в современном понимании для тех ЭВМ не существовало, поэтому Кею пришлось создавать микропрограммы, которые содержали в себе всю необходимую информацию и после запуска на других машинах через простой пользовательский интерфейс самостоятельно разворачивали нужные данные. Такие программы Кей первоначально назвал «модулями», объединявшими данные и код. В 1966 году он занялся научной деятельностью в области молекулярной биологии в Университете штата Колорадо, где к нему пришло понимание важности этой идеи. Кей задумал создание системы модулей (возможно, уже тогда он дал им название «объекты»), объединяющих данные и алгоритмы их обработки, способные взаимодействовать друг с другом через определенные разработчиком интерфейсы, но не имеющие представления о том, что происходит за их пределами. При этом он активно использовал аналогии с биологическими объектами и механизмами взаимодействия клеток в живом организме.

Позже Кей перешел в Стэнфордскую лабораторию по искусственному интеллекту, а в 1972 году устроился на работу в хорошо известный научный центр Xerox PARC, где и воплотил эти идеи в новом объектном языке SmallTalk, первоначально названном им Biological System и смоделированном на Бейсике, а затем реализованном на ассемблере. В процессе этой деятельности он предложил знаменитый термин «объектно-ориентированное программирование» (ООП). Заложенные в SmallTalk идеи ООП и по сей день остались непревзойденными ни в каких других языках и системах.

Если Кея нередко называют отцом SmallTalk, то матерью этого уникального языка можно считать профессора лингвистики Адель Голдберг, работавшую в те годы в тесной связке с Аланом в Xerox PARC. Она написала первую документацию к SmallTalk, а затем несколько книг и большое число статей по методологиям объектного анализа.

Мировую известность получила версия SmallTalk 80, коммерческие реализации которой вышли в 1981 году. Правда, они отличались невысокой производительностью. Впоследствии Кей принимал участие в работе группы Squeak, создавшей реализацию SmallTalk под девизом «Объектно-ориентированному программированию нужна скорость Си». Однако превзойти популярность Си проекту Squeak не удалось. Этот элегантный и шустрый язык, придуманный в лаборатории AT&T Bell при участии программистов Кена Томпсона и Денниса Ритчи (известных ЕС-программистам по легендарной игре Star Trek), добрался до ООП своим путем.

«Средние века» ООП. Хронология. 1974 году Марвин Мински, основоположник теории искусственного интеллекта, предложил идею фрейма, отделившего описание класса (структуры) объекта от его конкретного представления (экземпляра) и быстро завоевавшего популярность в языках искусственного

интеллекта. Фрейм стал прямым предшественником современного понятия объектов в Си++.

1976 год. Кристен Нюгорн создал новый язык BETA, в котором ввел концепцию шаблонов — более высокого уровня абстракций, нежели объекты. Его коллега Оле Джохан Дал еще долгое время работал профессором информатики в Университете Осло. А Нюгорн на некоторое время занялся политикой и активно выступал против вступления Норвегии в Евросоюз.

1980 год. Бьерн Страуструп, продолживший дело своих коллег из лаборатории Bell, дополнил язык Си концепцией классов, основанной на фреймах и объектных механизмах Симулы.

1982 год. В Мехико прошла 8-я Международная конференция по сверхбольшим базам данных (VLDB), где была предложена концепция объектно-ориентированных баз данных и рассматривались вопросы расширения существовавших языков запросов к базам данных новыми, объектными типами данных.

1983 год. Страуструп дал своему творению окончательное название Си++.

Наше время. 1986 году Первая всемирная конференция по объектно-ориентированным системам программирования прошла в Портленде. Возможно, именно прозвучавшие на ней доклады оказали стимулирующее влияние на Уильяма Аткинсона, инженера Apple, который через год после этого спроектировал систему HyperCard, прообраз современных визуальных сред быстрой разработки. Эффективность новой технологии оказалась столь высокой, что уже в 1989 году одиннадцать компаний, среди которых были 3Com, American Airlines, Canon, Data General, Hewlett-Packard, Philips Telecommunications, Sun Microsystems и Unisys, основали группу OMG (Object Management Group), призванную формировать индустриальные стандарты на объектное программирование и упрощать интеграцию приложений с помощью универсальных кроссплатформных технологий. Эта группа первым делом приступила к выработке единого стандарта компонентной модели CORBA (Common Object Request Broker Architecture) — набора спецификаций, определяющих способы объектно-ориентированного взаимодействия компонентов промежуточного уровня в гетерогенных средах без привязки к конкретным языкам программирования. С самого начала CORBA нацеливалась на поддержку крупных индустриальных проектов, и этот подход со временем себя полностью оправдал. Сегодня нет другого столь же распространенного независимого стандарта, поддерживающего самые разные операционные системы и объектные модели.

1992 год. Вышел стандарт CORBA 1.0, определяющий ключевые аспекты функционирования CORBA-систем. В него были включены базовое описание объектной модели, наборы программных интерфейсов поддержки CORBA-систем, а также декларативный язык определения интерфейсов Interface Definition Language (IDL), созданный OMG для описания распределенных интерфейсов.

1993 год. Корпорация Microsoft выпустила первую версию компонентной модели COM (Component Object Model), идея которой была, видимо, позаимствована из модели SOM (System Object Model), уже давно существовавшей в IBM OS/2 (в создании OS/2 Microsoft одно время принимала активное участие).

Первоначально COM готовилась только для поддержки технологии встраивания и связывания документов OLE, но быстро выделилась в самостоятельное направление.

В том же 1993 году была предложена компонентная технология Microsoft ActiveX, основанная на элементах управления OLE, пришедших из Visual Basic, где они назывались VBX/OCX.

1994 год. Опубликован стандарт CORBA 2.0, который быстро получил массовое признание, так как представлял собой богатый и глубоко проработанный набор документов и охватывал большинство востребованных рынком задач. В нем были ликвидированы недостатки прежней версии, в результате чего CORBA 2.0 начал поддерживать транзакции и понимать универсальную кодировку Unicode, а также появился набор средств обеспечения безопасности и взаимодействия COM- и CORBA-объектов.

Гради Буч и Джеймс Румбах из Rational Software решили объединить две методологии визуального моделирования Booch и OMT и создать на их основе новый язык UML (Unified Modeling Language).

1995 год. Sun Microsystems свободно распространяет в Интернете элемент технологии Java-среду HotJava, поддерживающую мобильный код, разработку проекта Green, которая к тому времени считалась в Sun практически пропащей, если бы не развитие Сети. Новинку сразу же лицензирует Netscape Communication, а следом за ней к Java проявляют коммерческий интерес десятки компаний, в том числе Microsoft, IBM, Adobe, Borland, Lotus, Oracle.

Корпорация Borland выпустила первую версию среды быстрой визуальной разработки Delphi 1, основанную на концепции библиотек стандартных компонентов.

Microsoft сообщает о новой технологии DCOM — распределенной версии COM, позволившей собирать приложения из компонентов, выполнявшихся на разных компьютерах. Первоначально эта технология называлась Network OLE (сетевое OLE), однако по мере выделения COM в самостоятельное направление решено было отказаться от упоминания OLE в ее названии.

1996 год. Microsoft называет ActiveX новой объектной стратегией, направленной на поддержку Интернета.

1997 год. Sun Microsystems предлагает концепцию Enterprise JavaBeans — технологию создания корпоративных Java-компонентов, которые можно исполнять на серверах приложений, реализуя логику крупных, хорошо масштабируемых и защищенных систем на платформно-независимой основе.

Эксперты OMG осознают не только важность объектных технологий программирования, но и острую потребность в универсальных методологических концепциях проектирования крупных систем. Секретом стабильности системы и высокой отдачи инвестиций специалисты OMG называют независимую UML-модель и приступают к созданию концепции «Архитектура, управляемая моделью» (Model Driven Architecture, MDA). В ее основу закладывается базовая платформно-независимая UML-модель системы, несколько платформно-зависимых моделей и коллекция определений программных интерфейсов. Первую

реализацию этой универсальной концепции (так называемое «отображение в объектный стандарт») OMG выполнила, конечно, для CORBA.

2000 год. Microsoft анонсирует новую объектную платформу .NET и новый язык программирования C#, сочетающий лучшие свойства C++ и Java. Он был предложен Microsoft во многом в противовес Java.

2001 год. OMG выпускает спецификацию CORBA 3.0. Она дополнена возможностями асинхронного обмена сообщениями, разработки систем реального времени и создания встраиваемых систем. В ней появились подключаемые компоненты, поддержка XML и средства интеграции различных интернет-технологий. Была продумана модель сборки системы из компонентов JavaBeans и ActiveX. Стало допустимым в рамках одного компонента описывать множество интерфейсов, а также использовать язык сценариев. Особый акцент в третьей версии CORBA сделан на эффективном взаимодействии с Java.

2002 год. Опубликована последняя официальная версия CORBA 3.0.2.

Что дальше? Несмотря на многомиллионные вложения, сделанные в 1970-1980 годах коммерческими компаниями и государственными структурами разных стран в универсальные языки программирования (такие как Алгол, PL/1, Си) и языки логического программирования (прежде всего Prolog), самой распространенной в мире программной технологией остается ООП. Наиболее известным событием нового тысячелетия в этой сфере стал быстро набравший популярность язык программирования C#. Его можно смело считать лучшим на сегодня объектно-ориентированным средством создания графических приложений (формально — для разных платформ, фактически же — только для .NET, так как без среды разработки ценность C# равна нулю). Правда, лучшим C# можно назвать среди небольшого числа коммерческих альтернатив (Java, C++ и, вероятно, Delphi). Если же сравнить его возможности со средствами SmallTalk, то окажется, что ряд сильных идей ООП, заложенных в SmallTalk продвинутыми инженерными умами, оказался в C# упущенным (см. врезку «Оригинальные возможности Smalltalk»).

В ближайшее десятилетие развитие ООП будет проходить под влиянием трех концепций: Microsoft.NET (прежде всего .NET Framework и ее подмножества и, соответственно, реализации C#), Java (все входящие в это понятие технологии) и CORBA. Причем важнейшей особенностью CORBA останется независимость от операционной системы и языка программирования CORBA-компонентов. Методология MDA станет основной объединяющей платформой моделирования для всех этих технологий, прежде всего в силу своей независимости от конкретного разработчика. Немаловажна и объявленная сквозная поддержка MDA во всех средствах разработки «Швейцарии в мире программного обеспечения» — корпорации Borland — и гиганта IBM (особенно учитывая недавнюю покупку Rational Software и развитие проекта Eclipse). Что касается «программирования в малом» (programming in small), то в этой области продолжится совершенствование шаблонов проектирования (высокоуровневого объектного программирования), их внедрение в среды разработки и дальнейшее сочетание шаблонного подхода с активно развивающимся языком моделирования UML, последняя версия которого 2.0 уже появилась.

Хочется также надеяться, что со временем в коммерческих оболочках создания программного обеспечения на популярных языках программирования все же будут реализованы возможности ООП, предложенные еще 25 лет назад авторами SmallTalk.

Оригинальные возможности Smalltalk

Объекты могут обмениваться сообщениями (одна из ключевых идей ООП).

SmallTalk — это не только язык программирования, но и комплексная стандартизованная ANSI-среда разработки. Она легко переносима между платформами, так как написана на самой себе и при этом очень компактна (например, Squeak «весит» 7 Мб). В нее входят средства создания многооконных приложений, встроенные службы конфигурационного управления и контроля версий, а кроме того, она поддерживает отладку на всех этапах создания и развертывания продукта.

SmallTalk создавался как среда прототипирования (важный принцип ООП, подразумевающий спиральную разработку приложения, быстрое получение и совершенствование действующего прототипа).

Подготовленное с помощью SmallTalk приложение представляет собой один файл, хранящий все состояние системы.

Можно динамически создавать отложенные последовательности действий (блоки), что под силу лишь интерпретируемым системам.

Хотя из SmallTalk, как из Java и C#, удалена возможность множественного наследования, только эта среда позволяет элегантно имитировать его с помощью протоколов обмена сообщениями между классами.

Объект умеет информировать внешнюю среду о попытках неправильного использования самого себя (метод doesNotUnderstand).

В SmallTalk нет разделения между типами и классами, благодаря чему можно избегать ошибок приведения типов. Концепция набора базовых типов, позволяющая выявлять такие ошибки на этапе компиляции, далеко не всегда себя оправдывает, поскольку уводит разработчика от мышления в истинно объектном духе.

Имеются механизмы динамического расширения классов и приведения типов, перезагрузки исключений, расширения возможностей операторов, организации замкнутых выражений и др.

Документацию и комментарии можно встраивать в код.

ЛАНЬ®

ПРИЛОЖЕНИЕ 2. ПРАВИЛА ИМЕНОВАНИЯ ИДЕНТИФИКАТОРОВ

Любая среда, будь то среда программирования или игра в бильярд, имеет свой набор терминов и общепринятый жаргон. Эти термины помогают улучшить взаимодействие между различными участниками среды.

Имена играют важную роль, так как они встречаются в программировании повсеместно. Мы присваиваем имена своим переменным, функциям, аргументам, классам. Мы присваиваем имена исходным файлам и каталогам, в которых они хранятся. Напрашивается вывод: то, что делается так часто, должно делаться хорошо.

Правильно выбранные имена могут сделать программу поистине самодокументированной, не требуя совсем или требуя мало дополнительного набора в виде явных комментариев. Плохо выбранные имена могут внести программисту дополнительные трудности в работу с программой.

C++Builder предоставляет множество различных типов данных, кроме того, имеется возможность создавать новые типы данных. Из-за большого количества типов данных, переменных, констант, свойств и небрежностей их именования чтение программы может вызывать определенные трудности. Например, попробуем понять следующую строку, содержащую операцию присваивания:

T=T0;

Пока можно утверждать, что переменной **T** присваивается какое-то начальное значение. Добавим к этому, что оператор присваивания записал начинающий программист. Какие типы имеют переменные **T** и **T0**? Чтобы ответить на этот вопрос, мы вынуждены искать в программе то место, где они объявлены, что неэффективно и мало способствует пониманию программы. Рассмотрим теперь присваивание:

realT=intT0;.

Здесь **real** и **int** используются в качестве префиксов к названиям переменных и характеризуют тип переменной. Мы сразу видим, что вещественной переменной присваивается целое значение. Появляется сомнение в правильности записанного оператора. Усовершенствуем наш пример:

realTime=intStartTime;.

Теперь достаточно одного взгляда, чтобы понять, что речь идет о моделировании времени и время задается вещественным числом. Кроме того, мы видим, что в рассмотренном операторе устанавливается начальное значение времени, причем это начальное значение времени задается на границе принятой в программе единицы измерения времени (часа или минуты, или секунды). В нашем распоряжении теперь имеется немало прямых и косвенных данных, необходимых для понимания программы. Мы сразу чувствуем руку опытного программиста.

Использование осмысленных имен не составляет никакого труда для разработчика, но упрощает жизнь всем тем, кому впоследствии придется читать код (в первую очередь это касается самого разработчика).

П2.1. Общие рекомендации по именованию идентификаторов

Одним из способов повышения читабельности кода является использование осмысленных имен для переменных, типов и функций. Рассмотрим некоторые правила и рекомендации, предлагаемые в литературе.

Правило 1. Имена должны быть обычными словами английского языка, описывающими то, что делает функция, или обозначающими назначение аргумента или переменной.

Избегайте аббревиатур; они ухудшают читабельность программ. Избегайте отбрасывания последних букв слова или удаления из него всех гласных. Например: странно названная функция **creat()**; очевидно, что **create()** лучше. Например: странности типа **lngth** вместо **length** и **mt** вместо **empty**.

Однако общепринятые аббревиатуры и сокращения являются очевидными исключениями. Вот некоторый минимум из них, который не вызывает неверных толкований:

— **col** — индекс столбца;

— **cur** — текущий;

— **i**, **j** — обобщенные счетчики циклов;

— **max** — максимум (появляется обычно в качестве префикса или суффикса);

— **min** — минимум (появляется обычно в качестве префикса или суффикса);

— **obj** — обобщенный объект (имеется указатель на базовый класс, но производный класс не известен);

— **p**, **ptr** — обобщенный указатель (появляется обычно в качестве префикса);

— **s**, **str** — строка (появляется обычно в качестве префикса), но не употребляйте их, если называемый объект используется не как обобщенный. Например, **i** имеет смысл использовать в качестве счетчика цикла в операторе **for**, если этот счетчик используется просто для подсчета числа итераций:

```
for( i=10; --i >= 0; ) // вывести десять тире
putchar('-');
```

Не используйте имена ***i***, ***j*** и т. п., если вы применяете счетчик для чего-нибудь отличного от счета. Сравните этот код:

```
for ( i=0; i < imax; ++i )
   for ( j=0; j < jmax; ++j )
      move_cursor( i,j );
```

со следующим:

```
for ( row=0; row < max_row; ++row )
   for ( col=0; col < max_col; ++col )
      move_cursor( row, col );
```

Дополнительное замечание к последнему примеру: избегайте использовать ***x*** и ***y*** вместо **row** и **col**.

Правило 2. Не используйте в качестве имен тарабарщину.

Ввиду очевидности правила примеры не приводим.

Правило 3. Имена макросов должны записываться заглавными буквами.

Макросы часто вызывают побочные эффекты. Поэтому полезно иметь возможность определить с первого взгляда, что у нас является макросом. Конечно, мы не должны использовать только заглавные буквы для чего-нибудь помимо макросов, иначе цель данного правила не будет достигнута.

Правило 4. Не пользуйтесь именами из стандарта ANSI Си.

Идентификаторы, начинающиеся с символа подчеркивания, и имена типов, оканчивающиеся на **_t**, были зарезервированы стандартом ANSI Си для использования разработчиками компиляторов. Не используйте эти приемы. Также избегайте имен функций, вошедших в стандарт ANSI Си и в проект стандарта ISO/ANSI для Си++. Популярно стандарты изложены в книге: Страуструп Б. Язык программирования С++(пер. с англ. 3-е изд. СПб.; М.: Невский диалект; БИНОМ, 1999. 991 с.).

Правило 5. Осторожно пользуйтесь именами Microsoft.

Это может показаться правилом, специфичным только для Microsoft, но на самом деле это не так (учитывая имеющуюся склонность Microsoft к мировому господству). Любой, кто заботится о переносимости, должен рассчитывать на то, что его или ее программа со временем может или работать под управлением операционной системы Microsoft, или взаимодействовать с библиотекой классов Microsoft. Библиотека MFC, например, перенесена на Macintosh и во многие операционные среды UNIX, и, вероятно, появится на других операционных системах в ближайшем будущем. Интерфейс прикладного программирования Windows (API) включает в себя около 1200 функций. Библиотека MFC, быстро вытесняющая первоначальный интерфейс на языке Си, добавляет около 80 определений классов. К сожалению, метод Microsoft состоит в добавлении каждый раз дополнительных функций и классов в новую версию компилятора. Если в дальнейшем Microsoft случайно выберет в качестве имени для функции или класса то, которое вы использовали для каких-то других целей, угадайте, кому придется его сменить?

Так как ни один из идентификаторов Microsoft не соответствует стандарту ANSI Си, требующему, чтобы имена поставляемых разработчиком объектов начинались с символа подчеркивания, то мы должны предохраняться, избегая использования соглашений по выбору имен в стиле Microsoft:

— все имена функций Microsoft используют соглашения в стиле Паскаля о СмесиЗаглавныхИСтрочныхБукв(), и они всегда начинаются с заглавной буквы. Методы в классах MFC используют то же самое соглашение;

— все имена классов Microsoft начинаются с заглавной «С» с последующей заглавной буквой (например, CString, CWnd, CDialog и т. д.);

— одна из наиболее фундаментальных заповедей объектно-ориентированного проектирования запрещает оставлять незащищенными поля в опре-

делении класса. Тем не менее, многие классы MFC имеют открытые поля данных. Все эти поля начинаются с **m_**. Не нужно начинать имена своих собственных полей с **m_,** таким образом мы легко отличим свои поля от унаследованных из базовых классов MFC.

Правило 6. Избегайте ненужных идентификаторов.

Имена для констант иногда вообще не нужны. Например, не определяйте значения, возвращаемые при ошибке; если возвращается всего одна ошибка, возвратите просто FALSE. Не делайте так:

```
enum ErrotMode { INSERT_ERROR, DELETE_ERROR };

void insert()
{
   //...
   return INSERT_ERROR;
}

void delete()
{
  //...
  return DELETE_ERROR;
}
```

а просто возвратите 0 в случае ошибки, и в случае успеха любое правильное значение типа 1.

Замечание (Барков И. А.). Это пожелание не кажется очевидным.

Правило 7. Именованные константы для булевых величин редко необходимы.

Выбор неверного имени может добавить значительную ненужную сложность в нашу программу. Рассмотрим следующую простейшую функцию, которая подсчитывает количество слов в строке:

```
int nwords(const char *str)
{
     typedef  enum { IN_WORD, BETWEEN_WORDS } wstate;
     int word_count=0;
     wstate state=BETWEEN_WORDS;
     for(; *str ; ++str )
          {
              if( isspace(*str) )
                 state=BETWEEN_WORDS;
              else
                    if( state != IN_WORD )
                      {
                         ++word_count;
                         state=IN_WORD;
                      }
        }
```

```
    return word_count;
}
```

Неправильно выбранное имя **state** заставило нас ввести два ненужных идентификатора: **IN_WORD** и **BETWEEN_WORDS**. Теперь взгляните на этот вариант:

```
int nwords2(const char *str)
{
    int word_count=0;
    int in_word=0;
    for(; *str ; ++str )
       {
         if( isspace(*str) )
           in_word=0;
         else
             if( !in_word )
               {
                  ++word_count;
                  in_word=1;
               }
      }
   return word_count;
}
```

Переименование нечетко названной переменной **state** во что-нибудь, что действительно описывает назначение переменной, позволяет исключить булевые именованные константы **IN_WORD** и **BETWEEN_WORDS**. Получившаяся подпрограмма меньше и легче читается. Вот другой пример. Следующая программа:

```
enum child_type { I_AM_A_LEFT_CHILD, I_AM_A_RIGHT_CHILD };
struct tnode
{
     child_type position;
     struct tnode *left,*right;
} t;
//...
t.position=I_AM_LEFT_CHILD;
if( t.position == I_AM_LEFT_CHILD )
//...
```

может быть упрощена подобным образом:

```
struct tnode
{
    unsigned is_left_child ;
    struct tnode *left,*right;
} t;
t.is_left_child=1;
if( t.is_left_child )
//...
```

тем самым исключая два ненужных идентификатора. И вот последний пример:

```
enum { SOME_BEHAVIOR, SOME_OTHER_BEHAVIOR,
SOME_THIRD_BEHAVIOR };
f( SOME_BEHAVIOR, x);
f( SOME_OTHER_BEHAVIOR, x);
f( SOME_THIRD_BEHAVIOR, x);,
```

требующий четырех идентификаторов (три именованные константы и имя функции). Лучше, хотя это не всегда возможно, исключить селекторную константу в пользу дополнительных функций:

```
some_behavior(x);
some_other_behavior(x);
some_third_behavior(x);.
```

Обратной стороной этой монеты является вызов функции. Рассмотрим следующий прототип:

```
create_window( int has_border, int is_scrollable,int is_maximized );.
```

Здесь снова выбраны рациональные имена для исключения необходимости в именованных константах. К сожалению, вызов этой функции плохо читаем:

```
create_window(TRUE, FALSE, TRUE );.
```

Просто взглянув на такой вызов, мы не получим никакого представления о том, как будет выглядеть это окно. Несколько именованных констант проясняют обстоятельства в этом вызове:

```
enum {UNBORDERED =0; BORDERED =1};//Нужно показать значения
enum {UNSCROLLABLE=0; SCROLLABLE =1}; // или create_window()
enum {NORMAL_SIZE =0; MAXIMIZED =1}; // не будет работать
//...
create_window( BORDERED, UNSCROLLABLE, MAXIMIZED );.
```

Но теперь у нас другая проблема. Мы не хотим использовать именованные константы внутри самой функции **create_window()**. Они здесь только для того, чтобы сделать ее вызов более читаемым, и мы не хотим загромождать эту функцию таким кодом, как:

```
if( has_border == BORDERED )
    //...,
```

сравнивая его с более простым:

```
if( has_border )
    //....
```

Первый вариант уродлив и многословен. К сожалению, если кто-то изменит значение именованной константы **BORDERED**, второй оператор **if** не будет работать. Поэтому можно согласиться с мнением, что программист, занимающийся сопровождением, не должен менять значения идентификаторов, как мы это проделали в предыдущем примере.

П2.2. Некоторые договоренности об именовании идентификаторов

П2.2.1. Соглашения об именах MFS

В качестве префикса, обозначающего имя класса, библиотека MFS использует заглавную букву «C» от слова «class» (класс), за которой следует имя, характеризующее его назначение. Например, **CWinApp** — класс, определяющий приложение, **CWnd** — базовый класс всех оконных объектов, **CWinDialog** — класс блоков диалога и т. д.

Для именования методов класса используются три способа. При одном имя метода объединяет глагол и существительное, например, **LoadIcon** (Загрузить пиктограмму) или **DrawText** (Нарисовать текст). При втором имя метода состоит только из существительного, например, **DialogBox** (Блок диалога). Для функций, преобразующих один тип в другой, обычными являются такие имена, как **XtoY** (из X в Y).

Для полей классов библиотеки MFS принят следующий способ назначения имен: обязательный префикс m_ (от class member — член класса), за которым идет префикс, характеризующий тип данных, и завершается все содержательным именем переменной, например, **m_pMainWnd**, где p — префикс, описывающий указатель (для переменных, не членов класса, префикс m_ не используется, чтобы можно было отличить обычные переменные от полей любых классов, а не только классов библиотеки MFS).

П2.2.2. Венгерская нотация

Венгерским программистом фирмы Microsoft Чарльзом Симони (Charles Simonyi) были предложены соглашения о правилах именования идентификаторов, получивших впоследствии название *венгерской нотации*. Суть этой нотации заключается в том, что имя переменной начинается с префикса, который содержит одну или несколько строчных букв. Этот префикс определяет тип переменной. За префиксом следует само имя, которое может состоять как из прописных, так из строчных букв, но первая буква имени всегда прописная. Например, идентификатор **nInformationContent** обозначает некоторую целую переменную (назначение переменной предлагаем определить самостоятельно). Основные префиксы венгерской нотации приведены в таблице П1.

Таблица П1

Префиксы имен переменных для указания типа данных

Префикс	Соответствующий тип данных
b	BOOL
by	BYTE
cc	CHAR
dw	DWORD
fn	Функция
h	Дескриптор
i	INT
l	LONG

Продолжение табл. П1

Префикс	Соответствующий тип данных
lp	Дальний указатель (Барков И. А.: понятие можно считать устаревшим)
n	short или int
p	Указатель
pfn	Указатель на функцию
psz	Указатель на строку, заканчивающуюся нулевым байтом
pv	Указатель на тип void
sz	Строка, заканчивающаяся нулевым байтом
U	UINT
v	void
w	WORD
x	Короткое целое число, используемое в качестве координаты x
y	Короткое целое число, используемое в качестве координаты y

Теперь можно легко сообразить, что обозначают такие имена переменных, как **lpText** или **uType**.

П2.3. Наши соглашения об именовании идентификаторов

Можно придерживаться любой системы обозначений или придумать что-то свое. Однако солидные программистские фирмы обычно разрабатывают свои стандарты. Мы также примем за основу приведенные далее соглашения.

П2.3.1. Именование переменных, констант и свойств

Введем правило именования идентификаторов переменных, констант и свойств. В общем случае идентификатор имеет следующую структуру:

<префикс> <источник> <имя>,

где **<префикс>** представляет собой короткое слово, содержащее сведения о типе идентификатора; **<источник>** — символ, характеризующий объявление идентификатора в программе: константа, переменная, поле класса и т. п.; **<имя>** — название идентификатора, несущее в себе смысловое содержание идентификатора.

Префикс идентификатора. Название идентификатора начинается с префикса, в котором содержится краткая информация о типе данных. Рекомендуемые значения префиксов для различных типов данных приведены в таблице П2 и П3.

Таблица П2

Префиксы идентификаторов

Тип данных C++	Тип данных Object Pascal	Префикс	Характеристика значений
char, signed char	ShortInt	chr	Целое 8 бит
	Char, AnsiChar	chr	Символ 8 бит
unsigned char	Byte, ByteBool	by	true/false или целое без знака 8 бит
wchar_t	WideChar	wchr	Символ Unicode размером в слово

Продолжение табл. П2

Тип данных C++	Тип данных Object Pascal	Префикс	Характеристика значений
short	SmallInt	sint	Целое 16 бит
unsigned short	Word, WordBool	wrd	true/false или целое без знака 16 бит
enum	Перечислимый тип данных	enum	0, 1, 2, …
long, int	LongInt, Integer	int	Целое 32 бит
unsigned long, unsigned int	Cardinal	card	Целое без знака 32 бит
SmallString<n>	String[n]	str	n=1, 2, …, 255
SmallString<255>	ShortString	str	
AnsiString	String	str	
float	Real		Устаревшие типы данных
	Single	sgl	Вещественное 4 байта
double, Double,	duble	dub	
long double, Extended	Extended	ext	
bool, Boolean, LongBool	bool	bool	
void *, Pointer	ptr	ptr	
unsigned char *, PChar, PAnsiChar	pchr	pchr	

Замечание. В таблице П2 нетрадиционные для языка C++ идентификаторы типов данных взяты из языка Object Pascal. Для использования их программах на языке C++ возможно придется добавить директиву **#include <sysdefs.h>.**

Таблица П3

Префиксы идентификаторов

Класс	Префикс	Класс	Префикс
TObject	obj	TRadioButton	rbtn
TForm	frm	TMainMenu	mm
TTable	tbl	TMenuItem	mi
TQuery	qry	TOpenDialog	od
TStoredProc	sp	TSaveDialog	sd
TDataSource	ds	TListBox	lbox
TDBGrid	dbg	TList	lst
TButton	btn	TStringGrid	sgr
TPanel	pnl	TEdit	edt
TDirectoryListBox	dlb	TActionList	alst
TGroupBox	gbox	TImageList	ilst
TBitBtn	bbtn	TStringList	slst

Признак происхождения описания идентификатора.

Признаки описания идентификатора приведены в таблице П4.

Таблица П4

Признаки описания идентификатора

Вид идентификатора	Признак
Константа	C
Переменная	V
Поле класса или записи	F
Формальный параметр	A
Свойство	P

Название идентификатора формулируется с указанием его семантики.

Допускается именование идентификаторов с отступлением от указанных правил для обозначения локальных рабочих переменных и констант, а также формальных параметров в небольших по объему функциях и процедурах.

При обработке массивов с помощью операторов цикла использовать для названий индексов буквы *i*, *j*, *k*, *l*, *m*.

Примеры идентификаторов:

— **strFDivisionName** — переменная типа String, является полем класса, название переменной — DivisionName (Имя раздела);

— **byCQuantityTables** — константа типа byte, название QuantityTables (Количество таблиц);

— **crdarVConcepts** — массив с элементами типа Cardinal, объявлен как переменная, название Concepts (Понятия);

— **pnlFOpenDirectory** — поле класса **TPanel**, объявлено, вероятней всего, в потомках класса **TForm;**

— **lblFDirectoryPath** — поле класса TLabel.

Приведенные примеры показывают, как использовать заглавные буквы в названиях идентификаторов, при этом от использования символа _ (подчеркивание) можно отказаться.

ЛАНЬ

П2.3.2. Именование типов данных

Описание типа данных класса формируется по принятым в RAD Studio соглашениям: должно начинаться с префикса T, за которым следует осмысленное название типа данных.

Примеры:

- **TfrmProductionSpecification** — класс потомок **TForm**, название ProductionSpecification (Спецификация изделия). Инспектор объектов Delphi не позволяет по разному называть класс и переменную класса;
- **TConditions** — тип данных Conditions (Условия).

П2.3.3. Именование функций

Каждая функция представляет собой набор операторов, предназначенных для достижения конкретной цели. Имя функции обычно должно отражать эту цель. Как правило, имя функции составляется из двух компонентов: глагола, описывающего действие, и существительного, описывающего объект (или субъект) действия (например, **processTransaction()**). Когда объект действия ясен из контекста (например, когда он передается функции как параметр), можно использовать только глагол (**add()**, **delete()** и т. д.).

ЛАНЬ®

ПРИЛОЖЕНИЕ 3. ПРОГРАММА ТАБЛИЧНОГО ХРАНЕНИЯ СЛОВ

П3.1. Общее описание

Программа предназначена для поддержки табличного метода хранения и поиска слов. Пользователь может создать произвольное количество таблиц слов. Используется алгоритм поиска слов в таблице методом хеширования. Каждая таблица при переполнении имеет возможность динамически увеличивать свои размеры. С каждой таблицей пользователь имеет возможность связывать свои данные. Имеется возможность сохранять систему таблиц во внешней памяти и восстанавливать ее в оперативной памяти.

П3.2. Описание алгоритмов

Пусть имеется массив слов. Рассматривается три задачи: поиска слова в таблице, размещения нового слова в таблице, удаление слова из таблицы. Наиболее распространенным алгоритмом решения указанных задач является алгоритм хеширования. Название алгоритма произошло от английского слова «hash», которое имеет несколько значений, но в данном контексте наиболее подходит значение «мешанина». В случае применения алгоритма хеширования таблицы получили краткое название «хеш-таблицы».

Выполнение операции в хеш-таблице начинается с вычисления хеш-функции. Получающееся хеш-значение играет роль индекса в массиве слов.

Ситуация, когда для различных слов получается одно и то же хеш-значение, называется коллизией. Такие события не так уж и редки — например, при вставке в хеш-таблицу размером 365 ячеек всего лишь 23 элементов вероятность коллизии уже превысит 50% (если каждый элемент может равновероятно попасть в любую ячейку). Поэтому механизм разрешения коллизий — важная составляющая любой хеш-таблицы.

Алгоритм вычисления значения хеш-функции слова s в таблице имеет вид:

$$h = \sum_i t_i \bmod(n),$$

где h — значение хеш-функции; i — номер тройки подряд идущих символов слова s; t_i — переведенный в целое число код i-й тройки символов ($i = 1, ..., k$, k — количество троек символов в слове s); n — размер таблицы; mod — операция взятия остатка от деления левой части на правую[52].

Алгоритм разрешения коллизии часто называют рехешированием. Целью алгоритма рехеширования является поиск альтернативного места слова в массиве. Использован алгоритм линейного рехеширования: для вычисленного значения хеш-функции h на каждом i-м шаге вычисляется новый индекс h_i в массиве слов:

$$h_i = (i+1) \cdot h \cdot \bmod(n).$$

[52] В программе использовались более десятка различных хеш-функций.

Для успешного использования алгоритма линейного рехеширования необходимо, чтобы размер массива слов выражался простым числом. Для иллюстрации алгоритма рассмотрим случай h=3, n=7. Тогда h_1=6, h_2=2, h_3=5, h_4=1, h_5=4, h_6=0.

П3.3. Структуры данных

Структуры данных табличного метода хранения слов представлены на рисунке П3.1.

Все поступающие в таблицы слова хранятся в массиве «Куча». Размещение слов в массиве «Куча» происходит по мере поступления. При переполнении массива «Куча» его размер увеличивается.

«Массив хеш-таблиц» предназначен для хранения тематических массивов слов. При поступлении очередного слова в тематический массив вычисляется индекс слова в тематическом массиве. Затем слово размещается в первой свободной ячейке массива «Куча». Значение индекса в массиве «Куча» записывается в вычисленную ячейку тематического массива, а поля «Номер таблицы», «Номер строки» массива «Куча» заполняются координатами слова в тематическом массиве (Номер таблицы, Хеш-индекс слова).

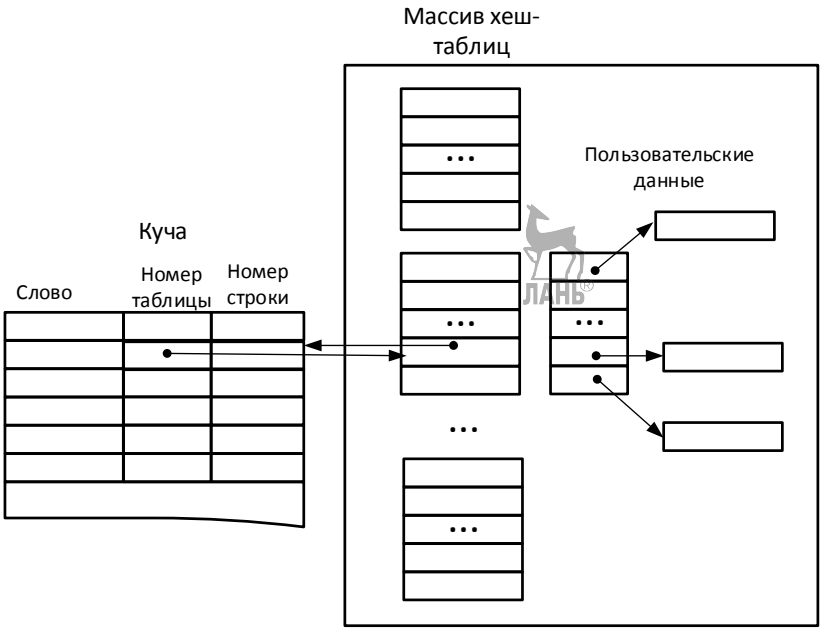


Рис. П3.1
Структуры данных табличного метода хранения слов

Внешние ссылки на хранящиеся в таблицах слова задаются индексом в массиве «Куча». Положение слова в массиве «Куча» получило название «лексический код». Лексический код полностью характеризует хранящееся в таблицах слово. При переполнении тематического массива его размеры увеличиваются, при этом каждое слово тематического массива получает новый индекс.

Соответственно, корректируется в массиве «Куча» ссылка на слово в тематическом массиве. В результате при увеличении тематического массива внешние ссылки на слова в виде лексического кода остаются неизменными.

Каждой хеш-таблице можно сопоставить массив ссылок на пользовательские данные. Любому слову по желанию пользователя можно приписать дополнительные структуры данных произвольного вида.

П3.4. Диаграмма классов

Диаграмма классов программы представлена на рисунке П3.2.

Рассмотрим свойства и методы класса **THashTableList**, который моделирует массив хеш-таблиц (рис. П3.1).

Свойство

-arrFHashTableList[0..*] : THashTable*

представляет собой массив ссылок на хеш-таблицы. Свойство находится в закрытой области видимости, массив может иметь произвольный размер.

Свойство

+bytePTableSize : Byte

выдает информацию о текущем количестве хеш-таблиц.

Свойство

+cardPMaxLexicalCode : Cardinal

информирует об общем количестве слов, хранящихся в системе таблиц.

Векторное свойство

+Tables[0..*] : Byte

сопоставляет заданному лексическим кодом слову номер тематического массива.

Векторное свойство

+LexicalUnits[0..*] : String

сопоставляет лексическому коду хранящееся в системе таблиц слово.

Векторное свойство **+UserTables[0..*] : void*** представляет собой массив ссылок на пользовательские данные.

Конструктор

+THashTableList(Byte byteATableCount)

позволяет задать параметром количество создаваемых хеш-таблиц.

Конструктор

+THashTableList(byteAFileType:Byte, strAFileName:String)

воссоздает предварительно сохраненную в файле **strAFileName** систему хеш-таблиц.

Деструктор

+~THashTableList()

разрушает систему хеш-таблиц. Построенные пользовательские данные необходимо ликвидировать отдельными действиями.

Метод

+SearchLexicalUnit(strALexicalUnit:String, byteATable:Byte, *cardALexicalCode:Cardinal):bool

предназначен для поиска слова **strALexicalUnit** в хеш-таблице с номером **byteATable.** Результат поиска в виде лексического кода записывается в параметр **cardALexicalCode** (при неуспешном поиске записывается 0). Метод возвращает логическое значение, истина соответствует успешному завершению операции.

Метод

+AddLexicalUnit(strALexicalUnit:String, byteATable:Byte, *cardALexicalCode:Cardinal):bool

позволяет добавить слово **strALexicalUnit** в хеш-таблицу с номером **byteATable.** Результат размещения слова в виде лексического кода **cardALexicalCode** записывается в параметр **cardALexicalCode**. Метод возвращает логическое значение «истина» в случае добавления в таблицу нового слова и значение «ложь», если слово **strALexicalUnit** уже находилось в таблице **byteATable**.

Метод

+DeleteLexicalUnit(strALexicalUnit:String, byteATable:Byte)

удаляет слово **strALexicalUnit** из хеш-таблицы **byteATable** в случае, если оно там содержится.

Метод

+DeleteLexicalCode(cardALexicalCode:Cardinal)

удаляет заданное лексическим кодом **cardALexicalCode** слово из системы хеш-таблиц.

Метод

+SetUserTable(const Byte byteATable)

создает пользовательскую таблицу, ассоциированную с хеш-таблицей **byteATable**.

Метод

+GetLexicalUnitsList(byteATable:Byte, *slstALexicalUnits:TStringList)

копирует все слова хеш-таблицы **byteATable** в массив строк **slstALexicalUnits**.

Метод

+Expantion()

позволяет в динамике добавить хеш-таблицу к системе хеш-таблиц. Таблица получает номер, который на единицу больше текущего значения свойства **bytePTableSize**. Естественно, что после добавления таблицы значение свойства **bytePTableSize** увеличивается на единицу.

Метод

+Save(byteAFileType:Byte, strAFileName:String)

позволяет сохранить систему хеш-таблиц во внешней памяти. Параметр **byteAFileType** определяет тип файла. Значениями параметра могут быть: **TEXT_FILE** соответствует текстовому файлу, **STREAM_FILE** соответствует потоку. Второй параметр **strAFileName** задает имя файла. О сохранении в файле пользовательских данных должен заботиться сам пользователь.

Метод

+Load(byteAFileType:Byte, strAFileName:String)

позволяет восстановить в оперативной памяти систему хеш-таблиц, предварительно записанных во внешнюю память методом **Save**. Параметры метода имеют тот же смысл. Пользовательские данные должен восстанавливать сам пользователь.

Переходим к классу **THashTable** — хеш-таблица. Хеш-таблица является элементом массива хеш-таблиц (рис. П3.1).

Закрытое свойств

–arrFHashTable[0..*] : Cardinal является хеш-таблицей.

Свойство

+arrPUserTable[0..*] : void*

представляет собой массив ссылок на пользовательские данные. Массив ссылок создается динамически по требованию пользователя вызовом метода **SetUserTable** (см. далее).

Свойство

+cardPTableSize : Cardinal

возвращает размер хеш-таблицы.

Конструктор

+THashTable(THeap *objAHeap)

возвращает в параметр **objAHeap** ссылку на объект «куча».

Деструктор

+~THashTable()

удаляет из оперативной памяти объект хеш-таблица.

Метод

+SetUserTable()

создает массив ссылок на пользовательские данные.

Метод

+SearchLexicalUnit(String strAlexicalUnit, Cardinal * cardALexicalCode):bool

выполняет поиск слова **strAlexicalUnit** в таблице и лексический код найденного слова записывает в параметр **cardALexicalCode**. В случае успешного выполнения операции функция возвращает значение «истина».

Метод

+AddLexicalUnit(String strAlexicalUnit, Byte byteATable, Cardinal *cardALexicalCode) : bool

добавляет слово **strAlexicalUnit** в хеш-таблицу, в параметр **cardALexicalCode** записывается лексический код добавленного или найденного слова. Параметр **byteATable** предназначен для передачи в объект «куча» номера таблицы, с которой происходят действия.

Метод

+DeleteLexicalUnit(String strAlexicalUnit)

удаляет слово **strAlexicalUnit** из текущей хеш-таблицы.

Метод

+DeleteLexicalCode(Cardinal cardALexicalCode)

удаляет из системы хеш-таблиц слово, хранящееся под лексическим кодом **cardALexicalCode**.

Метод

+Save(Byte byteAFileType)

сохраняет текущую таблицу в открытый файл типа **byteAFileType**. Значениями параметра могут быть: **TEXT_FILE** соответствует текстовому файлу, **STREAM_FILE** соответствует потоку.

Метод

+Load(Byte byteAFileType)

восстанавливает текущую таблицу из файла. Значения **byteAFileType** могут быть такими же, как в методе **Save**.

Метод

+GetLexicalUnitList(TStringList *slstALexicalUnits)

записывает всю коллекцию слов текущей таблицы в массив строк **slstALexicalUnits**.

Рассмотрим содержание класса **THeap**, моделирующего массив «куча» (рис. П3.1).

Свойство

+arrPItems[0..*] : THeapItem

является хранилищем всех находящихся в системе хеш-таблиц слов. Индекс каждого слова в свойстве является его лексическим кодом.

Свойство

+cardPFreeItem : Cardinal

хранит индекс первой свободной ячейки в свойстве **arrPItems**.

Конструктор **+THeap()** создает объект «куча».

Деструктор **+~THeap()** разрушает объект «куча».

Метод

+AddLexicalUnit(String strALexicalUnit, Byte byteAHashTable, Cardinal cardAHashIndex, Cardinal *cardALexicalCode)

добавляет новое слово **strALexicalUnit** в свойство **arrPItems**. Полученный лексический код **cardALexicalCode** и номер таблицы **byteAHashTable** образуют обратную ссылку на положение слова в хеш-таблице.

Метод

+DeleteLexicalUnit(Cardinal cardALexicalCode)

удаляет слово, имеющее лексический код **cardALexicalCode**.

Метод

+Save(Byte byteAFileType)

сохраняет массив «куча» в открытый файл типа **byteAFileType**.

Метод

+Load(Byte byteAFileType)

восстанавливает массив «куча» из открытого файла типа **byteAFileType**.

Значениями параметра в методах **Save** и **Load** могут быть: **TEXT_FILE** соответствует текстовому файлу, **STREAM_FILE** соответствует потоку.

Содержание структуры **THeapItem**. Данная структура образует строку массива «куча» (рис. П3.1).

Поле **+strFLexicalUnit : String** является хранилищем слова в массиве «куча».

Поле **+byteFHashTable : Byte** указывает номер таблицы (тип слова), к которой относится слово **strFLexicalUnit**.

Поле **+cardFHashIndex : Cardinal** указывает индекс слова в хеш-таблице с номером **byteFHashTable**. Пара (**byteFHashTable**, **cardFHashIndex**) образует обратную ссылку на положение слова в системе хеш-таблиц (рис. П3.1).

П3.5. Диаграмма последовательности

Приведем диаграмму последовательности для операции **AddLexicalUnit()** — добавление слова в систему хеш-таблиц. Диаграмма последовательности приведена на рисунке П3.3.

В реализации системы хеш-таблиц на языке C++Builder внешний пользователь-клиент выполняет вызов метода объекта «Массив хеш-таблиц»

AddLexicalUnit (String strALexicalUnit, Byte byteATable, Cardinal *cardALexicalCode),

где **strALexicalUnit** — слово, помещаемое в хеш-таблицу с номером **byteATable**; **cardALexicalCode** — лексический код, который получает слово в результате выполнения операции.

Операция возвращает значение **true** в случае, когда слово уже имеется в таблице, и **false** в случае, если слово размещается в таблице впервые.

В свою очередь, объект «Массив хеш-таблиц» вызывает одноименный метод объекта «Хеш-таблица»

AddLexicalUnit(String strAlexicalUnit, Byte byteATable, Cardinal *cardALexicalCode),

где значения параметров совпадают с значениями одноименной функции массива хеш-таблиц.

Далее вычисляется хеш-индекс слова в таблице **byteATable**. В случае, если слово в таблице уже присутствует, то операция завершается. Если слово в таблице не обнаружено, то проверяется — есть ли место для размещения нового слова с учетом гарантийного для алгоритма хеш-поиска запаса пространства хеш-таблицы. В случае отсутствия места текущая таблица получает приращение в методе **Expantion()**. При этом происходит перехеширование всей таблицы (на диаграмме не показано). Затем вычисляется хеш-индекс поступившего слова вызовом метода **HashIndex()**.

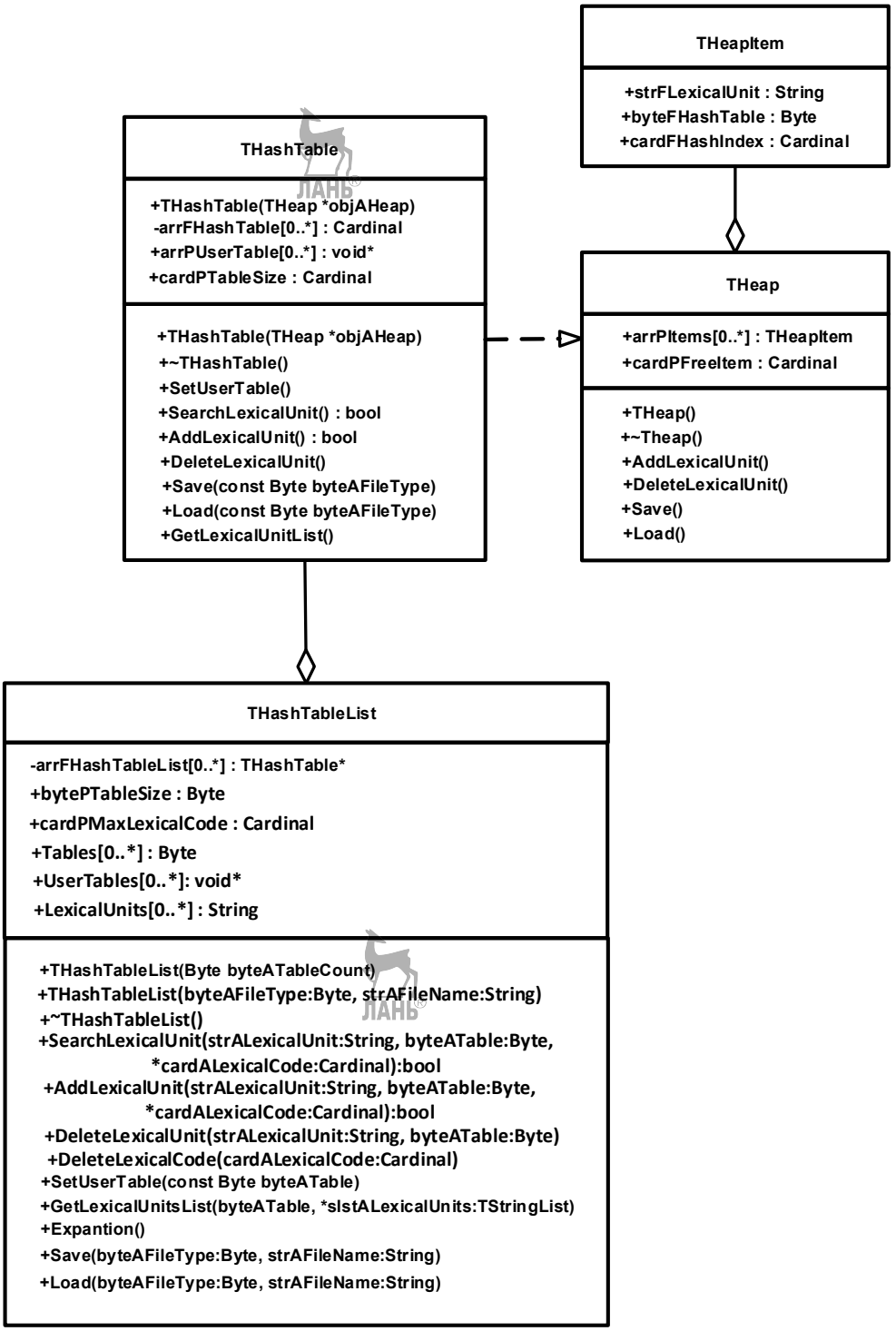


Рис. П3.2
Диаграмма классов

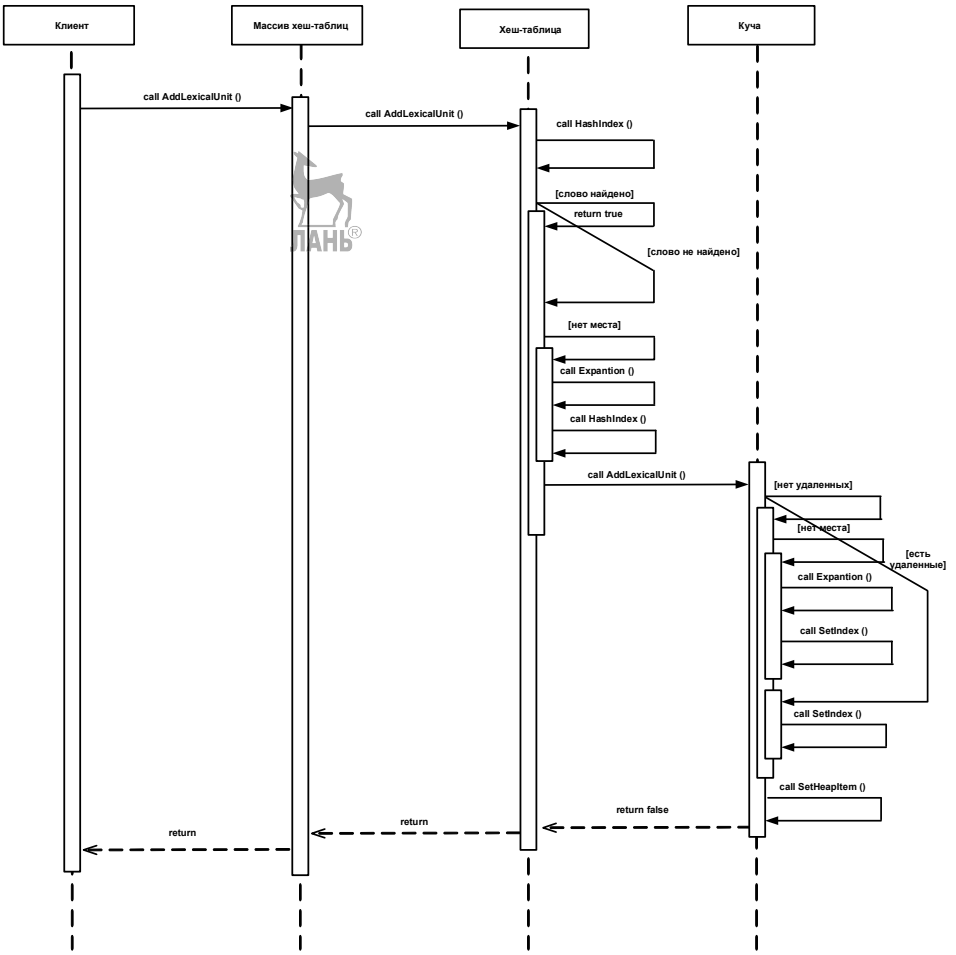


Рис. П3.3

Диаграмма последовательности операции AddLexicalUnit()

Теперь вызывается метод

AddLexicalUnit(String strALexicalUnit, Byte byteAHashTable, Cardinal cardAHashIndex, Cardinal *cardALexicalCode)

объекта «Куча», в который передаются параметрами новое слово **strALexicalUnit**, номер хеш-таблицы **byteAHashTable**, индекс **cardAHashIndex** слова в хеш-таблице. Объект «Куча» размещает слово и ссылку на хеш-таблицу, при этом вычисляется лексический код **cardALexicalCode** слова. При размещении слова проверяется, имеется ли для него место. Если место отсутствует, то вызовом метода **Expantion()** массив «Куча» увеличивает размеры.

Поясним также особенности использования освободившихся мест удаленных слов.

При удалении слова из массива «Куча» освобождается ячейка, индекс которой запоминается в специальном массиве индексов свободных ячеек. При поступлении нового слова в таблицу используются имеющиеся свободные ячейки.

П3.6. Исходный текст программы

```
struct THeapItem
{
        AnsiString  strFLexicalUnit;
        Byte byteFHashTable;
        Cardinal cardFHashIndex;
};

class THeap: public TObject
{
 private:
        std::vector<THeapItem>  vectFHeapTable;
        std::vector<Cardinal>   vectFDeleted;
        Cardinal                cardFFreeItem;
        void  Expansion(void);
        THeapItem GetItem(Cardinal cardAIndex);
        void SetItem(Cardinal cardAIndex,THeapItem New);
 public:
        __fastcall THeap();
        __fastcall ~THeap();
        __property THeapItem arrPItems[Cardinal i]={read=GetItem,write=SetItem};
        __property Cardinal cardPFreeItem={read=cardFFreeItem};
        void AddLexicalUnit(const AnsiString strALexicalUnit, const Byte byteAHashTable,
            const Cardinal cardAHashIndex, Cardinal *cardALexicalCode);
        void DeleteLexicalUnit(const Cardinal cardALexicalCode);
        void Save(const Byte byteAFileType);
        void Load(const Byte byteAFileType);
};

FILE * fileText;
ofstream outfile;
ifstream infile;
//---------------------------------------------------------------------------
__fastcall THeap::THeap()
{
        vectFHeapTable.resize(10);
        for (int i=0; i < vectFHeapTable.size(); i++)
                {
                  vectFHeapTable[i].strFLexicalUnit="";
                  vectFHeapTable[i].byteFHashTable=0;
                  vectFHeapTable[i].cardFHashIndex=0;
                }
        vectFDeleted.resize(0);
        cardFFreeItem=1;
}
//---------------------------------------------------------------------------
__fastcall THeap::~THeap()
{
```

ЛАНЬ®

```
  vectFHeapTable.resize(0);
  vectFDeleted.resize(0);
}
//---------------------------------------------------------------------
void THeap::Expansion(void)
{
        Cardinal cardVSize=vectFHeapTable.size();
        cardVSize=cardVSize+cardVSize % 10+1;
        vectFHeapTable.resize(cardVSize);
}
//---------------------------------------------------------------------
//bool THeap::IsSaved(void) { return boolIsSaved; }
//bool THeap::IsLoaded(void) { return boolIsLoaded; }
//---------------------------------------------------------------------
void THeap::AddLexicalUnit(const AnsiString strALexicalUnit, const Byte byteA-
HashTable, const Cardinal cardAHashIndex, Cardinal *cardALexicalCode)
{
        Cardinal cardVIndex;
        if (vectFDeleted.empty())
          {
                cardVIndex=cardFFreeItem;
                cardFFreeItem++;
                if(cardFFreeItem > vectFHeapTable.size()*0.9) Expansion();
          }
        else
          {
                cardVIndex=vectFDeleted[vectFDeleted.size()-1];
                vectFDeleted.resize(vectFDeleted.size()-1);
          }
        vectFHeapTable[cardVIndex].strFLexicalUnit=strALexicalUnit;
        vectFHeapTable[cardVIndex].byteFHashTable =byteAHashTable;
        vectFHeapTable[cardVIndex].cardFHashIndex =cardAHashIndex;
        *cardALexicalCode=cardVIndex;
}
//---------------------------------------------------------------------
void THeap::DeleteLexicalUnit(const Cardinal cardALexicalCode)
{
        Cardinal i;
        if (vectFDeleted.empty()) i=0; else i=vectFDeleted.size();
        vectFDeleted.resize(i+1);
        vectFDeleted[i]=cardALexicalCode;
        vectFHeapTable[cardALexicalCode].strFLexicalUnit=«»;
        vectFHeapTable[cardALexicalCode].byteFHashTable=0;
        vectFHeapTable[cardALexicalCode].cardFHashIndex=0;
}
//---------------------------------------------------------------------
void THeap::Save(const Byte byteAFileType)
{
```

```
switch(byteAFileType)
{
      case TEXT_FILE:
      try
            {
              if (vectFDeleted.empty())  fprintf(fileText,«%8u\n»,0);
              else
                 {
                    fprintf(fileText,«%8u\n»,vectFDeleted.size());
                    for (unsigned int i=0; i < vectFDeleted.size();i++)
                                          fprintf(fileText,«%8u»,vectFDeleted[i]);
                    fprintf(fileText,«%\n»);
                  }
              fprintf(fileText,«%8u\n»,cardFFreeItem);
              for (unsigned int i=1; i < cardFFreeItem; i++)
                 {
                    fprintf(fileText,«%s\n»,arrPItems[i].strFLexicalUnit);
                    fprintf(fileText,«%4u»,arrPItems[i].byteFHashTable);
                    fprintf(fileText,«%8u\n»,arrPItems[i].cardFHashIndex);
                 }
            } catch(...)
            {
               ShowMessage(«Ошибка при сохранении в текстовый файл»);
             }
          break;
      case STREAM_FILE:
          try
            {
               if (vectFDeleted.empty()) outfile << setw(8) << 0 << endl;
               else
                 {
                    outfile << setw(8) << vectFDeleted.size() << endl;
                    for (unsigned int i=0; i < vectFDeleted.size();i++)
                        outfile << setw(8) << vectFDeleted[i];
                    outfile << endl;
                 }
               outfile << setw(8) << cardFFreeItem << endl;
               for (unsigned int i=1; i < cardFFreeItem; i++)
                 {
                     outfile << arrPItems[i].strFLexicalUnit.c_str() << endl;
                   outfile << setw(4) << (int)arrPItems[i].byteFHashTable;
                   outfile << setw(8) << (int)arrPItems[i].cardFHashIndex << endl;
                 }
            }
            catch(...)
            {
                 ShowMessage(«Ошибка при сохранении в поток»);
            }
```

```
  }
}
//---------------------------------------------------------------------
void THeap::Load(const Byte byteAFileType)
{
        Cardinal cardVarrDeletedsSize;  char * s;   char c;   char s1[100];  char * s2;
        Cardinal cardVAssistant;
        vector<char> strVAssistant;
        switch(byteAFileType)
        {
           case TEXT_FILE:
                try
                {
                    fscanf(fileText,«%8u\n»,&cardVarrDeletedsSize);
                    if (cardVarrDeletedsSize != 0)
                        {
                            vectFDeleted.resize(cardVarrDeletedsSize);
                          for (unsigned int i=0; i < cardVarrDeletedsSize;i++)
                                    fscanf(fileText,«%8u»,vectFDeleted[i]);
                         }
                     fscanf(fileText,«%8u»,&cardFFreeItem);
                     vectFHeapTable.resize(cardFFreeItem+1);
                     int k=vectFHeapTable.size();
                     for (unsigned int i=1; i < cardFFreeItem; i++)
                       {
                           c=getc(fileText);      // пропуск перевода строки, так как под
форматом пропускаются сразу два /n
                          c=getc(fileText); strVAssistant.resize(0);
                          for (int j=0; c != '\n'; j++)
                              {
                                    strVAssistant.resize(j+1);
                                 strVAssistant[j]=c;
                                 c=getc(fileText);
                               }
                           int nn=strVAssistant.size();
                           s=(char *)(malloc(nn+1)); //strVAssistant.size()));
                           s[nn]='\0';
                           for (int j=0; j < nn; j++)   s[j]=strVAssistant[j];
                           vectFHeapTable[i].strFLexicalUnit=AnsiString(s);
                           fscanf(fileText,«%4u»,&cardVAssistant);
                           vectFHeapTable[i].byteFHashTable=cardVAssistant;
                           fscanf(fileText,«%8u»,&cardVAssistant);
                           vectFHeapTable[i].cardFHashIndex=cardVAssistant;
                       }
                     fscanf(fileText,«%\n»);
                }
                catch(...)
                {
```

```
                ShowMessage(«Ошибка при загрузке из текстового файла»);
            }
        break;

        case STREAM_FILE:
            try
                {
                    char charWord[255];
                    infile >> setw(8) >> cardVarrDeletedsSize;
                    if (cardVarrDeletedsSize != 0)
                      {
                       vectFDeleted.resize(cardVarrDeletedsSize);
                       for (unsigned int i=0; i < cardVarrDeletedsSize;i++)
                                    infile >> setw(8) >> vectFDeleted[i];
                      }
                    infile >> setw(8) >> cardFFreeItem;
                    vectFHeapTable.resize(cardFFreeItem+1);
                    for (unsigned int i=1;i<cardFFreeItem;i++)
                        {
                         int t;
                         infile >> setw(255) >> charWord;
                         char* strWrite=(char*)(malloc(strlen(charWord)+1));
                         strWrite[strlen(charWord)+1]='\0';
                         for(int j=0;j<strlen(charWord);j++)  strWrite[j]=charWord[j];
                         THeapItem hItm;
                         hItm.strFLexicalUnit=AnsiString(strWrite);
                         infile >> setw(4) >> t; hItm.byteFHashTable=t;
                         infile >> setw(8) >> t; hItm.cardFHashIndex=t;
                         arrPItems[i]=hItm;
                          }
                    }
                    catch(...)
                    {
                         ShowMessage(«Ошибка при загрузке из потока»);
                     }
        }
}
//---------------------------------------------------------------------
THeapItem THeap::GetItem(Cardinal cardAIndex)
{
  return  vectFHeapTable[cardAIndex];
}
//---------------------------------------------------------------------
void THeap::SetItem(Cardinal cardAIndex,THeapItem NewItem)
{
  vectFHeapTable[cardAIndex]=NewItem;
}
//---------------------------------------------------------------------
```

```
class THashTable: public TObject
{
 private:
        std::vector <unsigned int> vectFHashTable;
        Cardinal       cardFCurrentPrimeNumber;
        Cardinal       cardFItemReserve;
        Cardinal       cardFHashIndex;
        std::vector <void*>          vectFUserTable;
        THeap                  *objFHeap;
        void             HashIndex(const AnsiString strALexicalUnit);
        void             ReHashing();
        void             Expansion(void);
        void*            GetUserPointer(Cardinal cardALexicalCode);
        void             SetUserPointer(Cardinal cardALexicalCode, void * ptrANewPoint);
        Cardinal GetTableSize() {return vectFHashTable.size();};
        Cardinal NextPrimeNumber(const Cardinal cardAOldPrimeNumber);
        Cardinal LinearRehashing(Cardinal h, const AnsiString strALexicalUnit);
 public:
        THashTable(THeap *objAHeap);
        __fastcall ~THashTable();
        __property void * arrPUserTable[Cardinal cardILexicalCode]=
                                                        { read=GetUserPointer,
write=SetUserPointer };
        __property Cardinal cardPTableSize={read=GetTableSize};
        void  SetUserTable();
        bool  SearchLexicalUnit(const AnsiString strALexicalUnit, Cardinal
*cardALexicalCode);
        bool  AddLexicalUnit(const AnsiString strALexicalUnit, const Byte byteATable,
                                                        Cardinal *cardALexicalCode);
        void  DeleteLexicalUnit(const AnsiString strALexicalUnit);
        void  DeleteLexicalCode(const Cardinal cardALexicalCode);
        void  Save(const Byte byteAFileType);
        void  Load(const Byte byteAFileType);
        void  GetLexicalUnitList(TStringList *slstALexicalUnits);
};

Cardinal THashTable::NextPrimeNumber(const Cardinal cardAOldPrimeNumber)
{
        // вычисляет следующее за cardAOldPrimeNumber простое число
cardVNextPrimeNumber
        // отстоящее не менее, чем на 10 процентов от cardAOldPrimeNumber
    Cardinal cardVLowerBound, cardVUpperBound, cardVNextPrimeNumber;
    bool boolVIsDivisor;
    cardVNextPrimeNumber=cardAOldPrimeNumber+cardAOldPrimeNumber/10+1;
// увеличиваем на 10%
```

```
    if ((cardVNextPrimeNumber % 2) == 0) cardVNextPrimeNumber++;
        // рассматриваем только нечетные числа, так как простые среди них
    do {
         boolVIsDivisor=true; cardVNextPrimeNumber=cardVNextPrimeNumber+2;
         cardVLowerBound=3; cardVUpperBound=cardVNextPrimeNumber/3+1; //
диапазон делителей
         while(boolVIsDivisor && (cardVLowerBound < cardVUpperBound))
           {
              if ((cardVNextPrimeNumber % cardVLowerBound) == 0) boolVIsDivi-
sor=false;
              else cardVLowerBound=cardVLowerBound+2;
                 }
        } while(!boolVIsDivisor);
    return cardVNextPrimeNumber;
}
//---------------------------------------------------------------------------
Cardinal HashFunction(const AnsiString strALexicalUnit)
//суммируем коды символов строки как последовательность 3-х байтных чисел
{
     unsigned int h=0;
     for(int i=1; i<=strALexicalUnit.Length(); i++)
       {
          switch(i % 3)
          {
               case 1: h=h+strALexicalUnit[i]; break;
              case 2: h=h+strALexicalUnit[i]*256; break;
              case 0: h=h+strALexicalUnit[i]*65536; break;
          }
       }
     return h;
}
//---------------------------------------------------------------------------
Cardinal THashTable::LinearRehashing(Cardinal h, const AnsiString strALexicalUnit)
// линейное рехеширование   hi=h (i+1) mod <текущий размер таблицы> , i=1,2,...
{
    if (h == 0) h=vectFHashTable.size()/3;
    else if (h == 1) h=vectFHashTable.size() * 3/4;
    Cardinal i=1, hi=h;
    bool boolVFinish=false;
    do {
        if(vectFHashTable[hi] == 0) boolVFinish=true;
        else
           if(objFHeap–>arrPItems[vectFHashTable[hi]].strFLexicalUnit == strALexi-
calUnit)
                boolVFinish=true;
           else
              {
                i++;
```

```
                hi=(h – 1) * i % (vectFHashTable.size() – 1)+1;
              }
        } while(!boolVFinish);
    return hi;
}
//---------------------------------------------------------------------------
void THashTable::HashIndex(const AnsiString strALexicalUnit)
{
        // вычисление Hash-индекса строки strALexicalUnit
        // результат заносится в cardFHashIndex
        unsigned int h;
        bool boolVFinish;
        h=HashFunction(strALexicalUnit) % (arrFHashTable.size() – 1)+1;
        cardFHashIndex=LinearRehashing(h, strALexicalUnit);
}
//---------------------------------------------------------------------------
void THashTable::ReHashing()
{
        bool IsPointer;
        unsigned int i, j;
        vector <Cardinal> cardarrVHashTableImage;
        vector <void*>  arrVUserTableImage;
        IsPointer=!vectFUserTable.empty();
        cardarrVHashTableImage.resize(vectFHashTable.size());
        if(IsPointer) arrVUserTableImage.resize(vectFHashTable.size());
        for(i=0;i<vectFHashTable.size();i++)
        {
                cardarrVHashTableImage[i]=vectFHashTable[i];
                if(IsPointer) arrVUserTableImage[i]=vectFUserTable[i];
        }
        vectFHashTable.resize(0);
        if(IsPointer) vectFUserTable.resize(0);
        vectFHashTable.resize(cardFCurrentPrimeNumber+1);
        if(IsPointer) vectFUserTable.resize(cardFCurrentPrimeNumber+1);
        for(i=0;i<cardarrVHashTableImage.size();i++)
        {
                if(cardarrVHashTableImage[i])
                {
                        j=cardarrVHashTableImage[i];
                        HashIndex(objFHeap–>arrPItems[j].strFLexicalUnit);
                        vectFHashTable[cardFHashIndex]=j;
                        if(IsPointer) vectFUserT-
able[cardFHashIndex]=arrVUserTableImage[i];
                        objFHeap–>arrPItems[j].cardFHashIndex=cardFHashIndex;
                }
        }
  cardarrVHashTableImage.resize(0);
  if(IsPointer) arrVUserTableImage.resize(0);
```

```
}
//---------------------------------------------------------------------------
void THashTable::Expansion(void)
{
  cardFCurrentPrimeNumber=NextPrimeNumber(cardFCurrentPrimeNumber);
  ReHashing();
}
//---------------------------------------------------------------------------
void* THashTable::GetUserPointer(Cardinal cardALexicalCode)
{
  if (objFHeap–>arrPItems[cardALexicalCode].cardFHashIndex >= cardPTableSize)
        {ShowMessage(«Индекс пользовательского массива вышел за
диапазон!»);return NULL;}
  else return vectFUserTable[objFHeap–
>arrPItems[cardALexicalCode].cardFHashIndex];
}
//---------------------------------------------------------------------------
void THashTable::SetUserPointer(Cardinal cardALexicalCode, void * ptrANewPoint)
{
  if (objFHeap–>arrPItems[cardALexicalCode].cardFHashIndex >= cardPTableSize)
ShowMessage(«Индекс пользовательского массива вышел за диапазон!»);
  else
        {
           vectFUserTable[objFHeap–
>arrPItems[cardALexicalCode].cardFHashIndex]=ptrANewPoint;
         }
}
//---------------------------------------------------------------------------
void  THashTable::SetUserTable()
{
  vectFUserTable.resize(cardPTableSize);
}
//---------------------------------------------------------------------------
bool THashTable::SearchLexicalUnit(const AnsiString strAlexicalUnit, Cardinal *
cardALexicalCode)
{
        HashIndex(strAlexicalUnit);
        if(vectFHashTable[cardFHashIndex] == 0) return false;
        else
          {
                *cardALexicalCode=vectFHashTable[cardFHashIndex];
                return true;
          }
}
//---------------------------------------------------------------------------
bool THashTable::AddLexicalUnit(const AnsiString strAlexicalUnit, const Byte
byteATable, Cardinal *cardALexicalCode)
{
```

```
        HashIndex(strAlexicalUnit);
        if(vectFHashTable[cardFHashIndex])
         {
              *cardALexicalCode=vectFHashTable[cardFHashIndex];
              return true;
         }
        else
         {
            if((cardFItemReserve+2) > (cardPTableSize * 0.9))
             {
               Expansion();
               HashIndex(strAlexicalUnit);
             }
             objFHeap–>AddLexicalUnit(strAlexicalUnit, byteATable,
                                                    cardFHashIndex,
&vectFHashTable[cardFHashIndex]);
            *cardALexicalCode=vectFHashTable[cardFHashIndex];
            cardFItemReserve++;
            return false;
           }
}
//---------------------------------------------------------------------------
void THashTable::DeleteLexicalUnit(const AnsiString strAlexicalUnit)
{
  HashIndex(strAlexicalUnit);
  if(vectFHashTable[cardFHashIndex] != 0)
   {
     if(!vectFUserTable.empty())
      {
       if(vectFUserTable[cardFHashIndex] != NULL)
                              ShowMessage(«Удаление из таблицы связанного дан-
ного»);
         else
         {
            objFHeap–>DeleteLexicalUnit(vectFHashTable[cardFHashIndex]);
            vectFHashTable[cardFHashIndex]=0;
            cardFItemReserve--;
            ReHashing();
         }
      }
     else
       {
          objFHeap–>DeleteLexicalUnit(vectFHashTable[cardFHashIndex]);
          vectFHashTable[cardFHashIndex]=0;
          cardFItemReserve--;
          ReHashing();
       }
   }
```

```
}
//---------------------------------------------------------------------------
void THashTable::DeleteLexicalCode(const unsigned int cardALexicalCode)
{
   unsigned int VHashIndex;
   VHashIndex=objFHeap–>arrPItems[cardALexicalCode].cardFHashIndex;
   if(vectFHashTable[VHashIndex])
      if(vectFUserTable.size())
        if(vectFUserTable[VHashIndex]) ShowMessage(«Удаление из таблицы свя-
занного данного»);
         else
              {
              objFHeap–>DeleteLexicalUnit(cardALexicalCode);
              vectFHashTable[VHashIndex]=0;
              cardFItemReserve--;
              ReHashing();
              }
        else
          {
          objFHeap–>DeleteLexicalUnit(cardALexicalCode);
          vectFHashTable[VHashIndex]=0;
          cardFItemReserve--;
          ReHashing();
        }
}
//---------------------------------------------------------------------------
void THashTable::Save(const Byte byteAFileType)
{
   switch(byteAFileType)
  {
     case TEXT_FILE:
        try
        {
          fprintf(fileText,«%8u\n»,cardPTableSize);
          fprintf(fileText,«%8u\n»,cardFItemReserve);
          for(Cardinal i=1; i < cardPTableSize; i++)
          fprintf(fileText,«%8u»,vectFHashTable[i]);
          fprintf(fileText,«\n»);
          }
        catch(...)
        {
             ShowMessage(«Ошибка при сохранении в текстовый файл»);
        }
        break;
     case STREAM_FILE:
        try {
                outfile << setw(8) << cardPTableSize << endl;
                outfile << setw(8) << cardFItemReserve << endl;
```

```
                for(Cardinal i=1; i < cardPTableSize; i++)
                outfile << setw(8) << vectFHashTable[i];
                outfile << endl;
               }
        catch(...)
        {
            ShowMessage(«Ошибка при сохранении в поток»);
        }
    }
}
//---------------------------------------------------------------------------
void THashTable::Load(const Byte byteAFileType)
{
   switch(byteAFileType)
  {
      case TEXT_FILE:
        try
        {
           Cardinal cardFHashTableSize;
           fscanf(fileText,«%8u\n»,&cardFHashTableSize);
           vectFHashTable.resize(cardFHashTableSize);
           fscanf(fileText,«%8u\n»,&cardFItemReserve);
           for(Cardinal i=1; i < cardPTableSize; i++)
                fscanf(fileText,«%8u»,&vectFHashTable[i]);
           fscanf(fileText,«\n»);
         }
         catch(...)
         {
             ShowMessage(«Ошибка при загрузке из текстового файла»);
         }
        break;
    case STREAM_FILE:
        try
        {
           Cardinal cardFHashTableSize;
           infile >> setw(8) >> cardFHashTableSize;
           vectFHashTable.resize(cardFHashTableSize);
           infile >> setw(8) >> cardFItemReserve;
           for(unsigned int i=1;i<cardPTableSize;i++)
                infile >> setw(8) >> vectFHashTable[i];
          }
        catch(...)
        {
            ShowMessage(«Ошибка при загрузке из потока»);
        }
    }
}
//---------------------------------------------------------------------------
```

ЛАНЬ®

```
void THashTable::GetLexicalUnitList(TStringList *slstALexicalUnits)
{
    bool boolVSortedOld;
    unsigned int i;
    slstALexicalUnits–>BeginUpdate();
    boolVSortedOld=slstALexicalUnits–>Sorted;
    slstALexicalUnits–>Sorted=false;
    for(i=1;i<cardPTableSize;i++)
        if(vectFHashTable[i])
                slstALexicalUnits–>Add(objFHeap–
>arrPItems[vectFHashTable[i]].strFLexicalUnit);
    slstALexicalUnits–>EndUpdate();
    slstALexicalUnits–>Sorted=boolVSortedOld;
}
//---------------------------------------------------------------------------
THashTable::THashTable(THeap *objAHeap)
{
     vectFHashTable.resize(8);
     objFHeap=(THeap*)objAHeap;
     cardFCurrentPrimeNumber=7;
     cardFItemReserve=0;
}
//---------------------------------------------------------------------------
__fastcall THashTable::~THashTable()
{
  vectFHashTable.resize(0);
}
//---------------------------------------------------------------------------


class THashTableList : public TObject
{
    private:
        std::vector <THashTable*> arrFHashTableList;
        THeap  *objFHeap;
        Cardinal   GetMaxLexicalCode(void);
        Byte  GetTable(Cardinal cardALexicalCode);
        void* GetUserData(Cardinal cardALexicalCode);
        void   SetUserData(Cardinal cardALexicalCode, void* ptrAUserData);
        AnsiString  GetLexicalUnit(Cardinal cardALexicalCode);
        Byte   GetTableSize(void);
         Byte   InputTableCount(const AnsiString strAFileName);
    public:
         THashTableList(Byte byteATableCount);
         THashTableList(const Byte byteAFileType, const AnsiString strAFileName);
         __fastcall ~THashTableList();
         __property Byte bytePTableSize={ read=GetTableSize, default=false };
         __property Cardinal cardPMaxLexicalCode={ read=GetMaxLexicalCode };
```

```
        __property Byte Tables[Cardinal cardILexicalCode]={ read=GetTable };
        __property void* UserTables[Cardinal cardILexicalCode]=
                                                        { read=GetUserData,
write=SetUserData };
        __property AnsiString LexicalUnits[Cardinal cardILexicalCode]=
                                                         { read=GetLexicalUnit };
        bool   SearchLexicalUnit(const AnsiString strALexicalUnit, const Byte byteATable,
                                                                    Cardinal
*cardALexicalCode);
        bool   AddLexicalUnit (const AnsiString strALexicalUnit, const Byte byteATable,
                                                                      Cardinal
*cardALexicalCode);
        void   DeleteLexicalUnit (const AnsiString strALexicalUnit, const Byte byteATable);
        void    DeleteLexicalCode(const Cardinal cardALexicalCode);
        void   SetUserTable(const Byte byteATable);
        void   GetLexicalUnitsList(const Byte ATable, TStringList *slstALexicalUnits);
        void   Expantion();
        void   Save(const Byte byteAFileType, const AnsiString strAFileName);
        void   Load(const Byte byteAFileType, const AnsiString strAFileName);
};
//---------------------------------------------------------------------------
Byte THashTableList::GetTable(Cardinal cardALexicalCode)
          { return objFHeap->arrPItems[cardALexicalCode].byteFHashTable; }
//---------------------------------------------------------------------------
void * THashTableList::GetUserData(Cardinal cardALexicalCode)
{
        if((0 < cardALexicalCode) && (cardALexicalCode < this->objFHeap-
>cardPFreeItem))
                return arrFHashTableList[Tables[cardALexicalCode]]-
>arrPUserTable[cardALexicalCode];
        else
          {
                ShowMessage("Неверно задан лексический код");
                return NULL;
          }
}
//---------------------------------------------------------------------------
void  THashTableList::SetUserData(Cardinal cardALexicalCode, void * ptrAUser-
Data)
{
  if ((0 < cardALexicalCode) && (cardALexicalCode < objFHeap->cardPFreeItem))
          {
                THashTable * hT = arrFHashTableList[Tables[cardALexicalCode]];
                int size = hT->cardPTableSize;
                if (hT->cardPTableSize > 0)
               arrFHashTableList[Tables[cardALexicalCode]]-
>arrPUserTable[cardALexicalCode] =
```

```
ptrAUserData;
                else ShowMessage(L"Попытка записи адреса в несозданный массив
указателей!");
          }
  else ShowMessage(L"Неверно задан лексический код!");
}
//---------------------------------------------------------------------------
AnsiString THashTableList::GetLexicalUnit(Cardinal cardALexicalCode)
{
        if((0 < cardALexicalCode) && (cardALexicalCode < objFHeap->cardPFreeItem))
                                        return objFHeap-
>arrPItems[cardALexicalCode].strFLexicalUnit;
        else
                {
                 ShowMessage("Неверно задан лексический код");
                 return "";
                  }
}
//---------------------------------------------------------------------------
bool THashTableList::SearchLexicalUnit(const AnsiString strALexicalUnit, const
Byte byteATable, Cardinal *cardALexicalCode)
                { return arrFHashTableList[byteATable]-
>SearchLexicalUnit(strALexicalUnit,
                                                                        car-
dALexicalCode); }
//---------------------------------------------------------------------------
bool THashTableList::AddLexicalUnit (const AnsiString strALexicalUnit, const Byte
byteATable, Cardinal *cardALexicalCode)
{
        return arrFHashTableList[byteATable]->AddLexicalUnit(strALexicalUnit,
byteATable,
                                                                        car-
dALexicalCode);
}
//---------------------------------------------------------------------------
void THashTableList::DeleteLexicalUnit (const AnsiString strALexicalUnit, const
Byte byteATable)
{
  arrFHashTableList[byteATable]->DeleteLexicalUnit(strALexicalUnit);
}
//---------------------------------------------------------------------------
void THashTableList::DeleteLexicalCode(const unsigned int cardALexicalCode) {
        unsigned short T = objFHeap->arrPItems[cardALexicalCode].byteFHashTable;
        arrFHashTableList[T]->DeleteLexicalCode(cardALexicalCode);
}
//---------------------------------------------------------------------------
Cardinal THashTableList::GetMaxLexicalCode(void)
{
```

```
        return objFHeap->cardPFreeItem - 1;
}
void THashTableList::SetUserTable(const Byte byteATable)
        { arrFHashTableList[byteATable]->SetUserTable(); }
//---------------------------------------------------------------------------
void THashTableList::GetLexicalUnitsList(const Byte ATable, TStringList
*slstALexicalUnits)
{
        arrFHashTableList[ATable]->GetLexicalUnitList(slstALexicalUnits);
}
//---------------------------------------------------------------------------
void THashTableList::Expantion()
{
        arrFHashTableList.resize(bytePTableSize + 1);
        arrFHashTableList[bytePTableSize - 1] = new THashTable(objFHeap);
}
//---------------------------------------------------------------------------
void THashTableList::Save(const Byte byteAFileType, const AnsiString strAFile-
Name)
{
        switch(byteAFileType)
        {
                case TEXT_FILE:
                        try
                         {
                                fileText = fopen(strAFileName.c_str(), "wt");  // wt - запись
в текстовый файл
                                fprintf(fileText,"%4u\n",bytePTableSize);
                                objFHeap->Save(TEXT_FILE);
                                for(Byte i = 1; i < bytePTableSize; i++)
                                        arrFHashTableList[i]->Save(TEXT_FILE);
                                fclose(fileText);
                         }
                         catch(...)
                         {
                                ShowMessage("Ошибка при сохранении в текстовый
файл");
                         }
                        break;

                case STREAM_FILE:
                        try
                         {
                                outfile.open(strAFileName.c_str());
                                if(!outfile)
                                  {
                                        ShowMessage("Файл не удается создать");
                                        return;
```

```
                        }
                        outfile << setw(4) << bytePTableSize << endl;
                        objFHeap->Save(STREAM_FILE);
                        for(Byte i = 1; i < bytePTableSize; i++)
                                arrFHashTableList[i]->Save(STREAM_FILE);
                        outfile.close();
                }
                catch(...)
                {
                        ShowMessage("Ошибка при сохранении в поток");
                }
                break;
        }
}
//---------------------------------------------------------------------------
void THashTableList::Load(const Byte byteAFileType, const AnsiString strAFileame)
{

        switch(byteAFileType)
        {
                case TEXT_FILE:
                        try
                        {
                                Byte byteFTableCount;
                                fileText = fopen(strAFileName.c_str(), "rt");     // rt - только
для чтения
                                fscanf(fileText,"%4u\n",&byteFTableCount);
                                objFHeap->Load(TEXT_FILE);
                                for(Byte i = 1; i < bytePTableSize; i++)
                                        arrFHashTableList[i]->Load(TEXT_FILE);
                                fclose(fileText);
//                              boolFIsLoaded = true;
                        }
                        catch(...)
                        {
                        ShowMessage("Ошибка при восстановлении из файла хеш-
таблиц!");
//                      boolFIsLoaded = false;
                        }
                        break;
                case STREAM_FILE:
                        try {
                                Byte byteFTableCount;
                                infile.open(strAFileName.c_str());
                                infile >> byteFTableCount;
                                objFHeap->Load(STREAM_FILE);
                                for(int i=1;i< bytePTableSize;i++)
                                        arrFHashTableList[i]->Load(STREAM_FILE);
```

```
                        infile.close();
                }
                catch (...)
                {
                        ShowMessage("Ошибка при восстановлении из потока");
                }
        break;
    }
}
//---------------------------------------------------------------------------
Byte THashTableList::GetTableSize(void) { return arrFHashTableList.size(); }
//---------------------------------------------------------------------------
THashTableList::THashTableList(Byte byteATableCount)
{
    arrFHashTableList.resize(byteATableCount+1);
    objFHeap = new THeap();
    for(int i=1;i<=byteATableCount;i++) arrFHashTableList[i] = new
THashTable(objFHeap);
}
//---------------------------------------------------------------------------
THashTableList::THashTableList(const Byte byteAFileType, const AnsiString
strAFileName)
{
    Byte byteVTableSize;
    switch (byteAFileType)
    {
        case TEXT_FILE:
            try
                {
                    fileText = fopen(strAFileName.c_str(), "rt");
                    fscanf(fileText,"%4u\n",&byteVTableSize);
                    fclose(fileText);
                    arrFHashTableList.resize(byteVTableSize);
                    objFHeap = new THeap();
                    for(int i = 1;i < bytePTableSize;i++)
                        arrFHashTableList[i] = new
THashTable(objFHeap);
                    Load(byteAFileType, strAFileName);
                }
            catch(...)
                {
                   ShowMessage("Ошибка при восстановлении из файла
хеш-таблиц!");
                }
            break;

        case STREAM_FILE:
            try
```

```
                {
                        infile.open(strAFileName.c_str());
                        infile >> setw(4) >> byteVTableSize;
                        infile.close();
                        arrFHashTableList.resize(byteVTableSize);
                        objFHeap = new THeap();
                        for(int i = 1;i < bytePTableSize;i++)
                              arrFHashTableList[i] = new THashTable(objFHeap);
                        Load(byteAFileType, strAFileName);
                }
                catch(...)
                {
                      ShowMessage("Ошибка при восстановлении из потока
хеш-таблиц!");
                }
                break;
        }
}
//---------------------------------------------------------------------------
__fastcall THashTableList::~THashTableList()
{
        for (Byte i= 1; i < bytePTableSize; i++) arrFHashTableList[i]->~THashTable();
        arrFHashTableList.resize(0);
        objFHeap->~THeap();
}
//---------------------------------------------------------------------------
```

П3.7. Использование программы

Для использования программы необходимо подключить файлы массива хеш-таблиц к своей программе, объявить и создать объект класса **THashTableList**.

1. Конструкторы и деструктор. Первоначально система хеш-таблиц создается конструктором:

THashTableList(Byte byteATableCount);,

где **byteATableCount** — количество создаваемых хеш-таблиц.

Имеется еще один конструктор, который позволяет создать систему таблиц по предварительно сохраненной в файле:

THashTableList(Byte byteAFileType, String strAFileName),

где **byteAFileType** — тип файла, **strAFileName** — имя файла.

Удаляются ранее созданные таблицы с помощью деструктора: **~THashTableList()**. В процессе работы можно к существующей системе хеш-таблиц добавить еще одну с помощью метода**: void Expantion()**.

2. После того как таблицы созданы, можно узнать их количество с помощью свойства:

__property Byte bytePTableCount.

3. Созданную во время работы программы систему таблиц можно сохранить в файле с помощью метода:

void Save(Byte byteAFileType, String strAFileName).

Ранее сохраненную в файле систему таблиц можно восстановить:

void Load(Byte byteAFileType, String strAFileName).

Параметры:

— **byteAFileType**: TEXT_FILE — текстовый файл в стиле C, STREAM_FILE — поток в стиле C++;

— **strAFileName** — имя физического файла с путями, по умолчанию файл создается в одной папке с exe-файлом.

4. Традиционно значение каждого помещаемого в таблицу слова кодируется в виде токена (Nтаблицы, Nстроки). В этой программе токен моделируется с помощью понятия **Cardinal LexicalCode**. **LexicalCode** несет в себе всю каноническую информацию. Для этого нужно использовать свойства:

— **__property Byte Tables[Cardinal cardILexicalCode]** — возвращает номер таблицы;

— **__property String LexicalUnits[Cardinal cardILexicalCode]** — возвращает значение **strLexicalUnit** слова.

Максимальное значение лексического кода можно узнать с помощью свойства

__property Cardinal cardPMaxLexicalCode.

Это значение указывает на общее число хранимых в системе хеш-таблиц слов.

5. Для работы с таблицами используются методы:

```
bool SearchLexicalUnit(String strALexicalUnit, Byte byteATable,
                                                Cardinal *cardALexicalCode),
bool AddLexicalUnit (String strALexicalUnit, Byte byteATable,
                                                Cardinal *cardALexicalCode),
void DeleteLexicalUnit (String strALexicalUnit, const Byte byteATable),
void DeleteLexicalCode(Cardinal cardALexicalCode).
```

Параметры:

— **strALexicalUnit** — значение слова;

— **byteATable** — номер таблицы;

— **cardALexicalCode** — лексический код слова.

6. С каждой хеш-таблицей можно связать массив указателей типа **void ***. Первоначально массив указателей создается методом:

void SetUserTable(const Byte byteATable).

Параметр **byteATable** указывает номер таблицы, при которой создается массив указателей.

Для каждого слова работа с указателем осуществляется с помощью свойства:

__property void* UserTables[Cardinal cardILexicalCode].

Индексом **cardILexicalCode** в векторном свойстве является лексический код слова.

Рассмотрим пример записи значения в пользовательскую таблицу системы хеш-таблиц **htl** при известном лексическом коде **LexCode**.

```
TPoint * p=new TPoint();
p–>X=1;  p–>Y=2;
TPoint **q1=&p1;
htl–>UserTables[LexCode]=(void*)(*q1);.
```

Извлечь значения из пользовательской таблицы можно следующим образом:

```
TPoint * p2=(TPoint *)(htl–>UserTables[LexCode]);.
```

Из каждой таблицы **ATable** можно прочитать коллекцию, записанных в нее слов. Коллекция возвращается в параметр **slstALexicalUnits**:

```
void GetLexicalUnitsList(const Byte ATable, TStringList *slstALexicalUnits);.
```

ЛИТЕРАТУРА

1. Borland C++Builder 6. Руководство разработчика / Б. Сворт [и др.]. — М. : Издат. дом «Вильямс», 2003. — 976 с.
2. Borland C++Builder 5. Энциклопедия программиста : пер. с англ. / Ч. Калверт, К. Рейнсдорф. — Киев : ДиаСофт, 2001. — 944 с.
3. *Архангельский*, *А. Я.* Программирование в C++Builder. — М. : Бином-Пресс, 2010. — 1230 с.
4. *Бобровский*, *С.* Технологии C++Builder. Разработка приложений для бизнеса. — СПб. : Питер, 2007. — 560 с.
5. *Буч*, *Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++ : пер. с англ. — М. : Бином ; СПб. : Невский Диалект, 1998. — 560 с.
6. *Вальпа*, *О. Д.* Borland C++Builder. Экспресс-курс. — СПб. : БХВ-Петербург, 2006. — 224 с.
7. *Верма*, *Р. Д.* Справочник по функциям WinAPI. — М. : Горячая линия — Телеком, 2002.
8. *Гудлиф*, *П.* Ремесло программиста. Практика написания хорошего кода : пер. с англ. — СПб. : Символ-Плюс, 2009. — 704 с.
9. *Ермолаев*, *В. Г.* C++Builder: Книга рецептов / В. Г. Ермолаев, Т. С. Сорока. — М. : КУДИЦ-ОБРАЗ, 2006. — 208 с.
10. *Замулин*, *А. В.* Типы данных в языках программирования и базах данных : монография. — Новосибирск : Наука, 1987. — 151 с.
11. *Кауфман*, *В. Ш.* Языки программирования. Концепции и принципы. — М. : ДМК Пресс, 2010. — 464 с.
12. Borland C++Builder 5. Энциклопедия программиста : пер. с англ. / Ч. Колверт, К. Рейнсдорф. — М. : ДиаСофт, 2001. — 944 с.
13. *Культин*, *Н. Б.* C++Builder в задачах и примерах. — СПб. : БХВ-Петербург, 2005. — 336 с.
14. *Культин*, *Н. Б.* Самоучитель C++Builder. — СПб. : БХВ-Петербург, 2004. — 320 с.
15. Объектно-ориентированное программирование / Г. С. Иванова [и др.]. — М. : Изд-во МГТУ им. Н. Э. Баумана, 2001. — 320 с.
16. *Майер*, *Б.* Объектно-ориентированное конструирование программных систем. — М. : Русская редакция, 2005. — 1204 с.
17. *Мартин*, *Р.* Чистый код: создание и рефакторинг. Библиотека программиста : пер. с англ. — СПб. : Питер, 2010. — 484 с.
18. *Пахомов*, *Б.* Самоучитель C/C++ и C++Builder 2007. — СПб. : БХВ-Петербург, 2008. — 672 с.
19. *Пол*, *А.* Объектно-ориентированное программирование с использованием C++. — М. : ДиаСофт Лтд, 2014. — 480 с.
20. *Сиддхартра*, *Р.* Освой самостоятельно C++ за 21 день. — М. : Вильямс, 2013. — 688 с.
21. *Федоренко*, *Ю. П.* Алгоритмы и программы на C++Builder. — М. : ДМК Пресс, 2010. — 544 с.
22. *Шамис*, *В.* Borland C++ Builder 6. — СПб. : Питер, 2003. — 800 с.
23. *Щупак*, *Ю. А.* Win32 API. Эффективная разработка приложений. — СПб. : Питер, 2007. — 572 с.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Абстрагирование *146*
Абстракция *12*, *146*
Агент *26*
Агрегация *28*
Актер *5*, *26*, *97*
Активное окно *206*, *212*, *255*
Алгебра слов *175*, *190*
Анализ вариантов *141*
Анализ поведения *139*
Анализ приложений *140*
Ассоциация *34–35*, *45*, *69*, *79*

Библиотека визуальных компонентов *145*, *203*, *459*, *511*, *535*

Векторная графика *543*
Ветвление потока управления *101*
Вид параметра операции *65*
Видимость *26*
Внешний вид визуального компонента *549*
Временные ограничения *102*, *106*
Вызовы виртуальных методов *532*
Вызовы статических методов *532*
Выравнивание визуального компонента *548*
Выражение *63*, *65*, *426–429*
Выражение типа возвращаемого операцией значения *65–66*
Выражение типа параметра операции *65*

Графические компоненты *515*
Графический интерфейс устройства *240*

Декомпозиция *120–121*, *123*, *125*, *127*, *366*
Дескриптор *210*
Дескриптор контекста устройства *211*
Деструктор *21*
Диаграмма вариантов *56*
Диаграмма деятельности *57*, *83–85*
Диаграмма классов *56–58*, *72*, *74*, *83*, *420*, *451*
Диаграмма компонентов *57*
Диаграмма последовательности *57, 95*, *102*
Диаграмма развертывания *57*
Диаграмма состояний *57*
Динамическая модель *128*
Динамические ошибки *478–479*
Директива #include *366–367*, *380*
Дорожки *90–91*

Жизненный цикл *123–124*

Защищенный код *481*, *483*
Значение параметра операции по умолчанию *65*

Идентификация механизмов *143*
Идентичность объекта *22–23*, *48*
Иерархичность *150*
Иерархия наследования *150*, *493*, *533*, *587*, *598–599*
Имя атрибута *61–64*
Имя класса *59–61*
Имя операции *64–65*, *101*
Имя параметра операции *65*
Индексированное семейство множеств *172, 174*
Инкапсуляция *3*, *7*, *17*, *32*, *145*, *147–149*
Инстанцирование *34*, *46–47*
Интерфейс *3*, *30–32*, *47*
Интерфейс класса *31–32*, *356, 360*, *393*
Интерфейс объекта *360*
Интерфейс прикладного программирования *203*
Исключительная ситуация как класс *490–491*
Использование класса *218*, *317*
Использующее вхождение имени *363*
Исходное значение атрибута *61*, *64*

Карта сообщения *464*
Квантор видимости атрибута *61*
Квантор видимости операции *65*
Кисть *229*, *238*, *240*, *628*

ЛАНЬ®

Класс *29–30*
Класс TApplication *572*
Класс TApplicationEvents *579*
Класс TCanvas *542*, *627*
Класс TClipboard *585*, *587*
Класс TComponent *534*, *538*
Класс TControl *534*, *546–547*, *551*
Класс TCustomCanvas *542*
Класс TCustomControl *534*, *570*
Класс TForm *598*
Класс TGraphicControl *535*, *571*
Класс TIniFile *593–594*
Класс TObject *39*, *290–295*, *533*, *536*
Класс TPersistent *533*, *537*
Класс TScreen *581–582*
Класс TWinControl *534*, *561*, *567*, *598*
Класс окна *214*, *217*, *224*, *262–263*
Классификация *29–30*, *132–135*, *482*
Классическая категоризация *135–137*
Клиент *21*, *24*, *26–27*, *45*, *75*, *99*, *102*, *146*
Клиентская область *245*, *248*
Клиентские координаты *248*
Ключевая абстракция *141–142*
Компонент-класс *513*, *539*
Компонент-объект *513*, *539*
Конструктор *19*, *21*, *23*, *30*
Контейнерные компоненты *516*
Контекст устройства *238*, *241–243*
Концептуальная кластеризация *137*

Линия жизни объекта *96*
Логические ошибки *478*

Метакласс *34*, *47*, *286*, *590*
Метод *13*, *21*, *310*
Методы класса *47*, *303–304*
Методы компонентов *531*
Многоуровневый защищенный блок *495*
Модификатор доступа *331*
Модификатор наследования *329*, *331*
Модифицируемость программ *125*, *401*
Модульность *145*, *148–149*

Наследование *34*, *36–39*, *326*
Наследование интерфейса *404*
Наследование методов *329*, *338*
Наследование методов статическое *338*
Наследование методов статическое *338*, *342*
Наследование полей *335–336*
Наследование реализации *404*
Наследование свойств *335*
Невидимые компоненты *515*
Недействительный регион *242–243*

Область видимости класса *363*
Область видимости локальная *363*
Область видимости пространства имен *363*
Обобщенное имя функции *403*
Обобщенный клиентский код *404*
Обобщенный указатель *403*
Обработка завершения *486*
Обработка исключительных ситуаций *492*
Обработка события *450*, *464*
Обработка сообщения *450*, *454*, *464*
Обработчик события *453–455*, *463*, *471*, *479*
Объект *5*, *12*
Объектная модель *145*, *155–156*
Объектно-ориентированное программирование *130*, *209*
Объектно-ориентированное проектирование *128–130*
Объектно-ориентированный анализ *112*, *130*, *132*, *138*
Объявление класса *298*
Ограниченный тип типов *185*
Одиночное наследование *37–38*, *329*
Одноосновная алгебра *174*
Окно *18*, *73*, *205–206*, *209*, *213*, *222*

Окно сообщений *223*
Оконная процедура *213–215*, *217–221*, *224*, *226, 237*, *258*
Оконные компоненты *515*, *528*
Оконные координаты *246*
Оконные сообщения *217*
Оконный класс *217–218*
Операции приведения типов *426*, *428*
Операция using *367*, *371*
Определение класса *316–317*
Определение событий *527*
Определение требований *123*
Определяющее вхождение имени *363*
Оригинальные компоненты *515*
Отношение агрегации *28*, *43*, *72*
Отношение ассоциации *34*, *69*
Отношение зависимости *45*, *68*, *81*
Отношение композиции *72*, *74*
Отношение обобщения *74*
Отношение реализации *46*, *75*
Отношение связи *24*
Очередь сообщений *213*, *220–221*, *234*, *455*
Очередь сообщений приложения *213*, *221*, *452*
Ошибки трансляции *478*

Параллелизм *153–154*
Параметризованный класс (шаблон) *46*, *76*
Параметризованный тип *187*
Параметрический полиморфизм *408–409*
Параметры сообщения *217*
Параметры шаблонов *434*
Перегрузка функций *408*, *423*
Переходы *86*, *90*
Поведение объекта *18*, *21*, *148*, *297*, *359*
Подкласс *37*, *39*, *74*, *135, 140*, *150–151*
Подтип типов *185*
Поле *13*, *33*, *308*
Полиморфизм *40*, *188*, *401*, *403–404*, *407*
Полиморфизм Ad hoc *408*
Полиморфизм динамический *407*
Полиморфизм подтипов *412–413*
Полиморфизм статический *407*
Полиморфная функция *408–409*
Полиморфный объект *407*
Право владения *516*
Препроцессор *364*
Приведение типов *303*, *424*, *425*
Принцип ацикличности зависимостей *164*
Принцип Деметера *157*
Принцип единственности абстракции *156*
Принцип инверсии зависимости *156*
Принцип подстановки *160*
Принцип подстановки Лисковой *160*
Принцип разделения интерфейсов *162*
Приращение алгебры *175*
Приращение сигнатуры *172*
Проектирование *112*, *123*, *125*
Пространство имен *363*, *367–370*, *632*

Развитие программы *115*
Размер экземпляра класса *285*, *288–289, 293, 299*
Растровая графика *543*
Растровое изображение *240*
Расширение алгебры *175*
Расширение сигнатуры *172*
Реализация *31*
Реализация класса *31*, *407*
Реализация объекта *360*
Реализация сигнатуры *173*
Реверсная инженерия *403*
Регион *240*, *242*
Регистр фиксации событий *452*
Редукт алгебры *175*
Редукт сигнатуры *172*
Реинженерия *403*
Рефакторинг *403*
Родительское право *517*
Родовой тип *187*, *193*

Свойство *15*, *312*, *314*
Свойства класса *312*
Свойства компонентов *520*

Свойства объекта *15*, *48*
Свойства типа массив *198*, *525*
Свойства типа множество *525*
Связь компонента с владельцем *551*
Сервер *26*, *27*
Сигнатура *172*
Синхронизация *27*
Система координат устройства *248*
Система обработки ошибок *481*
Сцепление модулей *366*
Сложность программной системы *113*
Собственные события *529*
Событие *212*, *449*
События и сообщения Windows *530*
События компонентов *526*, *529*
Совместимость по присваиванию *301*
Создание исключительных ситуаций *499*
Сообщение *18*, *98–99*, *212*
Сообщения операционной системы *203*, *213*
Сопровождение программы *123–124*
Состояние действия *84–85*
Состояние объекта *14*, *19*, *32*
Сохранение программы *124*
Сохраняемость *154*
Спецификация *123*
Список параметров операции *65*
Стандартные компоненты *514–515*
Стандартные события *528*
Статическая модель *128–129*
Стереотипы сообщений *101*
Строка-свойство атрибута *61*
Строка-свойство операции *64*
Структура класса *285*, *288*
Структура объекта *285*, *289*
Структура сообщения *457*
Структурная обработка исключений *486*
Суперкласс *37*

Таблица виртуальных методов *288–289*
Таблица полей *288–289*
Таблица свойств *288–289*
Таблица методов класса *288–289*
Таблица статических методов *285, 288–289*
Текущая позиция пера *250*
Теория прототипов *137*
Тестирование *116*, *123*
Техника защищенного программирования *480*
Технология перемещения визуальных компонентов *459*, *551*
Тип атрибута *61*, *63*
Типизация *153*
Типовый параметр *405*

Уведомляющие сообщения *468*
Универсальный полиморфизм *408–409*
Управление фокусом *564*
Упреждающее объявление класса *298*

Фокус ввода *206*, *255*
Фокус управления *96–97*
Фрактальная графика *543*
Функция обратного вызова *215*

Цикл обработки событий *450, 464*
Цикл обработки сообщений *450, 454*

Шаблон *76–77*
Шаблон класса *437–438*
Шаблон функции *431–432*

Экземпляр исключительной ситуации *494*
Экземпляр класса *14*

Язык UML *53*, *284*

ЛАНЬ®

ОГЛАВЛЕНИЕ

Предисловие ... 3
ВВЕДЕНИЕ ... 5
Часть I. ПРИНЦИПЫ И ТЕХНОЛОГИЯ СОЗДАНИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ ... 11
1. Метафора объектно-ориентированного программирования ... 12
1.1. Объекты ... 12
1.1.1. Понятие объекта ... 12
1.1.2. Свойства и состояние объекта ... 14
1.1.3. Поведение объекта ... 18
1.1.4. Идентичность объекта ... 22
1.2. Отношения между объектами ... 24
1.2.1. Типы отношений ... 24
1.2.2. Отношение связи ... 24
1.2.3. Отношение агрегации ... 28
1.3. Классы ... 29
1.3.1. Понятие класса ... 29
1.3.2. Интерфейс и реализация ... 31
1.4. Отношения между классами ... 33
1.4.1. Типы отношений ... 33
1.4.2. Отношение ассоциации ... 34
1.4.3. Отношение наследования ... 35
1.4.4. Отношение агрегации ... 43
1.4.5. Отношение зависимости ... 45
1.4.6. Отношение реализации (инстанцирования) ... 46
1.4.7. Метаклассы ... 47
1.5. Взаимосвязь классов и объектов ... 47
2. МОДЕЛИРОВАНИЕ И ДОКУМЕНТИРОВАНИЕ СЛОЖНЫХ СИСТЕМ ... 53
2.1. Особенности изображения диаграмм языка UML ... 57
2.2. Диаграмма классов ... 58
2.2.1. Классы ... 59
2.2.2. Отношения между классами ... 68
2.2.3. Шаблоны или параметризованные классы ... 76
2.2.4. Повторное использование диаграмм классов ... 78
2.2.5. Пример диаграммы классов ... 78
2.3. Диаграмма деятельности ... 83
2.3.1. Состояние действия ... 84
2.3.2. Переходы ... 86
2.3.3. Дорожки ... 90
2.3.4. Объекты ... 92
2.3.5. Рекомендации по построению диаграмм деятельности ... 93
2.4. Диаграмма последовательности ... 95
2.4.1. Объекты ... 95
2.4.2. Сообщения ... 99
2.4.3. Пример построения диаграммы последовательности ... 103
2.4.4. Заключительные рекомендации по построению диаграмм последовательности ... 105

ЛАНЬ®

2.5. Рекомендации по изображению диаграмм языка UML 106
3. ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ РАЗРАБОТКА ПРОГРАММНЫХ СИСТЕМ 112
3.1. Среда и объект проектирования 113
3.1.1. Проблемы создания программных систем 113
3.1.2. Признаки сложной системы 117
3.1.3. Каноническая модель сложной системы 118
3.1.4. Человеческие возможности и сложные системы 120
3.1.5. Жизненный цикл программного изделия 123
3.2. Проектирование программных систем 125
3.2.1. Инженерное дело как наука, искусство и ремесло 125
3.2.2. Смысл проектирования 126
3.2.3. Важность построения модели 127
3.2.4. Элементы программного проектирования 127
3.2.5. Объектно-ориентированные анализ, проектирование и программирование 130
3.3. Принципы и инструменты объектно-ориентированного анализа 132
3.3.1. Классификация и объектно-ориентированный анализ 132
3.3.2. Трудности классификации 133
3.3.3. Итеративная суть классификации 135
3.3.4. Классическая категоризация 136
3.3.5. Концептуальная кластеризация 137
3.3.6. Теория прототипов 137
3.3.7. Применение классических и новых теорий 138
3.3.8. Объектно-ориентированный анализ 138
3.4. Принципы и инструменты объектно-ориентированного проектирования 145
3.4.1. Абстрагирование 146
3.4.2. Инкапсуляция 147
3.4.3. Модульность 148
3.4.4. Иерархия 150
3.4.5. Типизация 153
3.4.6. Параллелизм 153
3.4.7. Сохраняемость 154
3.4.8. Преимущества объектной модели 155
3.4.9. Принципы объектно-ориентированного проектирования 156
4. АЛГЕБРАИЧЕСКИЕ ОСНОВЫ ТИПОВ ДАННЫХ 169
4.1. Сигнатуры 171
4.2. Алгебры 172
4.3. Алгебры слов 175
4.4. Теории 176
4.5. Абстрактные типы данных 179
4.6. Иерархические модели типов данных 180
4.7. Типы и подтипы 183
4.8. Типы типов 184
4.9. Родовые типы данных 186
4.10. Полиморфные операции 188
4.11. Программы как данные 190
4.12. Типовая полнота языка программирования 191

ЛАНЬ®

4.13. Стандартные и произвольные типы 193
4.14. Примеры спецификации типов данных 193
5. ПРОГРАММИРОВАНИЕ В ОПЕРАЦИОННОЙ СИСТЕМЕ WINDOWS 203
5.1. Основные концепции Windows 204
5.1.1. Графический интерфейс пользователя 205
5.1.2. Многозадачность 206
5.1.3. Управление памятью 207
5.1.4. Независимость от аппаратных средств 208
5.1.5. Вызов функций 208
5.1.6. Разделение аппаратных ресурсов 209
5.1.7. Объектно-ориентированное программирование 209
5.1.8. Типы данных 210
5.2. Архитектура, управляемая событиями 212
5.2.1. Оконные процедуры и функции обратного вызова 213
5.2.2. Оконный класс 217
5.2.3. Сообщения 218
5.2.4. Цикл обработки сообщений 220
5.3. Примеры программ 221
5.3.1. Простейшая программа для Windows 221
5.3.2. Полноценная программа для Windows 224
5.4. Программирование графики 240
5.4.1. Основные средства 240
5.4.2. Контекст устройства 240
5.4.3. Типы контекстов устройства 242
5.4.4. Регионы Windows 242
5.4.5. Получение контекста устройства 243
5.4.6. Использование сообщения WM_PAINT 245
5.4.7. Использование функций InvalidateRect и InvalidateRgn 245
5.4.8. Системы координат и преобразования 246
5.5. Рисование линий, фигур, текста 249
5.5.1. Позиция пера 250
5.5.2. Рисование прямой линии 250
5.5.3. Рисование ломаной линии 251
5.5.4. Рисование дуги 251
5.5.5. Рисование прямоугольника 253
5.5.6. Рисование эллипса 253
5.5.7. Рисование многоугольника 253
5.6. Средства ввода 254
5.6.1. Клавиатура 254
5.6.2. Фокус ввода 255
5.6.3. Клавиши и символы 256
5.6.4. Аппаратные сообщения 256
5.6.5. Символьные сообщения 258
5.6.6. Мышь 260
5.6.7. Сообщения мыши 261
5.6.8. Обработка двойного щелчка 262
5.6.9. Обработка сообщений от колеса мыши 262
5.6.10. Пример программы «Просмотрщик текста» 262

Часть II. РЕАЛИЗАЦИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ ТЕХНОЛОГИИ В СИСТЕМЕ ПРОГРАММИРОВАНИЯ 281
6. УСТРОЙСТВО И ФУНКЦИОНИРОВАНИЕ ПРОГРАММЫ 283
6.1. Структура класса и структура объекта 285
6.2. Класс TObject 290
6.3. Жизнь классов и объектов 295
6.4. Доступ к структуре класса и структуре объекта 299
6.5. Конструкторы и деструкторы 306
6.6. Поля 308
6.7. Методы 310
6.8. Свойства 312
6.9. Программирование классов и объектов 316
6.9.1. Определение класса 316
6.9.2. Использование класса 317
7. НАСЛЕДОВАНИЕ 326
7.1. Назначение и способы наследования 326
7.1.1. Метафора наследования 326
7.1.2. Одиночное наследование классов 329
7.1.3. Комбинация модификаторов наследования и модификаторов доступа 331
7.1.4. Использование конструкторов и деструкторов при наследовании 333
7.2. Наследование полей и свойств 335
7.3. Наследование методов 338
7.3.1. Статическое и динамическое наследование методов 338
7.3.2. Статическое наследование методов 339
7.3.3. Динамическое наследование методов 346
7.4. Использование наследования в разработке программ 349
8. ИНКАПСУЛЯЦИЯ 356
8.1. Назначение и свойства инкапсуляции 356
8.2. Средства инкапсуляции 358
8.3. Использование программных модулей 362
8.3.1. От исходного кода к исполняемому модулю 363
8.3.2. Разделение текста программы на модули 365
8.3.3. Интерфейс и реализация 375
8.4. Использование полей и свойств с разными областями видимости 382
8.5. Обеспечение инкапсуляции в проектах 383
8.6. Абстрагирование и инкапсуляция 393
9. ПОЛИМОРФИЗМ 401
9.1. Назначение и виды полиморфизма 401
9.1.1. Метафора полиморфизма 401
9.1.2. Виды полиморфизма 407
9.2. Универсальный полиморфизм 409
9.2.1. Параметрический полиморфизм 409
9.2.2. Полиморфизм подтипов 412
9.3. Специальный полиморфизм 423
9.3.1. Перегрузка функций 423
9.3.2. Приведение типов 424
9.4. Шаблоны функций и классов 431
9.4.1. Шаблоны функций 431

9.4.2. Шаблоны классов ... 437
9.4.3. Пример использования шаблона ... 440
10. ОБРАБОТКА СОБЫТИЙ И СООБЩЕНИЙ ... 449
10.1. Метафора обработки событий и сообщений ... 449
10.2. Обработка сообщения в программе ... 454
10.3. Отправка сообщения из программы ... 455
10.4. Программирование сообщений и событий ... 457
10.4.1. Структура сообщения ... 457
10.4.2. Включение обработки сообщений в программу ... 464
10.4.3. Уведомляющие сообщения ... 468
10.4.4. Включение событий в программу ... 470
11. ОБРАБОТКА ОШИБОК ... 478
11.1. Метафора обработки ошибок ... 478
11.2. Техника защищенного программирования с использованием собственных средств C++ ... 482
11.2.1. Создание и перехват исключений ... 483
11.2.2. Обработка различных исключений ... 484
11.2.3. Повторная обработка исключений ... 486
11.3. Техника защищенного программирования с использованием структурной обработки исключений ... 486
11.3.1. Использование обработчиков завершения ... 487
11.3.2. Использование обработчиков исключения ... 490
11.3.3. Исключительная ситуация как класс ... 491
11.3.4. Использование экземпляров исключительной ситуации ... 494
11.3.5. Использование многоуровневых защищенных блоков ... 495
11.3.6. Создание новых исключительных ситуаций ... 499
12. БИБЛИОТЕКА ВИЗУАЛЬНЫХ КОМПОНЕНТОВ ... 511
12.1. Метафора библиотеки визуальных компонентов ... 511
12.1.1. Назначение библиотеки визуальных компонентов ... 511
12.1.2. Типы компонентов ... 514
12.1.3. Свойства компонент ... 520
12.1.4. События компонентов ... 526
12.1.5. Методы компонентов ... 531
12.2. Базовые классы ... 533
12.3. Класс TObject ... 536
12.4. Класс TPersistent ... 536
12.5. Класс TComponent ... 538
12.6. Класс TCanvas ... 542
12.7. Класс TControl ... 546
12.7.1. Свойства и методы ... 546
12.7.2. События ... 552
12.7.3. Технология перемещения визуальных компонентов ... 556
12.8. Класс TWinControl ... 561
12.8.1. Свойства и методы ... 561
12.8.2. События ... 566
12.9. Класс TCustomControl ... 570
12.10. Класс TGraphicControl ... 571
12.11. Класс TApplication ... 572

12.11.1. Свойства и методы 572
12.11.2. Класс TApplicationEvents 579
12.12. Класс TScreen 581
12.13. Класс TClipboard 585
12.13.1. Общее описание 585
12.13.2. Обработка текстовой информации 589
12.13.3. Обработка графической информации 590
12.13.4. Копирование компонента в буфер обмена 590
12.13.5. Копирование произвольных данных в буфер обмена 591
12.14. Класс TIniFile 593
12.15. Класс TForm 597
12.15.1. Класс TForm в иерархии наследования 598
12.15.2. Свойства и методы класса TForm 600
12.15.3. События класса TForm 606
ПРИЛОЖЕНИЕ 1. ИСТОРИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ 637
П1.1. Историческая справка 1 637
П1.2. Историческая справка 2 641
ПРИЛОЖЕНИЕ 2. ПРАВИЛА ИМЕНОВАНИЯ ИДЕНТИФИКАТОРОВ 648
П2.1. Общие рекомендации по именованию идентификаторов 649
П2.2. Некоторые договоренности об именовании идентификаторов 654
П2.2.1. Соглашения об именах MFS 654
П2.2.2. Венгерская нотация 654
П2.3. Наши соглашения об именовании идентификаторов 655
П2.3.1. Именование переменных, констант и свойств 655
П2.3.2. Именование типов данных 657
П2.3.3. Именование функций 657
ПРИЛОЖЕНИЕ 3. ПРОГРАММА ТАБЛИЧНОГО ХРАНЕНИЯ СЛОВ 658
П3.1. Общее описание 658
П3.2. Описание алгоритмов 658
П3.3. Структуры данных 659
П3.4. Диаграмма классов 660
П3.5. Диаграмма последовательности 664
П3.6. Исходный текст программы 667
П3.7. Использование программы 685
ЛИТЕРАТУРА 688
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ 689

ЛАНЬ®

Игорь Александрович БАРКОВ

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебник

Зав. редакцией
литературы по информационным технологиям
и системам связи *О. Е. Гайнутдинова*
Ответственный редактор *Т. С. Спирина*
Подготовка макета *И. Г. Попов*
Корректор *А. В. Финкельштейн*
Выпускающий *С. Ю. Гаганов*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com
196105, Санкт-Петербург, пр. Юрия Гагарина, д. 1, лит. А
Тел./факс: (812) 336-25-09, 412-92-72
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 15.05.19.
Бумага офсетная. Гарнитура Школьная. Формат 70×100 $^{1}/_{16}$.
Печать офсетная. Усл. п. л. 56,88. Тираж 100 экз.

Заказ № 322-19.

Отпечатано в полном соответствии с качеством
предоставленного оригинал-макета в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.