
М. М. МАРАН



ПРОГРАММНАЯ ИНЖЕНЕРИЯ

Учебное пособие

Издание второе, стереотипное



САНКТ-ПЕТЕРБУРГ
МОСКВА
КРАСНОДАР
2021

УДК 004. 4 (075)
ББК 32.973-018.2я73

М 25 Маран / М. М. Программная инженерия : учебное пособие для вузов / М. М. Маран. — 2-е изд., стер. — Санкт-Петербург : Лань, 2021. — 196 с. : ил. — Текст : непосредственный.

ISBN 978-5-8114-8367-9

Учебное пособие предназначено для студентов, впервые приступающих к изучению методов разработки программного обеспечения, но имеющих базовую подготовку по программированию и алгоритмизации. В нем рассмотрены этапы жизненного цикла программного обеспечения. Дан краткий обзор наиболее известных методик разработки программного обеспечения. Наибольшее внимание уделено объектно-ориентированному подходу, языку UML и унифицированному процессу. Рассмотрены основные диаграммы UML и их применение при выполнении этапов анализа и проектирования. Изложены возможности среды Microsoft Visual Studio для работы с UML. Имеется раздел, посвященный некоторым вопросам реализации на языке C#. Подробно рассмотрены рефакторинг и тестирование, а также обеспечивающие средства в названной среде.

Для студентов вузов всех направлений обучения, в учебных планах которых имеются дисциплины «Программная инженерия» или «Технология программирования».

УДК 004. 4 (075)
ББК 32.973-018.2я73



Обложка
Е. А. ВЛАСОВА

© Издательство «Лань», 2021
© М. М. Маран, 2021
© Издательство «Лань»,
художественное оформление, 2021

Оглавление

Введение	6
1. Жизненный цикл программного обеспечения	7
1.1. Основные понятия	7
1.2. Процесс анализа требований к программным средствам	9
1.3. Процесс проектирования архитектуры программных средств	10
1.4. Процесс детального проектирования программных средств	11
1.5. Процесс конструирования программных средств	12
1.6. Процесс комплексирования программных средств	13
1.7. Процесс квалификационного тестирования программных средств	14
1.8. Поставка и внедрение	15
1.9. Сопровождение программного продукта	16
1.10. Модели жизненного цикла	17
2. Методологии разработки программного обеспечения	21
2.1. Классические методологии	21
2.1.1. Метод функциональной декомпозиции	21
2.1.2. Метод анализа потоков данных	23
2.2. Объектно-ориентированный подход	24
2.3. Agile-методики	24
3. Языки UML и OCL	27
3.1. Общие понятия	27
3.2. Диаграмма вариантов использования	28
3.3. Диаграмма классов	33
3.4. Диаграмма состояний	37
3.5. Диаграмма деятельности	40
3.6. Диаграмма последовательностей	42
3.7. Диаграмма кооперации	46
3.8. Диаграмма компонентов	46
3.9. Диаграмма развертывания	47
3.10. Язык OCL	48
4. Выполнение этапов анализа и проектирования на языке UML	50
4.1. Основные положения	50

4.2. Определение требований.....	53
4.3. Уточнение и структурирование требований	57
4.4. Проектирование архитектуры.....	60
4.5. Проектирование классов и подсистем	63
4.6. Проектирование вариантов использования.....	65
4.7. Использование среды <i>Microsoft Visual Studio</i> для выполнения анализа ...	67
4.8. Создание диаграмм проектирования.....	69
5. Реализация на языке C# в среде <i>Microsoft Visual Studio</i>	76
5.1. Простейший пример.....	76
5.1.1. Постановка задачи	76
5.1.2. Средства управления работой программы	77
5.1.3. Создание меню	78
5.1.4. Ввод/вывод массивов.....	79
5.1.5. Форматированный ввод/вывод двумерного массива	83
5.2. Создание многооконных приложений	87
5.2.1. Создание SDI-приложения.....	88
5.2.2. Создание MDI-приложения	91
5.3. Реализация алгоритмов.....	96
5.4. Работа с библиотекой классов	98
5.5. Реализация взаимодействия с базой данных.....	101
5.5.1. Структура базы данных.....	101
5.5.2. Структура проекта	104
5.5.3. Просмотр данных без изменений	105
5.5.4. Просмотр и изменение данных.....	107
5.5.5. Извлечение данных.....	108
5.5.6. Изменение данных хранимыми процедурами	110
5.5.7. Использование данных из базы для вычислений	113
5.6. Использование стандартных классов в реализации	115
5.6.1. Работа с классом List	116
5.6.2. Работа со словарем — классом Dictionary	123
5.6.3. Список из списков.....	127
5.6.4. Использование диалоговых окон для работы со стандартными классами	129
5.7. Дополнительные средства создания интерфейса пользователя.....	136

5.8. Вывод на печать	142
5.9. Рефакторинг	145
5.10. Работа в <i>WPF</i>	150
5.10.1. Простейший пример	151
5.10.2. Работа с двумерным форматизированным массивом	154
5.10.3. Работа со списком	156
6. Тестирование программного обеспечения	160
6.1. Методы проверки программного обеспечения	160
6.2. Тестирование примитивных программ	162
6.2.1. Тестирование «черного ящика»	162
6.2.2. Тестирование «белого ящика»	168
6.2.3. Метод функциональных диаграмм	171
6.2.4. Предположение об ошибке	174
6.3. Тестирование программных комплексов, построенных методом функциональной декомпозиции	174
6.4. Тестирование программных комплексов, построенных по объектно-ориентированной методике	177
6.5. Отладка программ	180
6.6. Средства тестирования <i>Microsoft Visual Studio</i>	181
6.7. Современный подход к проверке	190
Заключение	193
Библиографический список	194



Введение

Программная инженерия — это комплексный подход построения программного обеспечения для самых разнообразных задач. Она охватывает весь процесс от возникновения необходимости в новой программе до завершения ее эксплуатации. Применение методов программной инженерии способствует составлению программ высокого качества и с приемлемыми затратами.

Целью данного учебного пособия является дать студентам первое представление об этой науке. В нем коротко рассмотрены все этапы жизненного цикла программного обеспечения, приведены практические рекомендации по их выполнению, а также описание инструментальных средств для этого. Автор надеется, что после данного пособия читатель может самостоятельно освоить более подробные книги по этой тематике.

В первой главе даны определения основных понятий, описание жизненного цикла программного продукта и его основных этапов, рассмотрены модели жизненного цикла и области их применения.

Вторая глава содержит краткое описание методик создания программного обеспечения и условий их применения.

Третья глава посвящена языкам *UML* и *OCL*, приведены описания диаграмм, их назначение и рекомендации по составлению. Язык *OCL* рассмотрен как средство дополнения некоторых моделей на *UML*.

В четвертой главе описана методика выполнения этапов жизненного цикла анализа и проектирования средствами *UML* и *OCL*. В качестве CASE-средства составления диаграмм используется *Microsoft Visual Studio*. Подробно проанализированы отличия моделей анализа и проектирования и их реализации в упомянутой среде.

Пятая глава содержит рекомендации по практической работе в среде *Microsoft Visual Studio* на языке *C#*. Автор попытался дать минимальные сведения для разработки законченных приложений, состоящие как из интерфейса пользователя, так и логики решения задач, не вдаваясь при этом в алгоритмические сложности и в тонкости применения визуальных компонентов. В этой же главе рассмотрены вопросы рефакторинга и средства рефакторинга в применяемой среде. В основном рассмотрена технология *WindowsForms*, коротко изложены и принципы *WPF*.

Шестая глава посвящена рассмотрению тестирования программ и средствам тестирования в среде. Отмечено, что проверка результатов должна сопровождать весь жизненный цикл программного продукта, начиная с анализа, ведь известно, что стоимость устранения допущенных на начальных стадиях жизненного цикла ошибок обходится значительно дороже устранения ошибок реализации.

Для чтения пособия необходима базовая подготовка по основам алгоритмизации и программирования, а также по базам данных. Примеры приведены на языке *C#* в среде *Microsoft Visual Studio*, поэтому легче всего будет читателю, знакомому именно с этим языком. В некоторых примерах используется язык *LINQ*.

1. Жизненный цикл программного обеспечения

1.1. Основные понятия

Стандарт [1] устанавливает строгую связь между системой и применяемыми в ней программными средствами. Такая связь основывается на общих принципах системной инженерии. Программное средство трактуется как единая часть общей системы, выполняющая определенные функции в данной системе, что осуществляется посредством выделения требований к программным средствам из требований к системе, проектирования, производства программных средств и объединения их в систему. Этот принцип является фундаментальной предпосылкой для упомянутого стандарта, в котором программные средства всегда существуют в контексте системы, даже если система состоит из единственного процессора, выполняющего программы. В таком случае программный продукт или услуга всегда рассматриваются как одна из составных частей системы. Существует различие между анализом системных требований и анализом требований к программным средствам, так как в общем случае построение системной архитектуры определяет системные требования для различных составных частей системы, а анализ требований к программным средствам предопределяет требования к ним, исходя из системных требований, назначенных каждой программной составной части. Конечно, в некоторых случаях непрограммных элементов в системе может быть настолько мало, что можно не делать различия между анализом системы и анализом программных средств. Дальнейшее изложение в настоящем пособии касается только программных средств.

Программная инженерия определяется как системный подход к анализу, проектированию, оценке, реализации, тестированию, обслуживанию и модернизации программного обеспечения. Программная инженерия занимается разработкой систематических моделей и надежных методов производства высококачественного программного обеспечения, и данный подход распространяется на все уровни — от теории и принципов до реальной практики создания программного обеспечения.

Технология программирования — изучение процессов программной инженерии, методов и инструментальных средств.

В литературе и нормативно-технических документах встречаются следующие определения **Программного обеспечения (ПО)**:

- Все или часть программ, процедур, правил и соответствующей документации системы обработки информации.
- Компьютерные программы, процедуры и, возможно, соответствующая документация и данные, относящиеся к функционированию компьютерной системы.
- Совокупность программ системы обработки информации и программных документов, необходимых для эксплуатации этих программ.

Используется и понятие программный продукт: совокупность компьютерных программ, процедур и, возможно, связанных с ними документации и данных.

Согласно [1], **жизненный цикл** — это развитие системы, продукта, услуги, проекта или других изготовленных человеком объектов, начиная со стадии разработки концепции и заканчивая прекращением применения. Другими словами можно сказать, что жизненный цикл программного обеспечения охватывает промежуток времени от возникновения необходимости в нем до завершения его эксплуатации. При возникновении такой необходимости возникает и вечный вопрос: приобрести уже готовый продукт или разработать собственный. Конечно, изобретать велосипед ни к чему, но на практике встречается много случаев, когда по многим причинам выгоднее разработать самому. Данное учебное пособие предназначено будущим разработчикам, поэтому в дальнейшем рассмотрим только эту проблематику, оставляя вопросы выбора, приобретения, внедрения и эксплуатации готовых продуктов за кадром. Другими словами, мы ставим перед собой цель реализации программного продукта.

В ходе процесса разработки происходит преобразование заданных поведенческих, интерфейсных и производственных ограничений в действия, которые создают системный элемент, выполненный в виде программного продукта. Результатом процесса является создание продукта, удовлетворяющего всем требованиям. В ходе процесса реализации программных средств:

- определяется стратегия реализации;
- определяются ограничения по технологии реализации проекта;
- изготавливается программная составная часть.

В дополнение к этим действиям процесс реализации программных средств имеет следующие процессы более низкого уровня:

- процесс анализа требований к программным средствам;
- процесс проектирования архитектуры программных средств;
- процесс детального проектирования программных средств;
- процесс конструирования программных средств;
- процесс комплексирования программных средств;
- процесс квалификационного тестирования программных средств.

В ходе реализации разработчик должен определить модель жизненного цикла, соответствующую области применения, размерам и сложности проекта. Модель жизненного цикла должна содержать стадии, цели и выходы каждой стадии. Виды деятельности и задачи процесса реализации программных средств должны быть выбраны и отражены в модели жизненного цикла. Эти виды деятельности и задачи могут пересекаться или взаимодействовать друг с другом, могут выполняться итеративно или рекурсивно.

После завершения разработки предстоят этапы жизненного цикла — внедрение и сопровождение.

1.2. Процесс анализа требований к программным средствам

В ходе процесса анализа требований к программным средствам:

- определяются требования к программным элементам системы и их интерфейсам;
- требования к программным средствам анализируются на корректность и тестируемость;
- осознается воздействие требований к программным средствам на среду функционирования;
- определяются приоритеты реализации требований к программным средствам;
- требования к программным средствам принимаются и обновляются по мере необходимости;
- оцениваются изменения в требованиях к программным средствам по стоимости, графикам работ и техническим воздействиям.

В результате выполненной работы будут получены:

- спецификации функциональных характеристик и возможностей, включая эксплуатационные, физические характеристики и условия окружающей среды, при которых будет применяться программная составная часть;
- внешние интерфейсы к программной составной части;
- квалификационные требования;
- спецификации по безопасности, включая те спецификации, которые относятся к методам функционирования и сопровождения, влияния окружающей среды и ущерба для персонала;
- спецификации по защите, включая спецификации, связанные с угрозами для чувствительной информации;
- спецификации эргономических факторов, включая спецификации, связанные с ручными операциями, взаимодействием человека с оборудованием, ограничениями по персоналу и областям, требующим концентрации внимания и чувствительным к ошибкам человека и уровню его обученности;
- описание данных и требования к базам данных;
- инсталляция и требования к приемке поставляемого программного продукта в местах функционирования и сопровождения;
- требования к документации пользователя;
- операции пользователя и требования к их выполнению;
- пользовательские требования к сопровождению.

1.3. Процесс проектирования архитектуры программных средств

В результате реализации процесса проектирования архитектуры программных средств:

- разрабатывается проект архитектуры программных средств и устанавливается базовая линия, описывающая программные составные части, которые будут реализовывать требования к программным средствам;
- определяются внутренние и внешние интерфейсы каждой программной составной части.

Для достижения этих целей:

1. Исполнитель должен преобразовать требования к программным составным частям в архитектуру, которая описывает верхний уровень его структуры и идентифицирует программные компоненты. Необходимо гарантировать, что все требования к программным составным частям распределяются по программным компонентам и в дальнейшем уточняются для облегчения детального проектирования. Архитектуру программной составной части необходимо документировать.

Примечание. Проектирование архитектуры программных средств обеспечивает также основу для верификации программных составных частей, объединения программных составных частей друг с другом и их интеграции с остальными составными частями системы.

2. Исполнитель должен разработать и документально оформить проект верхнего уровня для внешних интерфейсов программной составной части и интерфейсов между ней и программными компонентами.
3. Исполнитель должен разработать и документально оформить проект верхнего уровня для базы данных.
4. Исполнитель должен разработать и документально оформить предварительные версии пользовательской документации.
5. Исполнитель должен определить и документировать требования к предварительному тестированию и график работ по комплексированию программных средств.
6. Исполнитель должен оценить архитектуру программной составной части, проекты по интерфейсам и базе данных, учитывая следующие критерии:
 - прослеживаемость к требованиям программной составной части;
 - внешняя согласованность с требованиями программной составной части;
 - внутренняя согласованность между программными компонентами;
 - приспособленность методов проектирования и используемых стандартов;
 - осуществимость детального проектирования;
 - осуществимость функционирования и сопровождения.

1.4. Процесс детального проектирования программных средств

Цель процесса детального проектирования программных средств заключается в обеспечении проекта для программных средств, которые реализуются и могут быть верифицированы относительно установленных требований и архитектуры программных средств, а также существенным образом детализируются для последующего кодирования и тестирования.

В результате осуществления процесса детального проектирования программных средств:

- разрабатывается детальный проект каждого программного компонента, описывающий создаваемые программные модули;
- определяются внешние интерфейсы каждого программного модуля и устанавливается совместимость и прослеживаемость между детальным проектированием, требованиями и проектированием архитектуры.

Для каждой программной составной части детальное проектирование состоит из решения следующих задач:

1. Исполнитель должен разработать детальный проект для каждого программного компонента. Программные компоненты должны быть детализированы на более низком уровне, включающем программные блоки, которые могут быть закодированы, откомпилированы и проверены. Следует гарантировать, что все требования к программным средствам распределяются от программных компонентов к программным блокам. Детальный проект должен быть документально оформлен.
2. Исполнитель должен разработать и документально оформить детальный проект для внешних интерфейсов к программным составным частям, между программными компонентами и между программными блоками. Необходимо, чтобы детальный проект для интерфейсов позволял проводить кодирование без потребности в получении дополнительной информации.
3. Исполнитель должен разработать и документально оформить детальный проект базы данных.
4. Исполнитель должен совершенствовать пользовательскую документацию по мере необходимости.
5. Исполнитель должен определять и документировать требования к тестированию и графики работ по тестированию программных блоков. Необходимо, чтобы требования к тестированию включали в себя проведение проверок программных блоков при граничных значениях параметров, установленных в требованиях.
6. Исполнитель должен обновлять требования к тестированию и графики работ по комплектованию программных средств.
7. Исполнитель должен оценивать детальный проект для программных средств и требования к тестированию по следующим критериям:

- прослеживаемость к требованиям программной составной части;
- внешняя согласованность с архитектурным проектом;
- внутренняя согласованность между программными компонентами и программными блоками;
- соответствие методов проектирования и используемых стандартов;
- осуществимость тестирования;
- осуществимость функционирования и сопровождения.

1.5. Процесс конструирования программных средств

Цель процесса конструирования программных средств заключается в создании исполняемых программных блоков.

В результате осуществления процесса конструирования программных средств:

- определяются критерии верификации для всех программных блоков относительно требований;
- изготавливаются программные блоки, определенные проектом;
- устанавливается совместимость и прослеживаемость между программными блоками, требованиями и проектом;
- завершается верификация программных блоков относительно требований и проекта.

Данный вид деятельности состоит из решения следующих задач:

1. Исполнитель должен разработать и документально оформить:
 - каждый программный блок и базу данных;
 - процедуры тестирования и данные для тестирования каждого программного блока и базы данных.
2. Исполнитель должен тестировать каждый программный блок и базу данных, гарантируя, что они удовлетворяют требованиям. Результаты тестирования должны быть документально оформлены.
3. Исполнитель должен улучшать документацию пользователя при необходимости.
4. Исполнитель должен совершенствовать требования к тестированию и графики работ по комплексированию программных средств.
5. Исполнитель должен оценивать программный код и результаты испытаний, учитывая следующие критерии:
 - прослеживаемость к требованиям и проекту программных элементов;
 - внешнюю согласованность с требованиями и проектом для программных составных частей;
 - внутреннюю согласованность между требованиями к блокам;
 - тестовое покрытие блоков;

- соответствие методов кодирования и используемых стандартов; осуществимость комплексирования и тестирования программных средств;
- осуществимость функционирования и сопровождения. Результаты оценки должны быть документально оформлены.

1.6. Процесс комплексирования программных средств

Цель процесса комплексирования программных средств заключается в объединении программных блоков и программных компонентов, создании интегрированных программных элементов, согласованных с проектом программных средств, которые демонстрируют, что функциональные и нефункциональные требования к программным средствам удовлетворяются на полностью укомплектованной или эквивалентной ей операционной платформе.

В результате осуществления процесса комплексирования программных средств:

- разрабатывается стратегия комплексирования для программных блоков, согласованная с программным проектом и расположенными по приоритетам требованиями к программным средствам;
- разрабатываются критерии верификации для программных составных частей, которые гарантируют соответствие с требованиями к программным средствам, связанными с этими составными частями;
- программные составные части верифицируются с использованием определенных критериев;
- изготавливаются программные составные части, определенные стратегией комплексирования;
- регистрируются результаты комплексного тестирования;
- устанавливаются согласованность и прослеживаемость между программным проектом и программными составными частями;
- разрабатывается и применяется стратегия регрессии для повторной верификации программных составных частей при возникновении изменений в программных блоках (в том числе в соответствующих требованиях, проекте и кодах).

Для каждой программной составной части (или составной части конфигурации, если она определена) данный вид деятельности состоит из решения следующих задач:

1. Исполнитель должен разработать план комплексирования для объединения программных блоков и программных компонентов в программную составную часть. План должен включать в себя требования к тестированию, процедуры, данные, обязанности и графики работ. План должен быть оформлен документально.

2. Исполнитель должен объединить программные блоки, программные компоненты и тесты, поскольку они разрабатываются в соответствии с планом комплексирования. Должны быть гарантии в том, что каждое такое объединение удовлетворяет требованиям к программной составной части и что составная часть комплексировается при завершении выполнения данной задачи. Результаты комплексирования и тестирования должны быть оформлены документально.

Примечание. Должна быть разработана стратегия регрессии для применения повторной верификации программных элементов, в случае когда изменения проводятся в программных блоках, включая соответствующие требования, проект и коды.

3. Исполнитель должен обновлять пользовательскую документацию по мере необходимости.
4. Исполнитель должен разработать и документально оформить для каждого квалификационного требования к программной составной части комплект тестов, тестовых примеров (входов, результатов, критериев тестирования) и процедур тестирования для проведения квалификационного тестирования программных средств. Разработчик должен гарантировать, что после комплексирования программная составная часть будет готова к квалификационному тестированию.
5. Исполнитель должен оценить план комплексирования, проект, код, тесты, результаты тестирования и пользовательскую документацию, учитывая:
 - прослеживаемость к системным требованиям;
 - внешнюю согласованность с системными требованиями;
 - внутреннюю согласованность;
 - тестовое покрытие требований к программной составной части;
 - приспособленность используемых методов и стандартов тестирования;
 - соответствие ожидаемым результатам;
 - осуществимость квалификационного тестирования программных средств;
 - осуществимость функционирования и сопровождения.

1.7. Процесс квалификационного тестирования программных средств

Цель процесса квалификационного тестирования программных средств заключается в подтверждении того, что комплектованный программный продукт удовлетворяет установленным требованиям.

В результате квалификационного тестирования программных средств:

- определяются критерии для комплектованных программных средств с целью демонстрации соответствия с требованиями к программным средствам;
- комплектованные программные средства верифицируются с использованием определенных критериев;

- записываются результаты тестирования;
- разрабатывается и применяется стратегия регрессии для повторного тестирования комплектованного программного средства при проведении изменений в программных составных частях.

Примечание. Должна быть разработана стратегия регрессии для повторного применения тестирования комплексированного программного средства при проведении изменений в программных составных частях.

Для каждой программной составной части (или составной части конфигурации, если она определена) данный вид деятельности состоит из решения следующих задач:

1. Исполнитель должен проводить квалификационное тестирование в соответствии с квалификационными требованиями к программному элементу. Должна обеспечиваться гарантия того, что реализация каждого требования к программным средствам тестируется на соответствие. Результаты квалификационного тестирования должны быть документально оформлены.
2. Исполнитель должен обновлять пользовательскую документацию по мере необходимости.
3. Исполнитель должен оценивать проект, код, тесты, результаты тестирования и пользовательскую документацию, учитывая следующие критерии:
 - тестовое покрытие требований к программной составной части;
 - соответствие с ожидаемыми результатами;
 - осуществимость системного комплексирования и тестирования, если они проводятся (осуществимость функционирования и сопровождения). Результаты оценки должны быть документально оформлены.
4. Исполнитель должен поддерживать проведение аудитов. Результаты аудитов должны быть документально оформлены. Если и технические, и программные средства разрабатываются или комплексировются, то аудиты могут быть отсрочены до тех пор, пока не будет выполнено системное квалификационное тестирование.
5. После успешного завершения аудитов (если они проводились) исполнитель должен обновить и подготовить поставляемый программный продукт для системного комплексирования, системного квалификационного тестирования, инсталляции программных средств или поддержки приемы программных средств.



1.8. Поставка и внедрение

Поставка — это передача разработанного программного продукта от разработчика заказчику для внедрения. Внедрение — это по существу процесс отчуждения разработки от автора. По завершении внедрения программный продукт должен работать у заказчика без постоянного участия разработчика. Внедрение состоит из двух этапов:

- опытная эксплуатация;
- промышленная эксплуатация.

В ходе опытной эксплуатации разработчик и заказчик работают вместе. Разработчик должен помогать освоить работу с новым программным продуктом, а заказчик освоить работу. Разработчик должен устранить выявленные в ходе опытной эксплуатации ошибки и недочёты (никому еще не удалось создать программный продукт без единой ошибки!), еще раз проверить работоспособность своей разработки в реальных условиях.

Опытная эксплуатация постепенно переходит в промышленную эксплуатацию, когда программный продукт работает в реальных условиях эксплуатации (объем базы данных, количество одновременно работающих пользователей, быстродействие в наихудших условиях и т. д.) и обеспечивает требуемые характеристики.

Во время промышленной эксплуатации разработчик выполняет сопровождение программного продукта.

1.9. Сопровождение программного продукта

Сопровождение — комплекс действий, требуемых для предоставления экономически эффективной поддержки для программных систем. Сопровождаемость — возможность изменения программного продукта. Изменения могут включать коррекции, улучшения или адаптацию программного средства к изменениям в окружающей среде, к требованиям и функциональному спецификациям. Содержание процесса сопровождения регламентируется в [2].

Различают следующие виды сопровождения:

1. Адаптивное сопровождение. Изменения (модификация) программного продукта после внедрения, обеспечивающие его работоспособность в измененных или изменяющихся условиях.
2. Корректирующее сопровождение. Реактивное изменение программного продукта, выполняемое после его поставки для корректировки обнаруженных проблем.
3. Экстренное сопровождение. Незапланированная модификация, выполняемая для временного сохранения работоспособности программного средства, ожидающего корректирующее сопровождение.
4. Сопровождаемая модификация. Изменения к существующему программному продукту для удовлетворения новых требований.
5. Предложение к модификации. Сбор предполагаемых изменений в сопровождаемом программном продукте.
6. Полное сопровождение. Модификация программного продукта после поставки для обнаружения и исправления скрытых недостатков, пока они не проявились в виде сбоев.
7. Профилактическое сопровождение. Модификация программного продукта после поставки в целях обнаружения и корректировки имеющихся в нем скрытых ошибок для предотвращения их явного проявления в процессе эксплуатации.

1.10. Модели жизненного цикла

Модель жизненного цикла — это структура процессов и действий, связанных с жизненным циклом, организуемых в стадии, которые также служат в качестве общей ссылки для установления связей и взаимопонимания сторон. Стадия — это период в пределах жизненного цикла некоторого объекта, который относится к состоянию его описания или реализации.

Данное учебное пособие посвящено введению в программную инженерию. Поэтому рассмотрим укрупненную модель жизненного цикла (разработка небольших программных средств), состоящую из следующих этапов:

- анализ требований;
- проектирование структуры;
- программирование, тестирование, отладка;
- сборка, валидация, верификация;
- внедрение;
- сопровождение.

Примечание. Согласно [1]: **валидация** (*validation*): подтверждение (на основе представления объективных свидетельств) того, что требования, предназначенные для конкретного использования или применения, выполнены. Валидация в контексте жизненного цикла представляет собой совокупность действий, гарантирующих и обеспечивающих уверенность в том, что система способна реализовать свое предназначение, текущие и перспективные цели.

Верификация (*verification*): подтверждение (на основе представления объективных свидетельств) того, что заданные требования полностью выполнены. Верификация в контексте жизненного цикла представляет собой совокупность действий по сравнению полученного результата жизненного цикла с требуемыми характеристиками для этого результата.

В дальнейшем будут рассмотрены стадии анализа, проектирования, программирования, тестирования и отладки. Содержание названных стадий будет разьяснено по ходу изложения.

Рассмотрим каскадную модель жизненного цикла и модель «спираль». Каскадная модель (под влиянием английского языка именуемая иногда «водопад» — Waterfall) представлена на рис. 1.1.

Характерная черта этой модели — полное завершение предыдущего этапа до начала следующего, другими словами, полное отсутствие обратных связей.

В связи с этим имеются следующие ограничения:

1. Эта модель предполагает, что требования к разрабатываемой системе заморожены до начала проектирования.
2. Замораживание требований к системе обычно влечет за собой и выбор технических средств в начале разработки (они являются частью требований к системе).

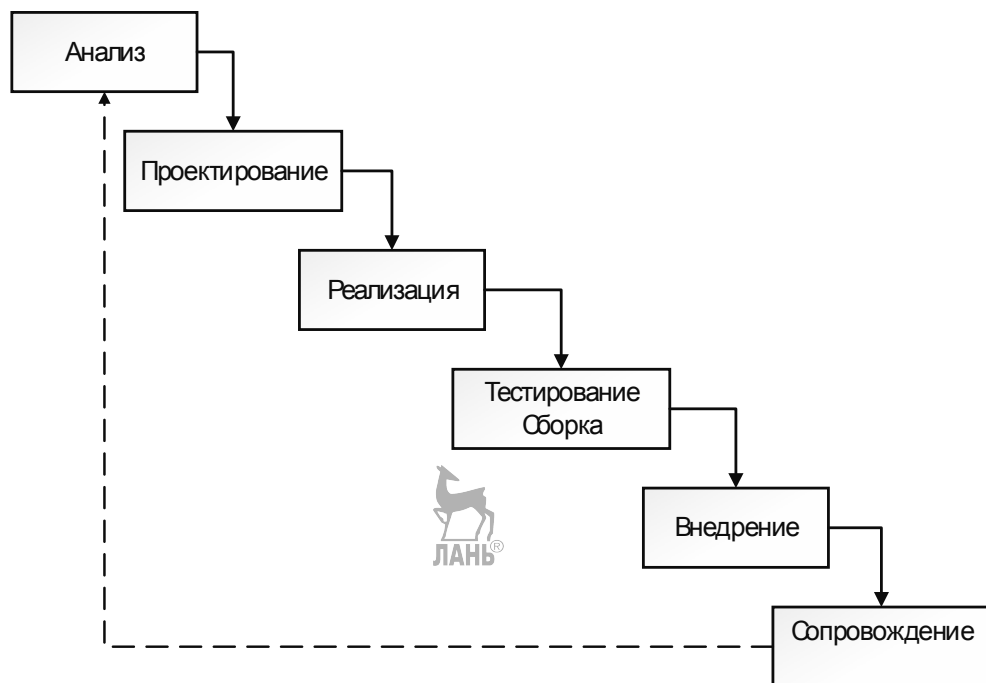


Рис. 1.1

При разработке нового программного средства трудно это обеспечить. Поэтому, согласно этой модели, можно организовать, главным образом, разработку простых систем или новых версий уже внедренных систем. Как было отмечено выше, одной из задач сопровождения является сбор предложений по модификации для разработки новых версий. Поэтому возникает показанная на рис. 1.1 обратная связь. На практике часто возникают и другие обратные связи от более поздних стадий к уже выполненным.

Примечание. В естественном водопаде вода тоже течет только в одном направлении, отсюда и название модели.

Модель «спираль» показана на рис. 1.2. Суть этой модели заключается в следующем: не ставится цель выполнения всего программного средства с начала до конца за один проход, а разработка выполняется постепенно, версия за версией, расширяя функциональные возможности. При создании очередной версии будут пройдены все этапы жизненного цикла, но только для одной подзадачи, а не для всей задачи сразу. В идеале версии должны быть аддитивными: каждая новая версия лишь расширяет возможности предыдущей, но не вносит в них изменений. При появлении новой версии сперва проводится регрессивное тестирование: проверяется, не потеряна ли работоспособность предыдущей версии при добавлении новых возможностей.

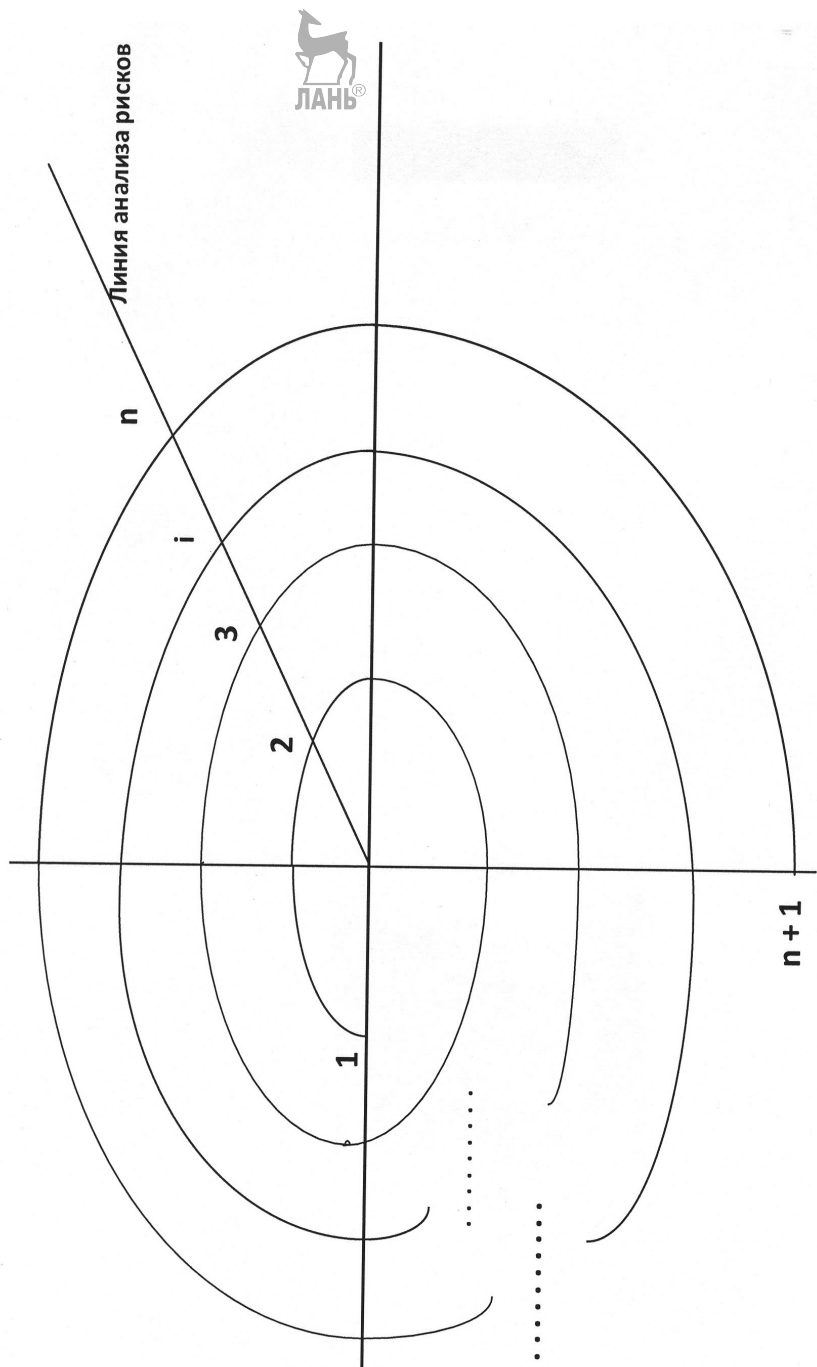


Рис. 1.2

Обеспечить аддитивность сразу с первой версии на практике часто затруднительно, тем более что первые 1–2 версии могут быть прототипами: их невозможно еще внедрить, они служат для лучшего взаимопонимания заказчика и разработчика, для уточнения постановки задачи, а также для проработки принципов реализации, особенно сложных компонентов.

Успех разработки по модели «спираль» во многом зависит от правильного определения очередности решаемых задач. Действует известный из строительства принцип: от фундамента к крыше, но не наоборот. Первые версии должны создать базу для следующих, следующие версии опираются на предыдущие. Попытка строить программное средство таким образом, что сначала решили ряд мелких прикладных задач, а потом занялись разработкой общих для них частей, привела к неудаче не одного проекта.

Рассмотрим этот процесс подробнее. Разработка начинается в точке 1. Участок 1–2 называется предпроектным исследованием: ничего не разрабатывается, заказчик излагает свои пожелания, а разработчик должен понять суть задачи и в первом приближении оценить свои возможности по их реализации. В точке 1 должно быть принято решение начинать работу или нет. При положительном решении будет поставлена задача разработки 1-й версии. От 1 до 2 будут выполнены все этапы жизненного цикла, но только для этой версии. В точке 2 подводятся итоги выполнения 1-й версии: что получилось, что нет, и принято решение: продолжать работу или нет. И так, версия за версией, будет разработано все программное средство. В точке n будет принято решение, что дальнейшая работа над этим программным средством нецелесообразна, надо приступить к разработке нового. В точке $n+1$ разработчик прекращает сопровождение.

Преимущества этой модели:

- объем работы при разработке одной версии меньше, поэтому работу легче организовать и управлять ею;
- опыт разработки предыдущих версий будет учтен при работе над следующими;
- в случае неудачи материальные потери будут меньше;
- разработанные версии могут быть внедрены, и заказчик получит реальную пользу.

2. Методологии разработки программного обеспечения

2.1. Классические методологии

2.1.1. Метод функциональной декомпозиции

Исторически первой методикой разработки программного обеспечения был метод функциональной декомпозиции. Метод и сегодня может успешно применяться при решении алгоритмически сложных задач, не связанных с обработкой больших массивов данных. Сюда относятся многие инженерные и научные задачи. Постановка задачи традиционная: Дано: исходные данные. Найти: требуемый результат. Могут быть добавлены дополнительные условия. Далее выполняется разделение заданной задачи на подзадачи. Каждая подзадача должна быть логически целой, иметь четко фиксированные вход и результат. В результате получится иерархическая диаграмма, представленная на рис. 2.1. Рекомендуется выделить при декомпозиции 2–8 (лучше 3–5) подзадач.

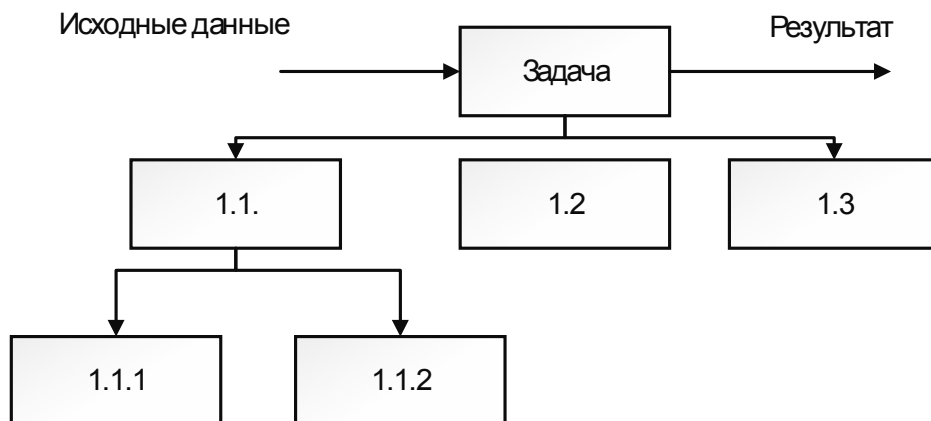


Рис. 2.1

Любая задача уровня 1 должна свестись к решению подзадач уровня 2. В противном случае допущена ошибка декомпозиции. Необязательно, чтобы все подзадачи уровня 2 были задействованы при решении всех задач уровня 1. По таким же принципам продолжаем декомпозицию.

Вопрос о вводе исходных данных и выводе результата всей программы. Часто это возлагается на главный модуль. Если в каком-то модуле нужны специфические данные (только для него), то их ввод (вывод) можно возлагать и на этот модуль. В принципе можно предусмотреть и специальный модуль ввода (вывода).

Когда заканчивать? Существуют два критерия:

1. «Счастливые случаи»:

- выделена подзадача, которая представляет собой хорошо изученную классическую задачу, методы решения которой известны (например, решить систему линейных алгебраических уравнений);
- выделена подзадача, для решения которой уже имеется программное обеспечение (например, сделанное в ходе предыдущих разработок).

2. «Обычный случай» — подзадаче нижнего уровня соответствует программа «разумных размеров», которыми считается объем 1–2 экрана.

Другим необходимым документом метода функциональной декомпозиции являются диаграммы «ввод — обработка — вывод», которые должны быть составлены для каждого узла иерархической диаграммы. Примерный внешний вид такой диаграммы показан на рис. 2.2.

Номер: например 1.1 Имя: Комментарий:		
Ввод	Обработка	Вывод
Описание исходных данных модуля	Описание процесса преобразования ввод → вывод, включая обращения к модулям более низкого уровня	Описание результата

Рис. 2.2

Процесс проектирования всегда итеративный: на начальных этапах ввод и вывод отдельных модулей могут быть заданы на естественном языке, в ходе проектирования должны быть определены язык и среда реализации, и к концу проектирования они должны быть заданы с использованием типов и структур данных выбранного языка. Также для каждого модуля должен быть определен интерфейс функции (процедуры), которая его реализует. В принципе возможно, что при выполнении какого-то модуля возникает исключительная ситуация, которая делает выполнение задачи этого модуля невозможной. Если ничего для этого не предусмотреть, то в таком случае произойдет аварийная остановка всей программы. Для уменьшения вероятности этого рекомендуют включить в состав вывода сигнальную переменную, которая сигнализирует о возникших проблемах. Чем точнее удастся определить причину исключительной ситуации, тем лучше, но хотя бы минимум: модуль выполнен / не выполнен. Вызвавший модуль более высокого уровня должен перед продолжением анализировать значение сигнальной переменной завершенного модуля. Простой пример: преобразование числа в символьную строку удастся всегда, и сигнальная переменная не нужна; перевод символьной строки в число — не всегда. Говорят, что программа должна либо выдать ответ, либо сообщить, по какой причине ответ получить не удалось — чем точнее, тем лучше.

После завершения проектирования написание программ для разных модулей можно распараллелить.

В данном пункте рассмотрена функциональная декомпозиция «сверху — вниз». В принципе возможно и проектирование «снизу — вверх», когда вся программа решения сложной задачи komponуется из уже имеющихся модулей решения отдельных задач.

2.1.2. Метод анализа потоков данных

В основе анализа потоков данных лежит диаграмма потоков данных (*DFD—Data Flow Diagram*). Она может быть успешно применена при автоматизации офисной деятельности, проектирования простых информационных систем. Более современным средством является язык *UML*, который будет подробно рассмотрен в данном пособии позже. Но и *DFD* имеют будущее из-за своей простоты и наглядности. Для рисования *DFD* применяют две нотации: Иордана — де Марко и Гейна — Сарсона. Они по существу ничем не отличаются, ограничимся рассмотрением первой из них. Применяемые обозначения и пример диаграммы приведены на рис. 2.3.

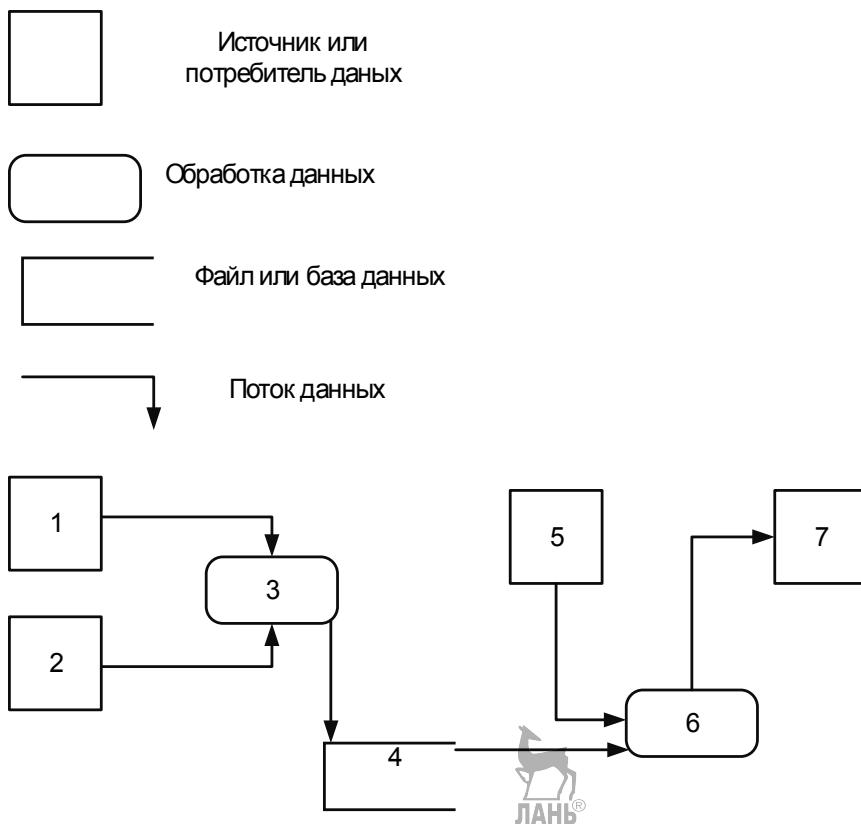


Рис. 2.3

Данные из источников 1 и 2 обрабатываются процессом 3, и результат будет записан в базу данных 4 и передан потребителю 5. Данные от потребителя 5 с привлечением данных из базы данных 4 обрабатываются в 6, результат для 7. Линии показывают передвижение данных между узлами. Для дальнейшей работы над процессами 3 и 6 может применяться метод функциональной декомпозиции: заданы ввод/вывод и описание обработки. Для реализации 4 — методы проектирования баз данных. Для 1, 2, 3, 7 необходимо разработать интерфейсы пользователя, позволяющие запустить нужные для них задачи и выполнить ввод/вывод. Представленная диаграмма может иметь и иерархическую структуру: на верхнем уровне источники (потребители) могут быть структурными подразделениями организации, но более низком уровне — сотрудники, на которых возложены те или иные обязанности.

2.2. Объектно-ориентированный подход

Изучению объектно-ориентированного подхода посвящена основная часть данного пособия, поэтому ограничимся здесь краткой характеристикой. Базовые понятия: объект и класс. **Объект** — это реально существующий предмет со всеми его индивидуальными особенностями. **Класс** — это множество объектов с одинаковыми свойствами и одинаковым поведением. Любой сложный объект может принадлежать многим классам. Например, в аудитории во время учебного процесса каждый студент — представитель класса «Студенты вуза», и у них совершенно одинаковые характеристики. Но каждый из них является и представителем других классов: клуба по интересам, своей семьи и т. д. В языках программирования под классом понимается структура данных, состоящая из данных и методов их обработки. Если дать значения данным класса, то он превратится в объект. Объект на языке программирования — это переменная типа класс.

Проектирование программного обеспечения заключается в определении классов и отношений между ними таким образом, чтобы, создавая объекты этих классов, решить поставленные задачи. В ходе работы программа, построенная по объектно-ориентированной методике, объекты обмениваются сообщениями (*Message*). С программистской точки зрения передача сообщения от одного объекта другому означает вызов функции объекта-адресата. Наиболее распространенной методологией разработки по объектно-ориентированной методике является унифицированный процесс (УП), в оригинале именуемый *RUP* (*Rational Unified Process*; *Rational* — название фирмы, проповедующей ее).

2.3. Agile-методики

Перечисленные выше методы разработки программного обеспечения относятся к так называемым «тяжелым» методикам. Это означает, что результаты всех стадий жизненного цикла должны быть документированы, и они выполняются строго последовательно. В 1990-х годах стали развиваться гибкие методики, в которых весь процесс разработки программного обеспечения заключается в тесном взаимодействии заказчика и разработчика. Будут разработаны

версия за версией (на разработку новой версии от нескольких дней до нескольких недель), каждая версия тут же тестируется с участием заказчика, и намечаются пути их усовершенствования.

Базовые принципы гибкой методик изложены в *Agile*-манифесте [3]:

1. Нашим наивысшим приоритетом является удовлетворение клиента посредством ранней и непрерывной поставки ценного программного обеспечения.
2. Приветствуем изменение требований даже на поздних стадиях разработки.
3. *Agile*-процессы позволяют использовать изменения для обеспечения заказчика конкурентного преимущества.
4. Работающий продукт следует выпускать как можно чаще, с периодичностью от пары дней до пары месяцев.
5. На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.
6. Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.
7. Непосредственное общение является наиболее практичным и эффективным способом обмена информацией как с самой командой, так и внутри команды.
8. Работающий продукт — основной показатель прогресса.
9. Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно. *Agile* помогает наладить такой устойчивый процесс разработки.
10. Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.
11. Простота — искусство минимизации лишней работы — крайне необходима.
12. Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.
13. Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

Наиболее распространенными гибкими методиками являются *SCRAM*-технология и экстремальное (*XP*) программирование. Коротко суть этих подходов заключается в следующем:

- Постоянное сотрудничество заказчика и разработчика — от постановки задачи до тестирования.
- Разработка маленькими шагами, результаты которых тут же тестируются и будут предъявлены заказчику.
- Пожелания заказчика по усовершенствованию по возможности тут же будут реализованы.
- Разработку ведет относительно малочисленная группа специалистов, которые друг другу полностью доверяют, могут договориться. Отсутствует

руководитель проекта, который распределяет задания и проверяет их выполнение.

Критерии применимости гибких методик четко изложены в [4]. Важны два параметра: критичность и масштаб. Критичность определяется последствиями потери работоспособности программного обеспечения. Выделены 4 уровня:

С — потеря удобства.

D — потеря восстанавливаемых ресурсов (материальных, финансовых).

E — потеря невозстанавливаемых ресурсов (материальных, финансовых).

L — угроза техногенных катастроф.

Масштаб определяется количеством занятых в разработке специалистов:

- 1–6 — малый;
- 7–20 — средний;
- Более 20 — большой.

Гибкие методики могут применяться при критичности С и D и при малых и средних разработках.



3. Языки UML и OCL

3.1. Общие понятия

Язык *UML* (*Unified Modeling Language*) — это язык графического моделирования, широко применяемый при разработке программных средств по объектно-ориентированной методике. Авторами языка являются «три друга», как их часто именуют в литературе, — *G. Booch*, *J. Rumbaugh* и *I. Jacobson*. Язык *UML* предназначен для выполнения этапов анализа и проектирования программных средств. Кроме того, язык *UML* может быть использован при тестировании и для управления выполнением проекта. С помощью языка *UML* можно выполнять следующие задачи:

- описание требований к разрабатываемой системе;
- описание структуры и бизнес-процессов предметной области;
- проектирование архитектуры программного продукта;
- проектирование размещения программного продукта в сети;
- генерация структуры объектно-ориентированной программы.

Как любой язык, *UML* имеет различные реализации. Так как *UML* является языком проектирования программных средств, то его реализации выполнены в виде *CASE*-средств (*Computer Aided Software Engineering*). Кроме того, для пользования им необходимо освоить методику работы с *UML*. Эти вопросы будут рассмотрены в следующих главах данного пособия. В этой главе рассмотрим структуру диаграмм.

В настоящее время последней версией является *UML 2.5*, полное описание которой приведено в [7], а сжатое описание ее диаграмм в [6]. Ограничимся рассмотрением базовых его средств. По мнению автора, удачное описание *UML* с множеством рекомендаций по его применению имеется в [8].

UML — это язык диаграмм. Приведем классификацию диаграмм, рассматриваемых далее (полную классификацию можно найти в [6]). Приведем их названия и на английском языке, потому что многие *CASE*-средства не русифицированы.

1. Структурные диаграммы:

- диаграмма классов (*Class Diagram*);
- диаграмма компонентов (*Component Diagram*);
- диаграмма размещения (*Deployment Diagram*).

2. Диаграммы поведения:

- диаграмма вариантов использования (*Use Case Diagram*);
- диаграмма деятельности (*Activity Diagram*);
- диаграмма состояний (*State Machine Diagram*);
- диаграммы взаимодействия (*Interaction Diagram*):
 - диаграмма коммуникации (*Communication Diagram*);
 - диаграмма последовательностей (*Sequence Diagram*).

Язык объектных ограничений (*Object Constraint Language, OCL*) [6] является формальным языком описания ограничений, которые могут быть использованы при различных компонентах языка *UML*. *OCL* может применяться как совместно с *UML*, так и самостоятельно. Средства *OCL* позволяют задавать не только объектные ограничения, но и другие логико-лингвистические выражения.

Приступим к рассмотрению упомянутых средств.

3.2. Диаграмма вариантов использования

Диаграмма вариантов использования (равнозначный термин «диаграмма прецедентов») предназначена для документирования требований к разрабатываемому программному продукту. Она показывает, кто является потенциальными пользователями и для решения каких задач они могут в будущем обращаться к создаваемому программному продукту.

Диаграмма вариантов использования может иметь иерархическую структуру: в таком случае на верхнем уровне используют **пакеты**. Пакет — это универсальный механизм организации элементов в группы. При проведении системного анализа с целью определения требований к новому программному продукту можно исследуемую предметную область разделить на подсистемы, каждой подсистеме поставить в соответствие пакет и на первом этапе рассматривать только связи между подсистемами. Простейшая диаграмма с использованием пакетов показана на рис. 3.1.

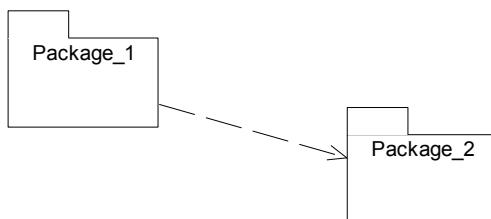


Рис. 3.1

Между пакетами допускается только одна разновидность отношений — отношение зависимости. В данном случае это означает, что пакет 1 зависит от пакета 2. Допускается использование и нескольких уровней пакетов: внутри пакета могут находиться пакеты следующего уровня.

Примечание. Пакеты могут использоваться не только в диаграммах вариантов использования, но и в других диаграммах.

Основными компонентами диаграммы вариантов использования являются **действующие лица** (в литературе используют равнозначные термины — актер, актанта), **варианты использования** (равнозначный термин — **прецедент**) и **отношения** между ними. На рис. 3.2 показаны символы *UML* для их обозначения.

Для облегчения чтения диаграммы следует использовать содержательные имена действующих лиц и вариантов использования. Кроме того, на диаграмме могут быть использованы пояснительный текст, прикрепленный к какому-то компоненту диаграммы, и комментарии, расположенные в любом месте диа-

граммы. Действующие лица – это люди, которые в будущем используют разрабатываемый программный продукт для решения прикладных задач, а также технические устройства, для управления которыми разрабатывается программа или другие программы, которые будут взаимодействовать с разрабатываемой.

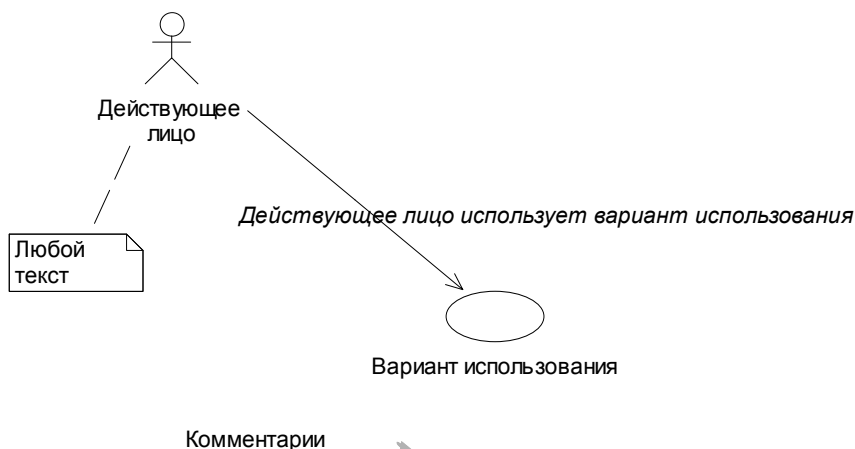


Рис. 3.2

Существенно то, что действующие лица являются внешними относительно разрабатываемого программного продукта, их внутренняя структура уточняться не будет. Варианты использования соответствуют задачам, для решения которых и разрабатывается программный продукт. В ходе дальнейшей работы их необходимо детализировать с помощью других диаграмм. На диаграмме вариантов использования нас их реализация не интересует.

На диаграмме вариантов использования могут быть применены (и показаны на ней) следующие виды отношений:

- отношение ассоциации (*association relationship*);
- отношение расширения (*extend relationship*);
- отношение включения (*include relationship*);
- отношение обобщения (*generalization relationship*).

Отношение **ассоциации** используется для задания взаимодействия действующего лица и вариантов использования: какое действующее лицо какие варианты использует. Пример этого отношения приведен на рис. 2.2. Рассмотрим применение стрелок этого отношения. Отсутствие стрелки означает, что действующее лицо запускает вариант использования, и результат передается ему же. Стрелка, направленная от действующего лица, означает, что оно запускает вариант, но результат ему не передается (за исключением сообщения типа «Выполнено»). Стрелка от варианта использования означает, что действующее лицо не может его запускать, ему лишь сообщают результат, если этот вариант был кем-то запущен.

Отношение **расширения** используется между вариантами использования: оно указывает, что один вариант расширяет возможности другого, но это рас-

ширение не имеет обязательного характера, а предоставляет дополнительные возможности, которые потребуются не всегда (например, подзадача, которая может возникнуть или нет).

Отношение **включения** показывает, что один вариант использования всегда включает и другой вариант использования (например, как подзадачу, которую придется всегда решить). Отношения расширения и включения представлены на рис. 3.3.

Отношение **обобщения** связывает менее общее с более общим. Подробно рассмотрим это отношение при обсуждении диаграмм классов. Пример отношения обобщения приведен на рис. 3.4.

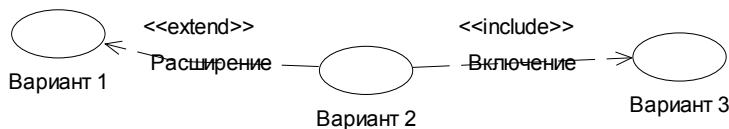


Рис. 3.3

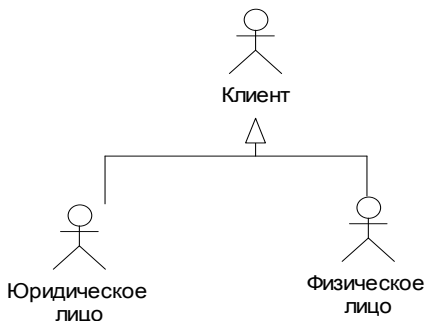


Рис. 3.4

Подведем итоги. Главное назначение диаграммы вариантов использования заключается в формализации функциональных требований к системе с помощью действующих лиц — потенциальных пользователей и вариантов использования — задач, с которыми они будут обращаться к разрабатываемому программному продукту. Рекомендуемые ограничения: действующих лиц не более 20, вариантов использования не более 50. В противном случае диаграмма теряет наглядность; в таком случае следует составить несколько диаграмм или использовать пакеты для структурирования модели. Важно сконцентрироваться исключительно на том, **что** должен делать создаваемый программный продукт, а не на том, **как** он это будет делать. Построение диаграммы вариантов использования специфицирует не только функциональные требования к проектируемой системе, но и выполняет исходную структуризацию предметной области.

Составление диаграммы вариантов использования позволяет следующее.

1. Моделируя поведение с помощью вариантов использования, эксперты предметной области могут описать взгляд на систему извне с такой

степенью детализации, что разработчики сумеют сконструировать ее внутреннее представление. Варианты использования дают возможность экспертам, конечным пользователям и разработчикам общаться на одном языке.

2. Варианты использования позволяют разработчикам понять назначение разрабатываемой системы.
3. Варианты использования являются основой тестирования элементов программного продукта на всем протяжении его жизненного цикла; они позволяют проверять корректность их реализации.

Составление диаграммы вариантов использования рекомендуется выполнять в следующей последовательности:

1. Выделить действующие лица. При этом надо четко ответить на вопрос: чем отличаются друг от друга отдельные действующие лица.
2. Для каждого действующего лица выделить варианты использования, после чего составить общий список вариантов использования.
3. Связать между собой действующие лица и варианты использования (установить отношение ассоциации).
4. Проанализировать в первом приближении наличие общих подзадач у разных вариантов использования, при их наличии выделить эти подзадачи и установить отношения расширения и включения.

Рассмотрим сказанное на небольшом учебном примере. Пусть требуется автоматизировать работу магазина. По телефону потенциальные покупатели могут получить справки о наличии и характеристиках товара, о ценах и при желании забронировать товар. Забронированный товар должен быть куплен в течение дня; забронированный, но не купленный товар после закрытия магазина считается свободным для продажи. Магазин торгует товарами малых размеров и массы, поэтому доставка товара покупателям не осуществляется. Покупатель, пришедший в магазин, может также получить информацию о наличии и характеристиках товара, о ценах. Кроме того, продавец может выписывать счет на оплату товара, копия счета передается на склад, кладовщик доставит товар на выдачу, и сотрудник на выдаче, убедившись, что товар оплачен, передает его покупателю. Товар между выписыванием счета и доставкой кладовщиком на выдачу считается забронированным. Продавец может выписывать счет и на забронированный товар, а может и сам забронировать товар. Забронированный у продавца товар обрабатывается аналогично забронированному по телефону. Выписанный, но не оплаченный в течение трех часов товар возвращается на склад и считается свободным для продажи другому покупателю. Автоматизация расчетов с покупателями и ведения бухгалтерского учета на данном этапе не рассматриваются. Менеджер магазина отслеживает расход товаров и состояние склада и принимает на этой основе решения о заказе новых партий у поставщиков или о продаже залежавшегося товара.

Действующие лица:

- менеджер;
- консультант у справочного телефона магазина;

- продавец;
- кладовщик;
- сотрудник на выдаче.

Примечание. В магазине несколько продавцов, кладовщиков и сотрудников на выдаче, но их задачи не отличаются друг от друга.

Варианты использования.

Консультант на телефоне:

- получение справок о характеристиках продаваемых товаров;
- получение справок о наличии товара на складе;
- резервирование товара;
- аннулирование резервирований по желанию покупателя или после закрытия магазина.

Продавец:

- получение справок о характеристиках продаваемых товаров;
- получение справок о наличии товара на складе;
- резервирование товара;
- выписывание счета;
- передача копии счета на склад.

Кладовщик:

- изменение количества товара на складе после отпуска;
- изменение количества товара на складе после возврата с выдачи.

Менеджер:

- получение справок о наличии товара на складе;
- анализ хода продаж отдельных видов товаров;
- анализ времени нахождения товара на складе.

Сотрудник на выдаче не осуществляет ввод данных и поэтому он не рассматривается в дальнейшем как действующее лицо. Составленные диаграммы вариантов использования представлены на рис. 3.5 и 3.6.

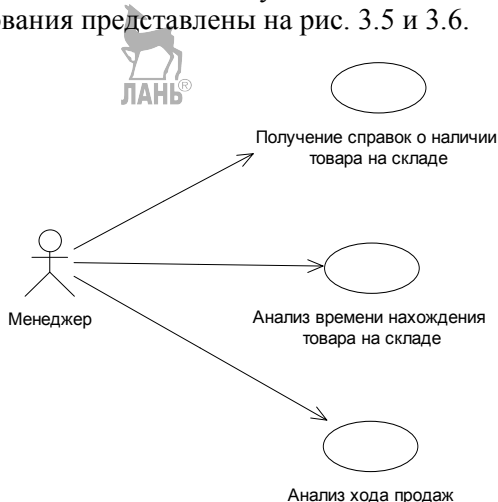


Рис. 3.5

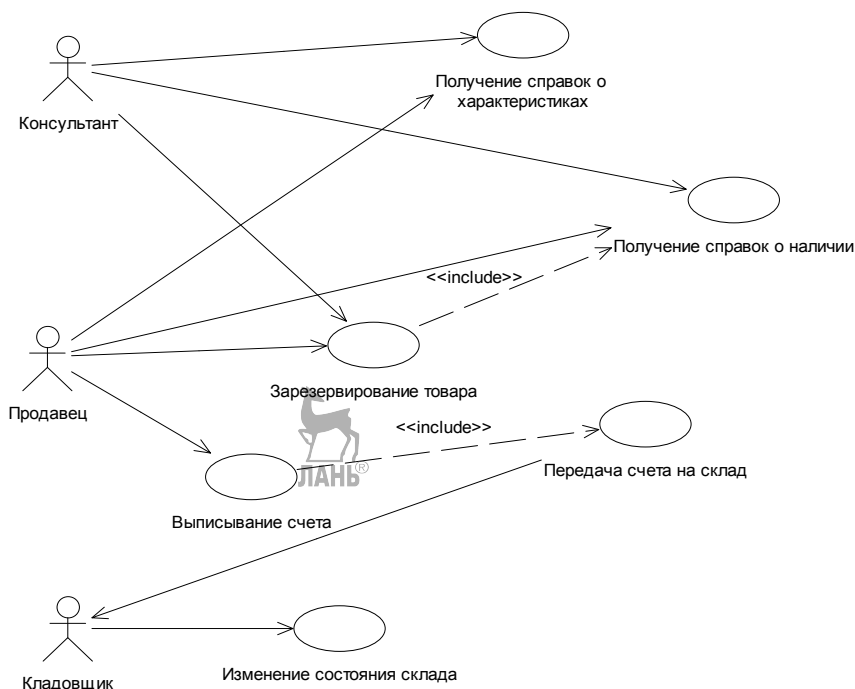


Рис. 3.6

3.3. Диаграмма классов

Диаграмма классов служит для представления статической структуры модели системы в терминологии объектно-ориентированного программирования. Диаграмма классов отражает структуру отдельных классов и их взаимосвязи и состоит из классов и отношений между ними. При большой сложности и/или объеме на диаграмме классов можно использовать пакеты. Их внешний вид, назначение и отношения между ними совпадают с описанными в п. 3.2. Обозначение класса на языке *UML* показано на рис. 3.7.

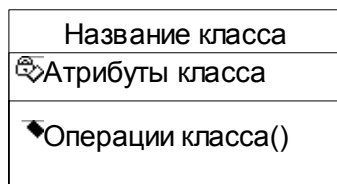


Рис. 3.7

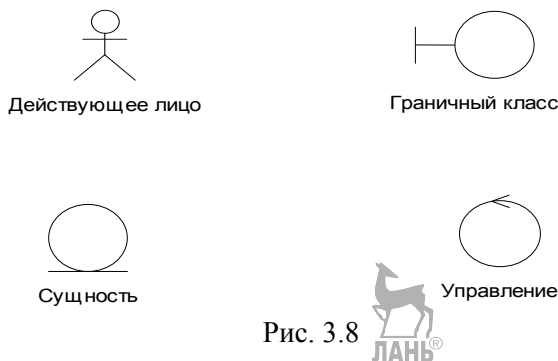
Название класса должно быть уникальным в пределах модели. Названия классов образуют в будущем словарь предметной области, и поэтому надо обращать особое внимание на выбор содержательных названий.

Атрибуты (или свойства) класса задают набор его характеристик. На первом этапе моделирования можно ограничиться лишь определением состава атрибутов, но, кроме этого, на языке *UML* имеются возможности помимо названий атрибутов определить видимость, тип данных, кратность, начальное значение. Приведенные перед именами атрибутов символы обозначают видимость. Мы вернемся к этому позже.

Операции класса задают услуги, которые могут быть запрошены у любого объекта данного класса. Операциями заданы задачи, которые могут быть решены на объектах данного класса. На *UML* для операции могут быть заданы видимость, состав аргументов, тип возвращаемого значения и некоторые другие характеристики.

Классы могут быть отнесены к определенному стереотипу. Стереотип позволяет задать назначение класса. Часть стереотипов определена на *UML*, но пользователь может расширить их состав. Примеры наиболее существенных стереотипов, рекомендованных в продуктах серии Rational, и их обозначения приведены на рис. 3.8.

Стереотип **Действующее лицо** (*Actor*) нам уже знакомо. Стереотип **Граничный класс** (*Boundary*) используется для задания взаимодействия разрабатываемого программного продукта с внешней средой. Поэтому каждому действующему лицу должен соответствовать граничный класс. Стереотип **Сущность** (*Entity*) используется для классов, предназначенных для создания баз данных (т. е. для длительного хранения данных). Сущность **Управление** (*Control*) предназначена для управления работой программного продукта. Все стереотипы классов имеют одинаковую структуру, рассмотренную нами выше: название, атрибуты, операции.



Между классами имеются следующие отношения:

- обобщения (*generalization*);
- ассоциации (*association*);
- зависимости (*dependency*);
- реализации (*realization*).

Отношение **обобщения** (рис. 3.9) связывает класс, соответствующий более общему понятию (предок), с менее общими понятиями – классами (потомками). Равносильные термины родительский класс — дочерний класс. Все, что

утверждается относительно класса-предка, должно без всяких ограничений быть верным для всех его потомков.

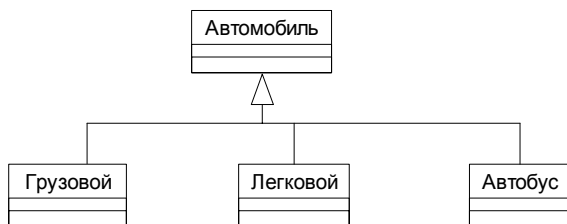


Рис. 3.9

Отношение **ассоциации** (ассоциативное отношение, рис. 3.10) показывает наличие содержательной связи между двумя классами.



Рис. 3.10

На этом рисунке:

Учеба — имя отношения (необязательно); *Имя студента*, *Название вуза* — имена ролей, которые играют связываемые этим отношением классы (необязательны); *1*, *1..** — мощность отношения, ее желательно задать.

В нашем случае имеем отношение «один ко многим»: в вузе учится много студентов, но каждый студент учится только в одном вузе. Кроме этого, существуют мощности «один к одному» и «многие ко многим». Вместо *1..** можно задать и более конкретные значения. Например, *1..4* соответствуют от 1 до 4 объектов класса; *0..3* — условное отношение от нуля до 3.

Имеются разновидности отношения ассоциации — отношение **агрегации** (*aggregation*) и **композиции** (*composition*). Отношение агрегации показывает, что один класс содержит в какой-то ситуации в своем составе другие классы, но они могут существовать и независимо друг от друга. Пример отношения агрегации приведен на рис. 3.11. Команда состоит из одного тренера и 12 игроков, тренер может тренировать 1 или 2 команды, игрок может играть в одной команде. Но как тренер, так и игроки могут существовать и за пределами своей команды. Характеристики этого отношения совпадают с характеристиками отношения ассоциации. Отношение **композиции** — это по существу отношение «целое — часть». Оно (рис. 3.12) предполагает, что составные части не имеют смысла без своего владельца.

Отношение **зависимости** определяет, что изменение одного класса может повлиять на другой класс, который его использует, причем обратное в общем случае неверно. Это отношение необходимо указать, если один класс (клиент) использует другой (сервер). Зависимость показана на рис. 3.13; изменения в *Класс_В* (сервер) могут повлиять на *Класс_А* (клиент).

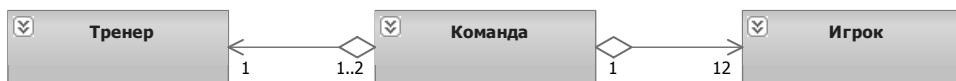


Рис. 3.11

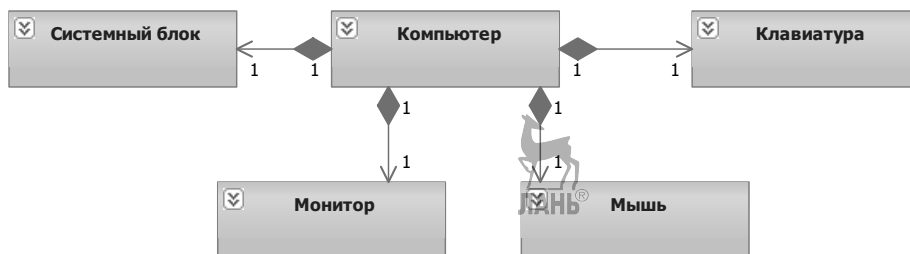


Рис. 3.12



Рис. 3.13

Отношение **реализации** (рис. 3.14) задается между классами, один из которых описывает действия, а другой гарантирует их выполнение. Чаще всего реализации используют для определения отношений между интерфейсом и классом, который предоставляет объявленные в интерфейсе операции или услуги. Интерфейс позволяет отделить спецификацию (сам интерфейс) от реализации. В нашем случае *Класс_C* реализует интерфейс *Класс_D*.



Рис. 3.14

В качестве примера рассмотрим часть диаграммы классов для магазина (рис. 3.15). Составление диаграммы классов занимает центральное место в объектно-ориентированном анализе и проектировании. От правильного выбора классов и отношений между ними в большой степени зависит успех выполнения проекта в целом. При наличии классов-сущностей возникает вопрос об обновлении условно-постоянных данных. В нашем случае данные о количестве товара можно изменять при их поступлении или отпуске; но номенклатуру данных и их характеристики необходимо время от времени изменять.

Различают диаграммы классов анализа и проектирования. Диаграмма классов анализа описывает структуру предметной области, классам соответствуют базовые ее понятия. Данные и операции при этом могут иметь качественный характер, без уточнения деталей реализации. Диаграмма классов проектирования должна быть составлена с такой степенью подробности, чтобы на ее основе можно было писать или даже генерировать заготовки программ реализации.

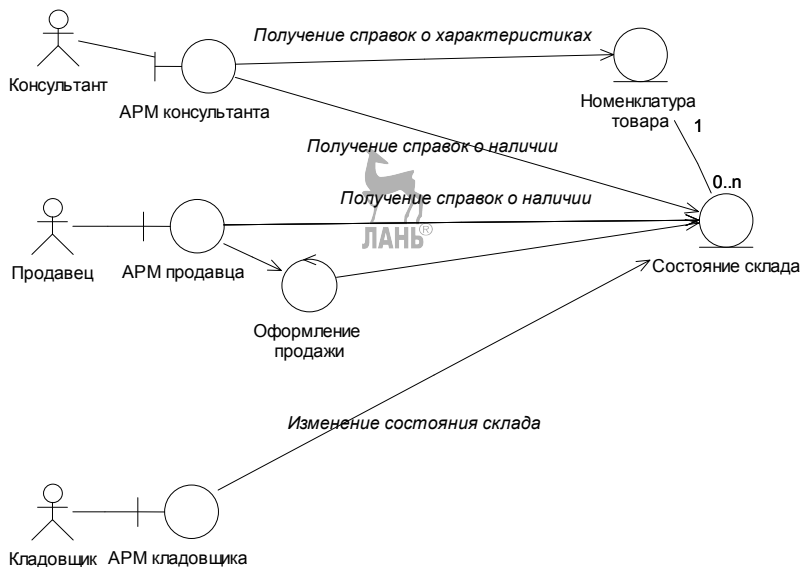


Рис. 3.15

3.4. Диаграмма состояний

В *UML 2.0* вводится понятие конечный автомат для моделирования динамического поведения объектов. Для их представления используют диаграммы состояний. Объекты многих классов могут находиться в разных состояниях. Простейший пример: лампочка может быть в двух состояниях: «горит», «не горит». Будем говорить о состояниях классов, подразумевая при этом, что все объекты этого класса будут находиться в одинаковых состояниях. Переход объектов класса из одного состояния в другое задает характерные черты их поведения. Исследование состояний проводим, исходя из следующих предположений:

- объект может в любой момент времени находиться только в одном состоянии;
- переход из одного состояния в другое происходит скачкообразно; имеются четкие критерии, позволяющие отличить одно состояние от другого (т. е. при изменении состояния должно измениться значение хотя бы одной переменной класса);

- допустимы не все переходы между состояниями: часть переходов противоречит законам природы, часть – правилам эксплуатации;
- существуют **события** (*events*), заставляющие выполнить переход из одного состояния в другое.

Составление диаграммы состояний необязательно, но она позволяет лучше понимать функционирование исследуемой системы. Если какой-либо класс имеет ярко выраженное динамическое поведение, то разработка этой диаграммы желательна. Каждая диаграмма состояний соответствует одному классу. При наличии между классами отношения обобщения необходимо в каждом случае определить, насколько отличаются диаграммы состояний классов различных уровней обобщения, и на этой основе решить, нужны ли отдельные диаграммы для классов разных уровней обобщения или нет. Такую диаграмму желательно построить в следующих случаях:

- объекты класса создаются и уничтожаются (активизируются и деактивируются) в ходе работы;
- объекты мигрируют от одного класса к другому (в пределах одного класса-предка);
- объекты накапливают значения своих атрибутов постепенно, на протяжении жизненного цикла;
- объект отчетливо проходит через разные стадии цикла;
- объект возникает или производится поэтапно;
- объект вступает в связи и выходит из них;
- если объект – это оборудование, нормальное функционирование или сбой которого оказывает влияние на другие объекты.

Если нет ни одного из перечисленных случаев, то можно обойтись и без этой диаграммы. На диаграмме используют обозначения, приведенные на рис. 3.16. На рис. 3.17 представлена упрощенная диаграмма состояний процесса обучения студента в вузе. Рекомендуем читателю на основе личного опыта усовершенствовать приведенную диаграмму.



Рис. 3.16

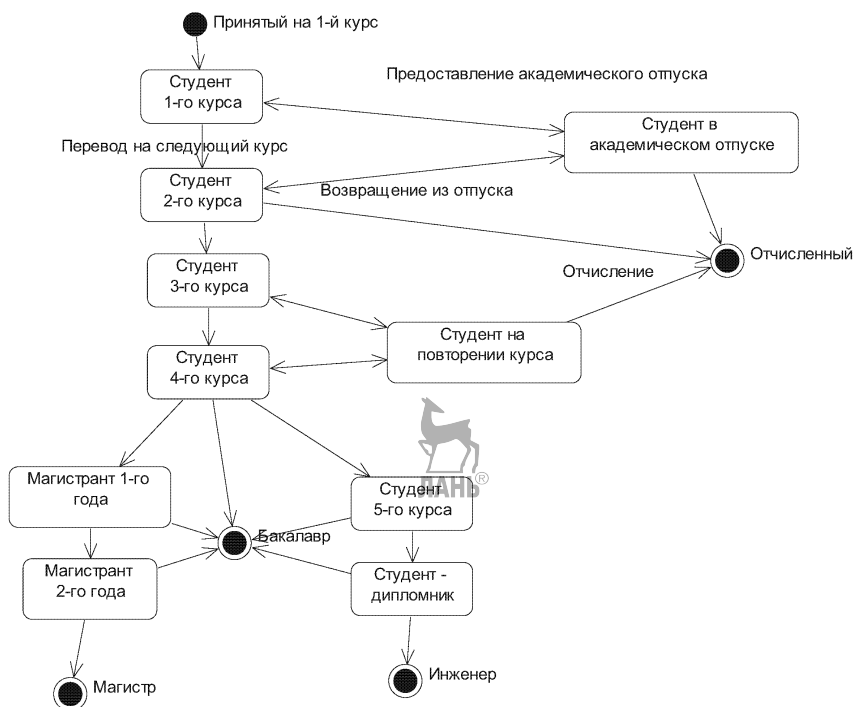


Рис. 3.17

На диаграмме состояний могут быть даны дополнительные характеристики состояний и событий (рис. 3.18).

Для **событий** можно указывать следующие дополнительные характеристики:

- имя события;
- аргументы события (при необходимости);
- условие, которое должно выполняться при возникновении данного события;
- дополнительные действия, возникающие вместе с этим событием.

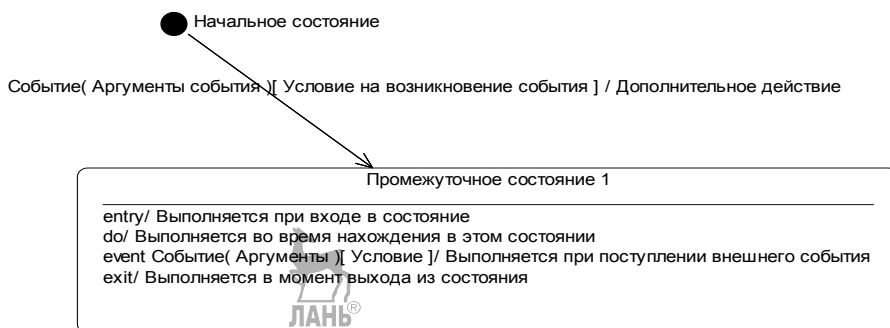


Рис. 3.18

Для состояний такими характеристиками являются:

- имя состояния;
- действия, которые должны быть выполнены в момент входа в состояние;
- действия, которые должны выполняться во время нахождения объекта в данном состоянии;
- действия, которые должны быть выполнены в момент выхода из состояния;
- действия, которые должны быть выполнены в момент поступления внешнего события; в таком случае дополнительно могут быть заданы аргументы внешнего события и условия на значения этих аргументов.

Все действия, как в состоянии, так и при событии, задаются в следующей форме:

\wedge имя_класса.имя_события(аргументы_события)

Состояние может иметь и подсостояния, например диаграмма состояний коробки передач автомобиля (рис. 3.19).

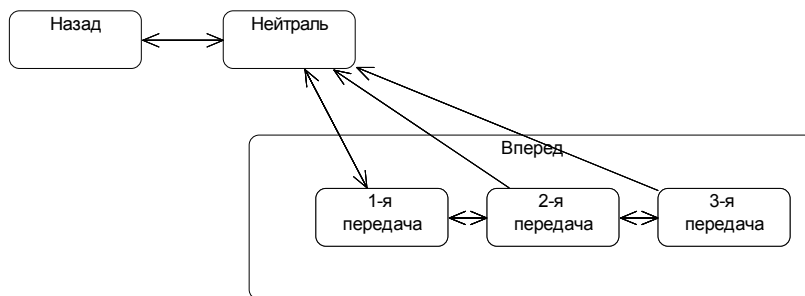


Рис. 3.19

3.5. Диаграмма деятельности

Диаграммы деятельности предназначены для моделирования поведения, иногда их называют «ОО блок-схемы». Например, с их помощью можно описать работу варианта использования. Обозначения, применяемые на этой диаграмме, поясним на примере рис. 3.20. **Действие** — это элементарная работа, не подлежащая дальнейшей детализации. На рис. 3.20 после начала процесса выполняется действие 1. После его завершения выполняется проверка условия 1 и в зависимости от результатов этой проверки выполняется лишь одно из действий (2, 3 или 4). После его завершения процесс продолжается. Обратим внимание на то, что после разветвления соединение должно обозначаться знаком, показанным на рисунке. Линия синхронизации означает, что следующие за ней действия могут выполняться одновременно, но все они должны быть выполнены: действия 5 и 7 могут начинаться одновременно, действие 6 может выполняться после завершения 5. Использование объектов необязательно, они используются лишь тогда, когда надо четко описать результат действия (в нашем случае действия 7). Сигнал означает, что информация о завершении действия

будет передана за пределы описываемой системы (например, таймер выключил прибор и подал звуковой сигнал). Прием сигнала означает, что в этот момент следует ждать сигнала со стороны. Далее от узла соединения можно двигаться лишь при условии, что все параллельно выполняемые ветви завершились, не обязательно одновременно. В нашем случае после завершения действий 6 и 7 и поступления сигнала следует проверка условия 2, выполнение действия 8 или 9 и один из двух вариантов завершения.

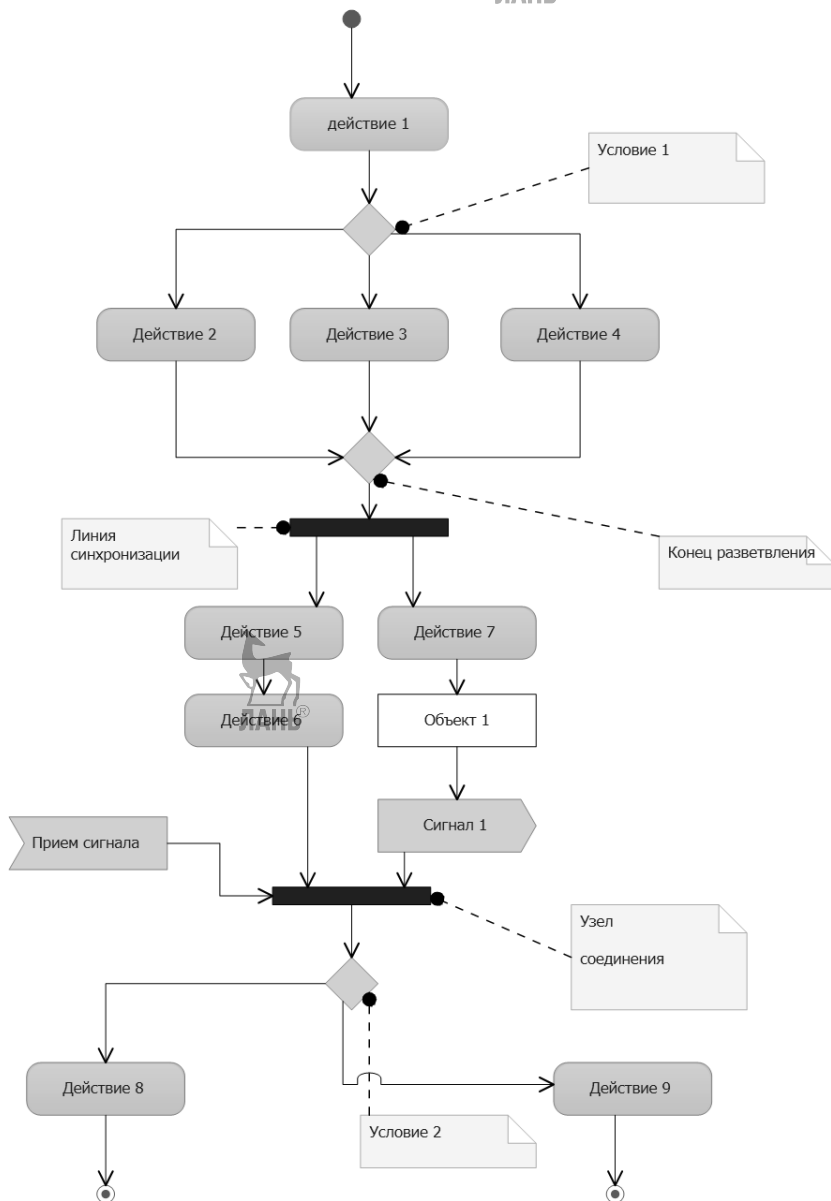


Рис. 3.20

Простой пример диаграммы деятельности магазина приведен на рис. 3.21.

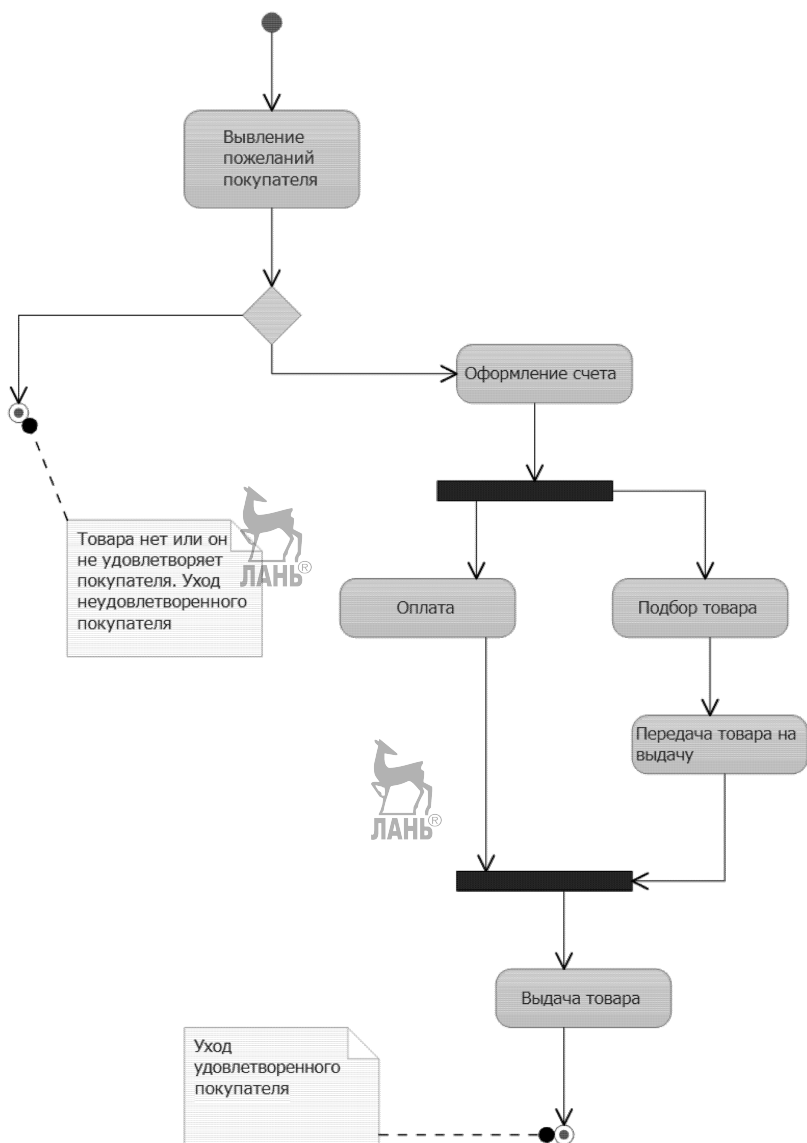


Рис. 3.21

3.6. Диаграмма последовательностей

Рассмотрим коротко функционирование программ, построенных по объектно-ориентированной методике. Работа программы заключается в обмене сообщениями между объектами. Любой объект является представителем какого-либо класса. В общем случае можно иметь один или более объектов одного класса. **Сообщение** (*message*) представляет собой законченный фрагмент ин-

формации, который отправляется одним объектом другому. С программистской точки зрения сообщение — это обращение из одного объекта к методу в составе другого объекта или использование свойства класса-адресата (в языках объектно-ориентированного программирования, в которых это понятие присутствует). Различают **синхронные** и **асинхронные** сообщения. При подаче синхронного сообщения процесс может продолжаться лишь после получения ответа от адресата. При асинхронном сообщении процесс продолжается сразу после его отправки. Естественно, что решение задачи сводится к обмену сообщениями между объектами в определенной последовательности. Для представления этого процесса и используется диаграмма последовательностей.

В общем случае каждому варианту использования должна соответствовать диаграмма последовательностей, которая отражает последовательность обмена сообщениями именно для этого варианта. Исполнение многих вариантов использования может протекать по-разному в зависимости от обстоятельств, например резервирование товара в случае наличия или отсутствия требуемого количества.

Рассмотрим компоненты диаграммы последовательностей на рис. 3.22. На диаграмме обозначены объекты, задействованные при исполнении данного варианта использования. Обозначение объектов: имя_объекта:имя_класса

Если принадлежность какого-то объекта к классу еще не установлена или нет требуемого класса, то можно указать только имя объекта. Имя класса будет установлено позже. Если имеется только один объект какого-то класса, то можно ограничиться только именем класса. У верхнего края диаграммы на горизонтальной линии находятся объекты, которые существуют с самого начала работы программы. Линия жизни объекта изображается пунктирной вертикальной линией. Эта линия служит для обозначения периода времени, в течение которого объект существует и может участвовать в обмене сообщениями. Очевидно, что для создания объекта должен быть запущен конструктор его класса, а также даны значения некоторому подмножеству его данных. Следует также иметь в виду, что один объект может передать сообщение другому лишь тогда, когда среди его данных имеется адрес того.

Отдельные объекты могут быть уничтожены до завершения работы программы — в таком случае на их линии жизни ставится знак × (объект :*Класс2*). Часто уничтожение является результатом направления к такому объекту сообщения со стереотипом *destroy()*. Возможно и создание нового объекта в ходе работы программы. Такой объект показан не у верхнего края диаграммы, а ниже; к такому объекту направлено сообщение стереотипа *create()*. (на рисунке *Объект3:Класс3*). При составлении этой диаграммы надо иметь в виду, что для создания объекта необходимо запустить конструктор его класса и обеспечить передачу созданному объекту нужных для его функционирования данных. поэтому на практике сообщение *create()*. для каждого объекта должно быть предусмотрено.

В процессе функционирования объектно-ориентированных систем одни объекты могут находиться в активном состоянии, непосредственно выполняя

определенные действия, или в состоянии пассивного ожидания сообщений от других объектов. Для явного выделения состояния объекта применяется **фокус управления**, который изображается в форме вытянутого узкого прямоугольника, верхняя сторона которого обозначает начало активности, а нижняя сторона – ее конец. Естественно, фокус управления должен целиком находиться на линии жизни объекта. Периоды активности объекта могут чередоваться с периодами ожидания. На рис. 3.22 *Объект1* и *Объект2* имеют постоянный фокус управления, у *:Класс2* периоды активности чередуются; потом объект *:Класс2* будет уничтожен. Даже если потом объект этого класса будет вновь создан, это будет уже новый объект.

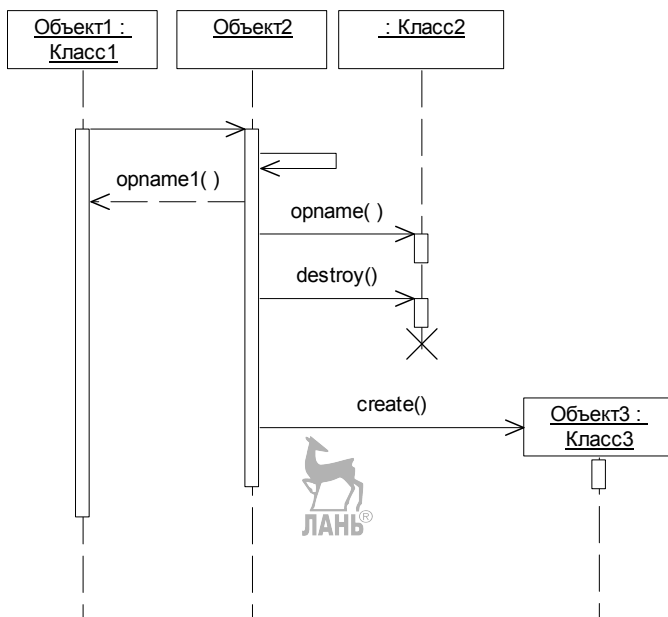


Рис. 3.22

Примечание. На диаграмме последовательности ось времени направлена сверху вниз.

Сообщения показаны на диаграмме горизонтальными линиями. Имеются следующие стереотипы сообщений:

- *Call* (вызвать) — сообщение, требующее вызова метода класса-адресата. Это сообщение может быть рефлексивным, тогда оно требует вызова метода в классе-отправителе;
- *Return* (возвратить) — возвращает результат выполнения сообщения Вызвать;
- *Create* (создать) — сообщение, требующее создание другого объекта;
- *Destroy* (уничтожить) — сообщение с требованием уничтожения объекта-адресата.

Рядом с сообщением может быть дано его имя. За именем в скобках могут быть при необходимости параметры сообщения. В качестве имени сообщения можно использовать только имена методов класса-адресата. Если не создан класс для объекта-адресата или в этом классе нет методов, то дать имена сообщениям невозможно.

Упрощенная диаграмма последовательности процесса продажи товара показана на рис. 3.23. Процесс продажи состоит из следующих действий:

- уточнение и обсуждение с покупателем характеристик товара и выбор подходящего изделия;
- подтверждение наличия товара на складе;
- продажа товара.

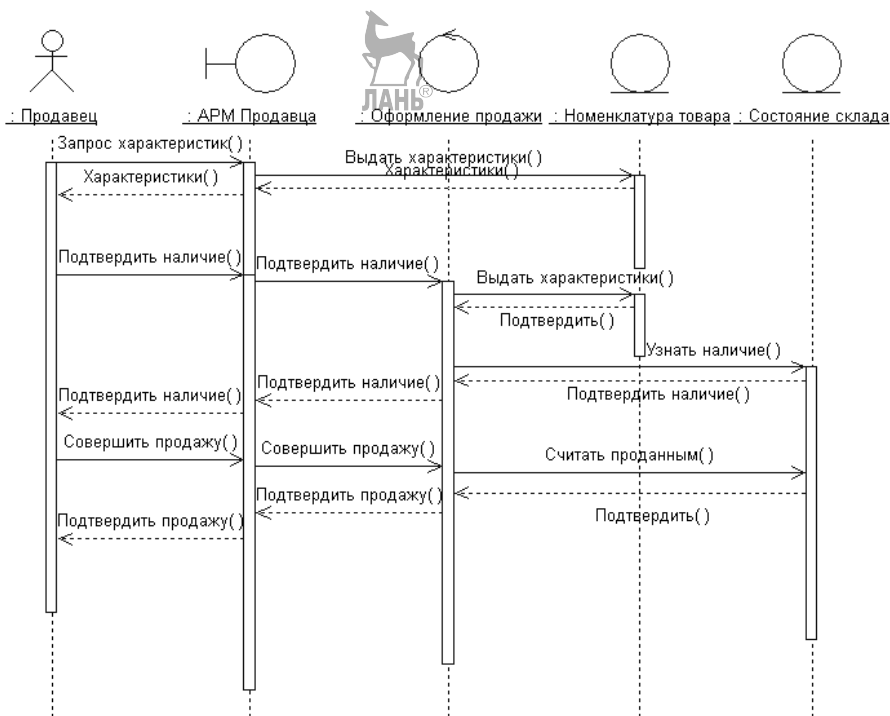


Рис. 3.23

Диаграмма последовательности служит, в первую очередь, для визуализации временных аспектов взаимодействия. Диаграмма кооперации служит для представления структурных аспектов. Эти две диаграммы вместе называются диаграммами взаимодействия. Простейшая диаграмма кооперации показана на рис. 3.24. Диаграмма кооперации соответствует одному варианту использования и показывает, объекты каких классов будут задействованы при его выполнении. Возможности диаграмм последовательностей намного шире рассмотренных, за более подробной информацией можно обратиться в [8].

3.7. Диаграмма кооперации

Диаграмма последовательности служит, в первую очередь, для визуализации временных аспектов взаимодействия. Диаграмма кооперации служит для представления структурных аспектов. Эти две диаграммы вместе называются диаграммами взаимодействия. Диаграмма кооперации соответствует одному варианту использования и показывает, объекты каких классов будут задействованы при его выполнении.

На рис. 3.25 приведена кооперативная диаграмма, соответствующая приведенной на рис. 3.23 диаграмме последовательностей.

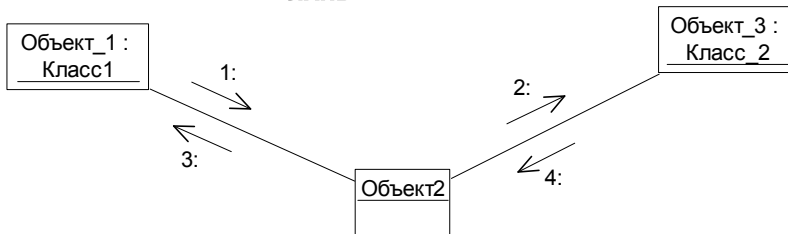


Рис. 3.24

На этой диаграмме показаны три объекта, принадлежность к классу одного из них (Объект2) не установлена. Наличие сплошной линии между объектами показывает, что между ними могут передаваться сообщения. Сами сообщения показаны рядом с линиями. Цифры у сообщений показывают очередность их возникновения. Рядом с сообщениями может быть указан метод класса адресата. На основе диаграммы последовательности кооперативная диаграмма может быть создана автоматически.

3.8. Диаграмма компонентов

Рассмотренные до сих пор диаграммы относятся к логическому представлению. Особенность логического представления состоит в том, что оно оперирует понятиями (классы, ассоциации, состояния, сообщения), которые не имеют самостоятельного материального воплощения. Эти понятия отражают наше понимание структуры физической системы или аспекты ее поведения.

Диаграмма компонентов описывает особенности физического представления создаваемой системы. Компонент — это модульная и замещаемая часть системы, который взаимодействует с другими компонентами через строго определенный интерфейс. Замена одного компонента на другой с таким же интерфейсом и функционалом не должна влиять на работу системы в целом. Есть термин «компонентно-ориентированная разработка»: использование в новой разработке созданных ранее компонентов с целью снижения затрат при создании нового программного средства.

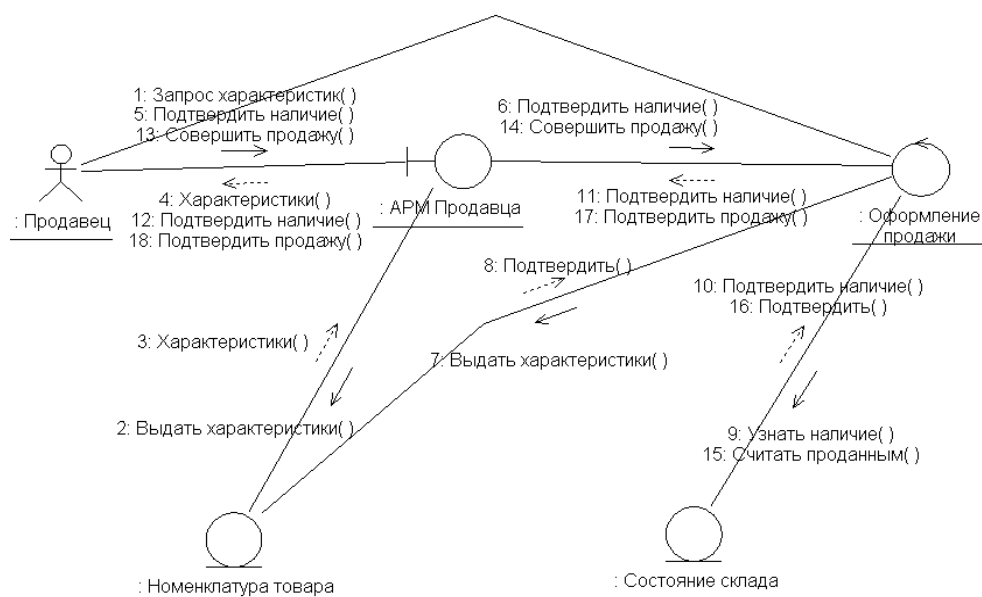


Рис. 3.25

На рис. 3.26 показана диаграмма компонентов. Все классы должны содержаться в компонентах. Другими словами, компоненты состоят из классов.

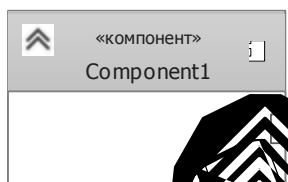


Рис. 3.26

3.9. Диаграмма развертывания

В наше время редко когда разрабатывается приложение, функционирующее на одном компьютере. Поэтому возникает необходимость показать, какие компоненты на каких компьютерах работают и как эти компьютеры связаны. Для этого служит диаграмма развертывания. В версии UML 2.5 имеется возможность показать разные типы узлов сети и связей между ними. Проектирование сетей — это самостоятельная интересная задача. При проектировании программного обеспечения необходимо учитывать и многие особенности функционирования сетей. Например, можем ли мы рассчитывать, что компьютеры связаны между собой всегда, или должны предусмотреть и их автономную работу. Обсуждение этой проблематики выходит за рамки данного пособия.

3.10. Язык OCL

При объектно-ориентированном анализе и проектировании рассмотренных выше средств (разные диаграммы) недостаточно для построения точных и непротиворечивых моделей сложных систем. Часто необходимо определить дополнительные условия на компоненты диаграммы. Традиционно для этого использовали естественный язык, но такие описания страдают неоднозначностью и нечеткостью.

Для адекватного отражения семантики основных процессов в анализируемой предметной области был разработан язык *OCL* [8]. *OCL* является языком формальных выражений, которые используются для записи отдельных ограничений. Этот язык дополняет составленные на *UML* графические модели. Выражения на *OCL* никогда не могут изменять значения атрибутов и состояние системы, они могут только учитываться при осуществлении таких изменений.

Язык *OCL* не является традиционным языком программирования: в нем нет средств для записи алгоритмов решаемых задач. Средства этого языка лишь фиксируют необходимость выполнения тех или иных условий применительно к отдельным компонентам моделей.

OCL применяется в следующих случаях:

- как язык запросов;
- для записи инвариантов классов и стереотипов;
- для записи пред- и постусловий операций;
- для описания условий защиты элементов модели;
- для записи условий сообщений;
- для записи ограничений на операции.

Выражения на языке *OCL* представляют собой утверждения относительно некоторого объекта на *UML*. Выражения *OCL* являются контекстно-зависимыми в том смысле, что каждое выражение рассматривается только в связи с тем объектом *UML*, к которому оно прикреплено. В модели выражения используются для записи некоторых условий, которым должны удовлетворять все объекты некоторого класса. В таком случае выражение называется инвариантом. Выражения также могут задавать пред- и постусловия операций. Ограничимся рассмотрением на примерах применения *OCL* для написания пред- и постусловий операций и инвариантов классов.

1. *x1.max(x2)*--нахождение максимума
post: *if x1 >= x2 then result=x1 else result =x2 endif*
2. *x.floor()*-- нахождение наибольшего целого
post: *result <= x1 and (result+1) > x*
3. *i1.div(i2)*-- целочисленное деление
pre: *i2 <> 0*
post: *if i1/i2 >= 0 then result = (i1/i2).floor()
else result = -(i1/i2).floor()endif*

pre: задает предусловие, т. е. условие, которое должно быть выполнено перед операцией, в противном случае операция не может быть выполнена (не определена);

post: задает постусловие, которое выполняется после операции.

Примечание. В предусловиях и постусловиях не задают условия на типы данных, для которых определена данная операция, или принадлежность результата к типу. В примере 1 тип данных несущественный; в примере 2: *x* должен быть вещественным числом, результат будет целым; в примере 3: *x1* и *x2* должны быть целыми, результат тоже будет целый.

Рассмотрим использование инвариантов класса. **Инвариант** — это условие, которое должно выполняться всегда, за исключением времени внесения изменений. Для каждого инварианта необходимо задать контекст. Пусть задан класс, представленный на рис. 3.27.

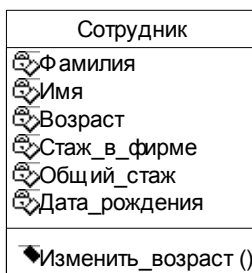


Рис. 3.27

Для задания контекста используем выражение:

context *Сотрудник inv*

self.Общий_стаж >= *self.Стаж_в_фирме*

self.Общий_стаж <= *self.Возраст* — 16

self.Общий_стаж >= 0

self.Возраст >= 16

Служебное слово **self** означает ссылку на текущий контекст (в данном случае **Сотрудник**), а слово **inv** означает инвариант. По существу написанное выше означает, что для любого сотрудника в любой момент стаж в фирме не может быть больше общего стажа; общий стаж должен быть хотя бы на 16 лет меньше возраста; стаж не может быть отрицательным, но может быть нулевым, и возраст не может быть меньше 16 лет. Если какие-то методы вносят или изменяют значения соответствующих полей, то эти условия должны всегда выполняться, и изменения, которые их нарушают, не должны быть осуществлены. Очевидно, что инварианты класса являются постусловиями всех его операций.

4. Выполнение этапов анализа и проектирования на языке UML

4.1. Основные положения

В предыдущих главах были рассмотрены средства для выполнения анализа и проектирования программного обеспечения. В этой главе будет рассмотрена методика использования этих средств, точнее, будет рассмотрен унифицированный процесс разработки программного обеспечения (*The Unified Software-Development Process*) согласно [8]. Фундаментальные принципы унифицированного процесса (в дальнейшем УП) заключаются в следующих утверждениях:

- УП управляется вариантами использования;
- УП является архитектурно-ориентированным;
- УП является итеративным и инкрементным.

Рассмотрим эти утверждения подробнее.

Любой программный продукт создается для обслуживания пользователей. Очевидно, для его построения мы должны знать, в чем нуждаются потенциальные пользователи. Понятие *пользователь* — это необязательно человек; это может быть и техническое устройство, для управления которым создается новая программа, или другой программный продукт, с которым новой программе предстоит взаимодействовать. В любом случае пользователь — это нечто внешнее относительно создаваемого программного продукта. Пользователь обращается к программному продукту со своими задачами и ждет от него определенной последовательности действий. Такое взаимодействие является *вариантом использования*. Вариант использования — это часть функциональности системы, необходимая для получения пользователем значимого результата. Сумма всех вариантов использования составляет *модель вариантов использования*, которая может быть представлена с помощью рассмотренных ранее *диаграмм вариантов использования*. Таким образом, модель вариантов использования позволяет описывать функциональность создаваемого программного продукта. На самом деле модель вариантов использования — это не только средства описания функциональных требований к программе — она направляет проектирование, реализацию и тестирование. На основе модели вариантов использования разработчики по очереди создают серию моделей проектирования и реализации, которые и осуществляют выделенные варианты использования. Завершается процесс разработки тестированием, в ходе которого среди прочего должно быть проверено качество реализации разработанной системой функциональных требований, заложенных в модели вариантов использования.

Управляемый вариантами использования процесс означает, что разработка проходит серию рабочих процессов, порожденных вариантами использования.

Понятие *архитектура программного продукта* включает в себя наиболее важные статические и динамические его аспекты. Архитектура — это представление всего проекта с выделением важных характеристик и затушевывани-

ем деталей. Варианты использования определяют функции, а архитектура — форму. Для определения формы разрабатываемой системы архитектор должен проанализировать ключевые функции будущей системы. Как показывает опыт, ключевые функции отражены примерно в 5–10% вариантов использования, которые содержат функции ядра системы. На основе анализа ключевых вариантов использования разрабатывается архитектура создаваемого программного продукта, которая в ходе дальнейшей работы над проектом должна расширяться и дополняться, но не переделываться кардинальным образом.

Разработка программных продуктов — это сложная работа. Поэтому целесообразно разделить работу на относительно небольшие части, так называемые мини-проекты. Каждый мини-проект является *итерацией*, результатом которой будет *приращение* (*инкремент*). Итерации относятся к процессу разработки, а приращения — к выполняемому проекту. Разработчики выбирают задачи, которые должны быть решены в ходе ближайшей итерации. В процессе итерации следует работать с группой вариантов использования, их реализация повышает применимость продукта. При этом надо уделить внимание рискам, которые могут возникнуть в ходе разработки. Итерация является мини-проектом еще и потому, что в ее ходе для выделенных вариантов использования выполняют все традиционные этапы жизненного цикла программного продукта: от анализа до тестирования. Приращение после выполнения очередной итерации необязательно аддитивно: разработчики могут заменить ранее разработанные компоненты более совершенными. На более поздних стадиях разработки приращения обычно аддитивные. Важно отметить, что успех реализации проекта в целом во многом зависит от успешного определения очередности реализации вариантов использования, т. е. от итераций.

Использование итеративной и инкрементной разработки имеет следующие преимущества:

- уменьшает финансовые риски затратами на одну итерацию;
- как показывает опыт, желания пользователей и связанные с ними требования не могут быть определены в начале разработки, а уточняются в последовательных итерациях. В [8] это сформулировано так: «Единственное, что постоянно в разработке программ — это изменение требований»;
- итерации позволяют ускорить процесс разработки в целом, потому что короткий и четкий план разработки предпочтительнее длинного и непонятного и способствует эффективной работе разработчиков;
- на каждой итерации выполняется тестирование, что позволяет вовремя заметить допущенные ошибки и недостатки и принять своевременно меры для их устранения.

При реализации сложных проектов большого объема результатом первых итераций может быть лишь прототип будущей системы, который не может быть внедрен, а используется лишь как база для будущих итераций, для отработки принципов реализации, для достижения лучшего взаимопонимания заказчика и разработчиков.

Для разработки программного продукта УП циклически повторяется: каждый цикл включает в себя фазы анализа, проектирования, реализации и внедрения. Каждая фаза состоит из 2–3 итераций. Каждый цикл представляет собой мини-проект, реализуемый по модели «мини-водопад».

Каждая фаза заканчивается вехой; веха определяется по наличию определенного набора моделей, представленных в заранее оговоренном виде. На каждой вехе следует ответить на вопросы: что удалось достичь на законченной фазе, что не удалось (в таком случае возникают вопросы: Почему? Что надо сделать?), что планируется на следующей фазе.

Разработка нового программного продукта начинается с составления модели вариантов использования, которая в ходе разработки:

- описывается в модели анализа;
- реализуется в модели проектирования;
- распределяется в модели развертывания;
- реализуется в модели реализации;
- проверяется в модели тестирования.

Модель вариантов использования дает ответ на вопрос: «Что должна делать система?», оставляя ответ на вопрос: «Как?» на потом.

Модель анализа предназначена для уточнения деталей вариантов использования и создания первичного распределения поведения системы по набору объектов. Составление модели анализа заключается в создании диаграмм классов.

Модель проектирования определяет статическую структуру системы (подсистемы, классы, интерфейсы) и реализованные в виде кооперации варианты использования. Эта модель описывается диаграммами классов и взаимодействий (диаграммы последовательности и кооперативная диаграмма).

Модель реализации состоит из компонентов и определяет раскладку классов по компонентам (диаграмма компонентов).

Модель развертывания определяет физические компьютеры и распределения компонентов по ним (диаграмма развертывания).

Модель тестирования описывает тесты для проверки вариантов использования.

Кроме перечисленных могут быть составлены модели предметной области и бизнес-модели для описания контекста системы, т. е. для описания принципов функционирования той предметной области, для которой создается новый программный продукт. Для модели предметной области подходит диаграмма классов, для бизнес-модели — диаграмма деятельности.

Подведем итоги. В ходе **фазы анализа** идея превращается в концепцию готового продукта и создается план его разработки. Можно сказать и так: прикладная задача превращается в программистскую задачу. Должны быть получены ответы на следующие вопросы:

- что система должна делать для ее основных пользователей;
- как должна выглядеть архитектура системы;
- каков план и во что обойдется разработка продукта?

В ходе **фазы проектирования** детально описывается большинство вариантов использования и разрабатывается архитектура системы. В ходе этой фазы определяются наиболее критичные варианты использования и создается базовая архитектура. Необходимо иметь ответ на вопрос: достаточно ли проработаны варианты использования, архитектура и план; взяты ли риски под контроль настолько, что можно взять на себя обязательство выполнения всей работы при заданных временных и финансовых ограничениях.

В ходе **фазы реализации** происходит создание продукта. Базовый уровень архитектуры разрастается до полной развитой системы. В конце этой фазы должен существовать полноценный продукт, готовый для передачи заказчику. Вопрос окончания фазы: удовлетворяет ли полученный продукт требованиям заказчика.

В ходе **фазы внедрения** продукт существует в виде бета-версии. Небольшое число квалифицированных пользователей, работая с бета-версией, сообщают об обнаруженных дефектах и недостатках. Дефекты и недостатки разделяют на две категории: требующие немедленного устранения и устранение которых можно отложить до следующей фазы построения.

4.2. Определение требований

Цель определения требований – ответить на вопрос, что должно делать создаваемое программное обеспечение для каждого потенциального пользователя? Правильный и исчерпывающий ответ на этот главный вопрос позволит направить процесс разработки на получение правильной системы. Определением требований занимаются системные аналитики. Основное внимание следует обратить на то, что клиенты, с которыми мы работаем (которые являются главным нашим источником информации и, как правило, не являются специалистами по информационным технологиям), должны быть способны прочесть и понять результаты определения требований. Для выполнения этого условия для описания требований следует использовать язык клиента. Типовой рабочий процесс определения требований включает следующие тесно взаимосвязанные шаги:

- перечисление возможных требований;
- описание контекста системы;
- определение функциональных требований;
- определение нефункциональных требований.

Перечисление возможных требований. На первом этапе могут рассматриваться все предложения о включении их в список требований (список идей). Каждое предложение имеет краткое название, описание его содержания и следующие характеристики:

- состояние предложения (предложено, одобрено, включено в план работ, отклонено);
- сметная стоимость реализации (оценивается разработчиком экспертным путем);
- приоритет (критический — если этого нет, то вся реализация теряет смысл; важный; вспомогательный);

- уровень риска, связанного с реализацией предложения (высокий, средний, низкий).

Накопленная таким образом информация позволяет решить, в какую итерацию следует включить его реализацию.

Описание контекста системы должно четко определить ее границы, что внутри и что останется за ее пределами. Это, в первую очередь, влияет на определение функциональных требований. Контекст определяется составом действующих лиц и их вариантов использования. Важной частью описания контекста является создание глоссария: списка применяемых в предметной области терминов и их определений. Это необходимо для того, чтобы в дальнейшем заказчик и разработчик «говорили на одном языке». В глоссарии должны быть устранены синонимия (несколько терминов для обозначения одного и того же понятия) и омонимия (одно слово обозначает разные понятия).

Определение функциональных требований (какими функциями должно обладать создаваемое программное обеспечение) заключается в разработке модели вариантов использования.

Определение нефункциональных требований заключается в определении характеристик создаваемого программного обеспечения, важнейшими среди них являются:

- ограничения на программно-техническую среду реализации (технические средства, операционная система, система управления базами данных, инструментальная система реализации);
- объемно-временные характеристики (предполагаемый объем базы данных, ограничения на время реализации, количество одновременно работающих пользователей);
- требования к защите данных (достаточно традиционных имени пользователя и пароля или нужно применять средства защиты данных);
- требования к надежности создаваемого программного обеспечения.

Нефункциональные требования могут быть связаны с вариантами использования (например, ограничения на время отклика) или относиться к системе в целом (например, надежность). В табл. 4.1 представлено резюме изложенного.

Таблица 4.1

Операция	Результат	
Перечисление кандидатов в требования	Список предложений	
Разобраться в контексте системы	Определение границ системы. Глоссарий.	
Определить функциональные требования	Модель вариантов использования	Соответствуют традиционному техническому заданию
Определить нефункциональные требования	Дополнительные требования	

Рассмотрим более подробно определение функциональных требований, потому что ни одна разработка программного продукта не может обойтись без этого. Процесс составления модели вариантов использования протекает следующим образом. Сначала *системный аналитик* выделяет действующие лица и варианты использования и готовит первую версию модели. Он должен обеспечить отражение в модели всех требований к функциональности. Затем *архитектор* выделяет существенные для архитектуры варианты использования, присваивает им приоритеты и определяет те из них, которые будут реализованы в первой итерации (в первом мини-проекте). После этого *спецификаторы вариантов использования* описывают все варианты использования в порядке их приоритетов. Параллельно с этим *разработчики пользовательского интерфейса* предлагают для каждого действующего лица интерфейсы пользователя, основанные на вариантах использования. Затем системный аналитик реструктурирует модель, определяя общие части вариантов использования. Для завершения этой работы требуется 1–2 итерации.

Действующих лиц и варианты использования выделяют для:

- отделения системы от окружения;
- определения, какие действующие лица взаимодействуют с системой и какие функциональные возможности ожидаются от системы.

Системный аналитик совместно с заказчиком должен идентифицировать потенциальных пользователей и разделить их по категориям. При этом надо иметь в виду следующее:

- при выявлении кандидата в действующее лицо следует найти хотя бы одного пользователя, который в состоянии оценить его и который был бы реальным его представителем;
- перекрытие ролей разных действующих лиц должно быть минимальным, другими словами, мы должны четко сформулировать, чем отличаются разные кандидаты в действующие лица.

Выделенные действующие лица должны быть идентифицированы и снабжены кратким описанием.

При определении вариантов использования надо помнить, что каждый вариант должен приносить для действующих лиц ценный для них результат. Кроме того, мы должны четко сформулировать, чем отличаются различные варианты использования. Каждый выделенный вариант использования должен иметь имя и краткое описание. В дальнейшем в ходе детализации вариантов использования должны быть получены ответы на следующие вопросы.

1. При каких предусловиях вариант использования может быть запущен, что делать, если предусловия не выполнены (например, из-за ошибок действующего лица)?
2. Какие существуют пути реализации варианта использования: выбираемые действующим лицом или зависящие от хода выполнения?
3. Как заканчивается выполнение варианта использования? Какие имеют конечные состояния? Как о результате узнает действующее лицо?

4. Какие существуют запрещенные пути выполнения? Как предотвращается их запуск? Должно ли передаваться сообщение о попытке их запуска?
5. Что в процессе выполнения делает действующее лицо и что делает программа?
6. Какие исходные данные (пока на качественном уровне) требуются от действующего лица?
7. Если один и тот же вариант использования связан с двумя или более действующими лицами, зависит ли его выполнение от очередности его запуска?

Результаты проведенного анализа можно оформлять в виде табл. 4.2. Обращаем еще раз внимание на то, что диаграмма вариантов использования должна дать ответ на вопрос «Что должна делать создаваемая программа?», недопустимо превращать ее в диаграмму функциональной декомпозиции!



Таблица 4.2

Название	Значение
Название варианта использования	
Краткое описание	
Действующие лица	
Предусловие запуска	
Основной путь выполнения	
Постусловие	
Альтернативный(ые) путь(и) выполнения	
Постусловия альтернативного пути	

В более сложных случаях можно рекомендовать использовать для этой цели диаграмму активности (показывая на ней процесс выполнения или хотя бы его основные моменты).

При разработке прототипов пользовательских интерфейсов для действующих лиц необходимо получить ответы на следующие вопросы:

1. Какие элементы пользовательского интерфейса необходимы ему для выполнения его вариантов использования?
2. Какие действия он может производить?
3. Какую информацию он должен вводить при запуске различных вариантов использования?
4. Каким образом он будет информирован о результатах выполнения (с учетом того, что ход выполнения и результат могут быть различными)?

В завершение выполняется структурирование модели вариантов использования, в ходе которого:

- извлекаются общие и совместно используемые разными действующими лицами описания функциональности;



- извлекаются дополнительные и необязательные описания функциональности;
- проводят классификацию действующих лиц.

В результате выполнения перечисленного в диаграммах вариантов использования могут появиться отношения обобщения между действующими лицами и отношения расширения и включения между вариантами использования.

Подведем итоги. В результате выполнения работ по определению требований должны быть получены (программа-минимум):

1. Описание контекста системы.
2. Модель вариантов использования (одна или несколько диаграмм).
3. Эскизы интерфейсов пользователей для всех действующих лиц.
4. Описания вариантов использования.
5. Основные нефункциональные требования.

4.3. Уточнение и структурирование требований

В результате выполнения предыдущей стадии жизненного цикла мы имеем модель вариантов использования, которая описана понятными заказчику средствами, дает ответы на вопросы «Что должна делать программа?», «Каким характеристиками должна обладать программа?» и ввиду своей специфики может содержать неточности и неоднозначности. На этом заканчиваются предпроектные исследования и начинается собственно разработка.

На предстоящей стадии жизненного цикла мы должны разработать аналитическую модель, которая составлена на языке разработчика, описывает реализацию функциональности, не должна содержать неопределенностей и неоднозначностей и может служить основой для проектирования.

Центральной задачей является составление модели классов анализа (состоит из диаграмм классов). Для этого необходимо выделить классы анализа и определить отношения между ними. Классы анализа обладают следующими характеристиками:

- сосредоточены на представлении функциональных требований, откладывают нефункциональные требования до фазы проектирования;
- имеют атрибуты, но в основном на концептуальном уровне, без уточнения типов и структур данных;
- их операции заданы концептуально — что надо делать?
- имеют отношения с другими классами, но эти отношения концептуального значения.

Составление модели классов желательно начинать с выделения классов-сущностей. Напомним, они предназначены для хранения условно-постоянных данных, необходимых при выполнении вариантов использования. Такие классы называют стабильными. Определение их структуры выполняется методами проектирования баз данных (в простых случаях можно ограничиться применением файлов). На данном этапе следует определить лишь состав хранимых данных. Операции таких классов, как правило, простые: добавление, удаление, изменение данных и выдача данных по запросу. Однако нельзя упустить следую-

щее: каким образом будет выполняться внесение изменений? Это делается уже выделенными вариантами использования или нет. В противном случае придется добавить новые варианты использования для обслуживания этих данных. Как показывает практика, часто труднее всего следить за изменениями редко изменяемых данных.

Классы анализа должны отражать ключевые понятия предметной области, объекты этих классов должны обеспечивать решение всех выделенных вариантов использования. Согласно [8], хороший класс обладает следующими признаками:

- имя означает назначение;
- ему соответствует конкретный объект предметной области;
- у него четко определенный набор обязанностей — операций;
- у него высокая внутренняя связность (его операции обрабатывают его же данные);
- у него низкая связанность с другими классами.

Рекомендованы следующие количественные характеристики:

- у класса не более 5 обязанностей, т. е. функций, доступных извне;
- классы связаны между собой отношениями;
- не должно быть классов, где одни данные или одни операции;
- уровней обобщения не более 3.

Для выявления состава классов анализа рекомендуют написать словесное описание предметной области, существительные и устойчивые словосочетания являются кандидатами на классы и атрибуты, глаголы — кандидаты на операции. В процессе определения классов анализа рекомендуют использовать *CRT*-карты (*class — responsibilities — collaborators*; класс — обязанности — участники), представленные на рис. 4.1.

Имя класса:	
Обязанности (операции)	Связи с другими классами

Рис. 4.1

Размер *CRT*-карты не более 5–7 сантиметров (чтобы нельзя было создать слишком большие классы). Заполнение *CRT*-карт выполняется в два этапа: сбор информации для их первоначального заполнения и анализ полученной информации для определения окончательного их состава. Как любой процесс проектирования, этот процесс тоже итеративный. После определения состава классов необходимо определить отношения между ними. Состав отношений был рассмотрен выше. Отношение обобщения часто задает классификацию между классами анализа. Напомним, что все что записано для класса-предка, должно остаться в силе для всех его наследников без всяких ограничений.

Можно рекомендовать при наличии многих общих атрибутов и операций у нескольких классов специально ввести для них обобщающий класс, компонентами которого они будут. Для отношения ассоциации важно не забыть указать их мощность, эта информация будет очень полезна при проектировании.

Следующей задачей при уточнении и структурировании требований является составление диаграмм деятельности для всех вариантов использования (может быть, за исключением самых простых). Помните, в предыдущем разделе мы говорили о необходимости в процессе определения требований задать для каждого варианта использования основной и альтернативные пути выполнения. Теперь настала пора проанализировать и отразить на диаграмме деятельности все возможные пути выполнения. Например, человек подошел к банкомату и хотел снять наличные. Основной поток очевиден: он их получил в желаемом количестве. Для определения альтернативных потоков следует тщательно проанализировать все причины, по каким основной поток не может быть выполнен. Программа управления банкоматом должна обрабатывать все эти случаи, сообщить пользователю о причинах неудач и восстановить исходное состояние.

Возникает еще одна проблема, решение которой находится за пределами программной инженерии. Мы моделируем с помощью диаграмм деятельности процессы такими, как они выполняются в настоящее время (называют это *as-is*). В общем случае протекание этих процессов может претерпеть заметные изменения после внедрения разрабатываемого программного обеспечения, особенно при разработке информационных систем. Поэтому требуется разработать диаграммы деятельности для протекания этих процессов после внедрения (называют это *to-be*). Эти вопросы необходимо обговорить с заказчиком.

Следующей задачей является создание **диаграмм кооперации**, показывающих участие объектов при выполнении вариантов использования. Диаграмма кооперации состоит из объектов и их взаимосвязей, которые отражают передачу сообщений между ними. Напомним, что *объект* — это реально существующий предмет со всеми своими индивидуальными характеристиками, выраженными значениями его свойств. *Класс* — это множество объектов, имеющих одинаковые свойства (список характеристик) и одинаковое поведение (состав методов). Минимальное количество диаграмм кооперации равно количеству вариантов использования. Каждая из них показывает, какие объекты каких классов будут задействованы при его выполнении. Связь между объектами на этой диаграмме показывает, что между этими объектами осуществляется обмен сообщениями в ходе выполнения варианта использования. Суть этих сообщений и их временная очередность будут уточнены в ходе проектирования.

Составление диаграммы кооперации позволяет критически осмыслить составленную ранее диаграмму классов. С одной стороны, может оказаться, что для выполнения какого-то варианта использования нет класса, объекты которого нужны, — следует добавить новый класс. С другой стороны, если имеется

класс, объекты которого не участвуют в выполнении ни одного варианта использования, то необходимо изучить вопрос: нужен ли он? Если такой класс является обобщением задействованных классов и/или связан отношением ассоциации с задействованным классом, то скорее всего он действительно нужен. В противном случае вопрос требует дополнительного рассмотрения.

В любом случае процесс анализа и уточнения требований итерационный и все названные работы выполняются взаимосвязанно.

В итоге мы должны иметь:

- модель классов анализа, на которой атрибуты и операции классов заданы концептуально (задает статическую структуру предметной области);
- модель деятельности (задает динамику выполнения вариантов использования);
- модель кооперации (задает участие классов анализа при выполнении вариантов использования).

4.4. Проектирование архитектуры

Согласно [4], архитектура — это базовая организация системы, воплощенная в ее компонентах, их отношениях между собой и с окружением, а также принципы, определяющие проектирование и развитие системы. В информационных технологиях понятие архитектур используется в следующих областях:

- функциональная архитектура;
- системная архитектура;
- технологическая архитектура;
- информационная архитектура.

Проблематика разработки архитектуры подробно рассмотрена в [4].

Основными задачами фазы проектирования являются следующие.

1. Понимание моментов, относящихся к нефункциональным требованиям и ограничениям, связанным с языками проектирования, многократным использованием компонентов, программно-технической средой реализации, технологиями баз данных, распределенной и параллельной обработками.
2. Создание исходных данных для последующей реализации установленных требований в подсистемах, интерфейсах, классах.
3. Получение возможности разбиения работ по реализации между несколькими исполнителями, работающими параллельно.
4. Определение интерфейсов между подсистемами на ранней стадии жизненного цикла. Эти интерфейсы можно потом использовать для синхронизации разных разработчиков.
5. Получение возможности визуализации проекта с помощью типовой нотации.
6. Создание абстракции реализации системы, при которой реализация связана с наращиванием, а не изменением структуры.

В проектировании участвуют *архитектор, инженер по вариантам использования и инженер по компонентам*, которые выполняют следующие виды деятельности.

Архитектор проектирует архитектуру и отвечает за целостность моделей проектирования, гарантирует корректность, согласованность и читаемость моделей в целом. Архитектор также отвечает за модель развертывания.

Инженер по вариантам использования проектирует варианты использования, отвечает за их целостность и гарантирует выполнение предъявленных к ним требований, отраженных в моделях вариантов использования и анализа.

Инженер по компонентам занимается проектированием классов и подсистем, гарантирует выполнение предъявленных к ним требований; отвечает за целостность подсистем: их классов и связей с другими подсистемами.

Проектирование архитектуры включает *определение узлов и сетевой конфигурации, определение компонентов и их интерфейсов, определение архитектурно значимых классов, определение обобщенных механизмов проектирования*. Рассмотрим по очереди названные задачи.

Определение узлов и сетевой конфигурации.

В наше время редко когда программный продукт реализуется только на одном компьютере. Возможны, в принципе, два варианта: сеть уже имеется, и новый продукт разрабатывается с учетом этого, или необходимо проектировать и сеть. Проектирование сетей — это самостоятельная задача, которая выходит за рамки данного пособия. Считаем, что сеть задана: известно, в каких узлах какие компьютеры (или другое оборудование) находятся, какие у них характеристики и как узлы связаны между собой. Физическая реализация связи для проектирования программ не очень существенна, важно учитывать, имеем ли мы постоянную связь с большой пропускной способностью, или связь может быть установлена только в определенное время, или пропускная способность каналов ограничена. Разрабатывается модель развертывания, которая оформляется диаграммой развертывания.

Определение подсистем и их интерфейсов.

Подсистемы представляют собой метод организации модели проектирования в виде набора обозримых частей. В ходе определения состава подсистем исследуется возможность повторного использования собственных разработок, а также целесообразность закупки готовых компонентов других разработчиков. Реализацией подсистемы являются компоненты, которые взаимодействуют друг с другом через строго фиксированные интерфейсы.

Принято выделять четыре уровня подсистем:

- специфический уровень приложений;
- общий уровень приложений;
- средний уровень;
- уровень системного программного обеспечения.

Специфический уровень приложений предназначен для реализации одного варианта использования. Общий уровень приложений необходим для реализации нескольких вариантов использования (например, общая для не-

скольких пользователей база данных). Средний уровень представляет собой основание разрабатываемой системы (среда реализации и ее компоненты). Уровень системного программного обеспечения — это операционные системы и стандартные программные продукты, используемые многими средами реализации.

Определение подсистем начинается с прикладных подсистем. Если во время анализа была проведена декомпозиция на пакеты анализа, можно использовать их в роли соответствующих подсистем проектирования. На фазе проектирования предстоит распределить подсистемы по узлам. Поэтому может возникнуть необходимость декомпозиции подсистем для их размещения в разных узлах.

После уточнения состава подсистем необходимо проанализировать зависимости между ними. Напомним, что отношения между подсистемами должны быть сведены до минимума. Если между подсистемами установлена зависимость, то между ними должен быть интерфейс, который будет обеспечивать реализацию этой зависимости.

Результатом проектирования архитектуры являются диаграммы развертывания, компонентов и классов. На диаграмме классов подсистеме соответствует пакет. Должно быть задано прикрепление подсистем к узлам сети.

Определение архитектурно значимых классов.

Некоторые классы проектирования могут быть изначально описаны по найденным в ходе анализа архитектурно значимым классам. Таким же образом отношения между этими классами анализа могут быть использованы для получения первого варианта отношений между классами проектирования. Каждый класс должен участвовать в реализации хотя бы одного варианта использования.

Классы разделяют на *активные* и *неактивные*. Методы неактивных классов запускаются только после получения сообщения извне. Активный класс имеет собственную нить управления и может работать параллельно с другими активными классами или «запущенными» извне неактивными классами. При необходимости наличия активных классов проектировщику необходимо тщательно проработать многие дополнительные вопросы (параллельное выполнение, запуск и останов активных классов, разрешение конфликтов между активными классами и т. д.).

Определение обобщенных механизмов проектирования

Перечислим коротко некоторые дополнительные вопросы, требующие решения на этапе проектирования.

- Возможность обеспечения длительного хранения объектов некоторых классов — проектирование для них базы данных, возможно, с обеспечением одновременного доступа многим пользователям.
- Вопросы защиты данных, прав доступа.
- Вопросы одновременной работы с одним объектом многими пользователями.

4.5. Проектирование классов и подсистем

В ходе проектирования классов и подсистем диаграмма классов анализа будет преобразована в диаграмму классов проектирования. Составление диаграмм классов является важнейшей задачей проектирования, его главным результатом. Все остальные диаграммы служат в основном для того, чтобы помочь качественно создать диаграмму классов. В конце проектирования классов диаграммы должны быть составлены с такой степенью точности, чтобы на их основе можно было написать или даже генерировать заготовки программ. Из этого следует, что до проектирования классов должны быть определены язык и среда реализации. Проектирование классов и проектирование вариантов использования являются итерационными процессами, выполняемыми параллельно.

В основе диаграммы классов проектирования лежат диаграмма классов анализа и классы, входящие в состав среды реализации и упрощающие реализацию разных функций. Например, создание интерфейса пользователя, обеспечение связи с базами данных, работу со сложными динамическими структурами данных.

В результате проектирования классов все данные класса должны быть заданы с использованием типов и структур данных языка реализации и иметь имена, допущенные там, а операции — с помощью функции, структура которых опять-таки соответствует требованиям среды реализации. Состав функций вытекает из обязанностей класса, определенных на предшествующих этапах разработки. Если окажется, что какая-то функция слишком сложная для непосредственной реализации, то можно выполнить ее декомпозицию рассмотренными выше методами. Тогда в составе класса образуются функции, вызываемые другими функциями этого же класса, но недоступные извне. Обязанности класса, выделенные ранее, образуют интерфейсную часть класса, а полученные в результате их декомпозиции функции — его внутренность.

«Хороший» класс должен обладать следующими свойствами:

- Быть внутренне связанным. Его функции обрабатывают его же данные, невозможно разделить класс на два компонента связности, каждая из которых имеет свои данные и функции, которые не пересекаются.
- Иметь минимально возможные связи с другими классами.
- Соблюдение количественных ограничений, приведенных выше.

Рассмотрим подробнее установление отношений между классами на диаграмме проектирования.

Отношение обобщения (все, что утверждаем относительно класса-предка, должно остаться в силе для всех его потомков). Это отношение (рис. 4.2 между *A* и *B*) трансформируется в отношение наследования на языках объектно-ориентированного программирования. Отношение наследования означает, что все данные и методы (кроме закрытых) класса предка доступны в классах-наследниках. В нашем случае в классе *B* будут доступны переменная *X1* и функция *f1*. С учетом этого может оказаться целесообразным их пересмотр в связанных с отношением наследования классах. Кроме того, в целях уменьшения объема программной реализации рекомендуется выделить общие в не-

скольких классах данные и методы, включить их в отдельный класс, который становится их предком. Это даже тогда, когда полученный класс не является обобщением. Обратим внимание на то, что отношение наследования является статическим, во время выполнения программы его менять невозможно. Изменения, внесенные в базовый класс, автоматически передаются всем его наследникам.

Программная реализация отношений агрегации и композиции (рис. 4.2 между *B* и *C*) сводится к появлению в *B* переменной типа класс *C*. Поэтому мы через нее получим в *B* доступ к данным и методам класса *C*. Эта связь динамическая: в ходе выполнения программы можно создать и уничтожить переменную типа *C*.

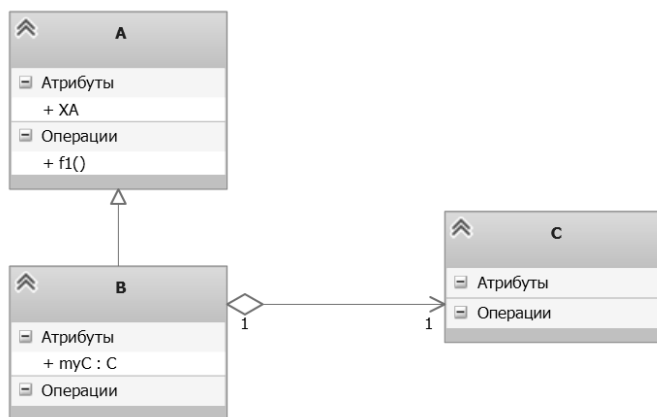


Рис. 4.2

Возникает вопрос: в каком случае предпочтительно отношение наследования, в каком случае агрегация? Когда предпочтительным является вариант рис. 4.3, а когда вариант рис. 4.4? Рекомендуется использовать вариант рис. 4.3, когда *Class2* никогда не может стать классом *Class3*. Например, автомобиль — грузовик — автобус. Когда такое изменение возможно, то предпочтителен вариант рис. 4.4. Например, сотрудник — программист — тестировщик. В случае такого изменения меняем лишь одно поле «должность», все остальное остается на местах.

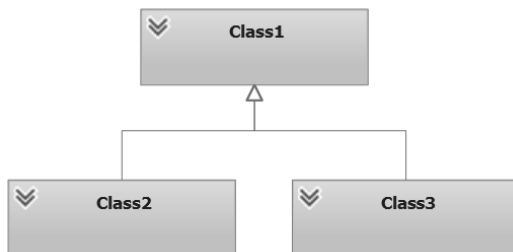


Рис. 4.3



Рис. 4.4

Особого рассмотрения требует случай, когда отношение ассоциации имеет мощность «многие ко многим». Отношение композиции не может в принципе иметь такую мощность, часть может в любой момент быть частью только одного целого, и целое может содержать несколько одинаковых частей. Отношение агрегации может иметь и мощность «многие ко многим». Например, фирма и сотрудник: в фирме много сотрудников, но каждый из них может тоже работать в нескольких фирмах. Программную реализацию отношения «один ко многим» рассмотрим позже, в разделе генерации программ. Но отношение «многие ко многим» должно быть устранено уже на стадии проектирования классов. Для этого используют дополнительный класс — ассоциацию. На рис. 4.5 приведен такой случай: в фильме заняты много актеров, каждый актер снимался в нескольких фильмах.

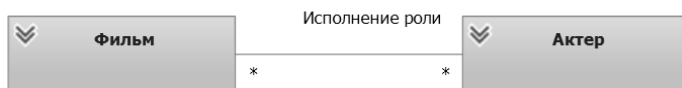


Рис. 4.5

Введем дополнительный класс-ассоциацию «Исполнение ролей». После этого класс «Фильм» содержит данные только о фильме, класс «Актер» — данные о актере, но класс «Исполнение ролей» показывает, в каком фильме какой актер кого играл (рис. 4.6).

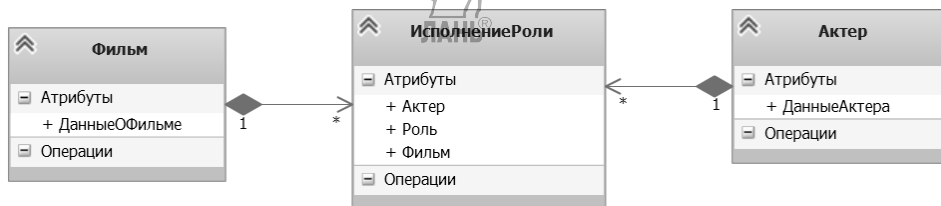


Рис. 4.6

4.6. Проектирование вариантов использования

Проектирование вариантов использования заключается в решении следующих задач:

- определение классов проектирования и/или подсистем, объекты которых необходимы для реализации вариантов использования;

- определение требований к методам классов и/или подсистем и их интерфейсам;
- определение требований к реализации вариантов использования.

Начинается процесс определения классов проектирования, объекты которых участвуют в реализации одного варианта использования, с изучения соответствующей части модели анализа. Если выяснится, что уже выделенных классов недостаточно, то инженер по вариантам использования по согласованию с архитектором и инженером по компонентам дополняют имеющийся состав классов. После этого для каждого варианта использования создается диаграмма последовательностей, которая показывает исполнение этого варианта во временной последовательности сообщений, передаваемых между объектами выделенных классов. Реализация варианта использования порождается сообщением, полученным от действующего лица.

Передача сообщения между объектами по сути означает вызов метода класса-адресата. Сообщения бывают синхронными (отправитель ждет ответ от адресата) или асинхронными (сообщение будет отправлено и отправитель продолжает работу). На полностью законченной диаграмме последовательностей:

- каждому объекту соответствует класс, представителем которого он является;
- каждому сообщению между классами соответствует метод в классе-адресате.

При динамическом поведении класса можно рекомендовать создать и для него диаграмму состояний, которая позволит лучше описать его функционирование.

Если возможна реализация какого-то варианта использования по-разному, то для всех этих вариантов разрабатывают свою диаграмму последовательностей. Надо обратить также внимание на возможные альтернативные пути выполнения варианта использования и составить для них тоже диаграмму последовательностей. Конечно, количество диаграмм последовательностей может быть сокращено, если один случай реализации варианта использования является сокращенным вариантом другого. В *UML 2.0* диаграммы последовательностей имеют еще много дополнений, которые не поместились в данное пособие и могут упростить этот процесс.

При простой логике выполнения варианта использования можно для него диаграмму последовательностей и не рисовать, а ограничиться диаграммой кооперации, дополненной операциями, передаваемыми между объектами.

Иногда в реализации варианта использования удобнее оперировать не только классами, но и подсистемами. В таком случае на диаграмме последовательностей будет показана передача сообщения подсистеме (а не объекту), и в дальнейшем придется проанализировать обработку полученного сообщения уже внутри подсистемы.

Определение требований к реализации заключается в уточнении в основном нефункциональных требований, связанных с реализацией варианта использования.

4.7. Использование среды *Microsoft Visual Studio*

для выполнения анализа

Рассмотрим в этом пункте создание диаграмм вариантов использования, классов анализа и деятельности в упомянутой среде. Создадим проект, выберем тип *Modelingproject*, дадим ему имя (в нашем случае (*Modelingproject47*) и определим его местонахождение. В этом проекте будут созданы названные диаграммы. В обозревателе решений поставим курсор на имя проекта, нажмем на правую клавишу мыши, выберем по очереди *Add —NewItem — UseCaseDiagram* (диаграмма вариантов использования). Откроется форма для создания выбранной диаграммы, на палитре инструментов (*Toolbars*) появятся рассмотренные нами выше компоненты этой диаграммы. Создание диаграммы заключается в перетаскивании нужных компонентов с палитры на диаграмму и уточнении их свойств. Перенесем на диаграмму два действующих лица (*Actor*) и три варианта использования (*Usecase*). В окне свойств (*Properties*) в поле Имя (*Name*) дадим им содержательные названия, которые будут показаны на диаграмме (рис. 4.7). Для установления связи между действующими лицами и вариантами использования используем отношение *Association*. Щелкнем левой клавишей мыши на имени отношения, после этого ставим курсор мыши на начало отношения и, держа нажатой левую клавишу, потянем конец отношения на нужный компонент диаграммы. Стрелка отношения будет соответствовать направлению рисования. Значение стрелок было рассмотрено выше. Если мы на диаграмме выделим отношение, то в свойствах появятся группы *FirstRole* и *SecondRole* (первая роль и вторая роль в соответствии с направлением рисования связи). Наличие и отсутствие стрелки определяется свойством *IsNavigable* (*True — False*), которое будет видно после открытия группы (знаком +). Отношения включения (*include*) и расширения (*extend*) можно установить аналогично. Их назначение тоже было рассмотрено выше.

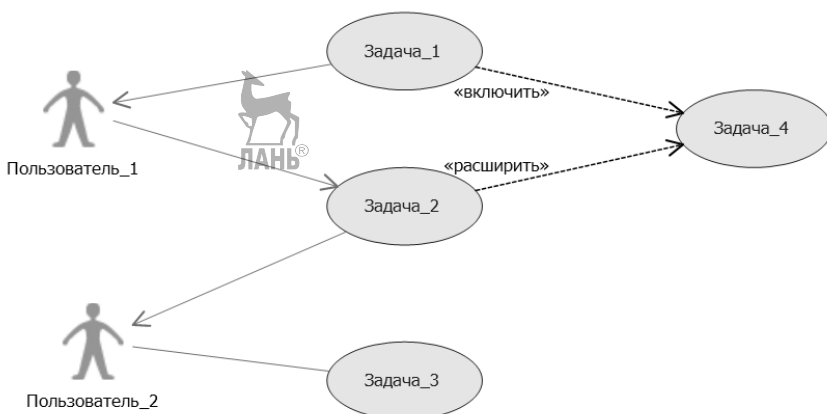


Рис. 4.7

Точно также создадим и диаграмму классов (*ClassDiagram*), как можно убедиться, в палитре инструментов теперь появились средства для создания именно диаграммы классов. Перенесем на диаграмму символ класса. В свойствах уточним его имя. Класс состоит из трех частей: имя, атрибуты и операции (именуемые иногда методы, функции). Ставим курсор мыши на обозначение класса, нажмем на правую клавишу и выберем Добавить (*Add*), далее *Attribute* или *Operation*. В окне свойств уточним их имена. Можно ставить курсор мыши и на раздел атрибутов или операции, тогда при нажатии правой клавиши предлагается добавить только их. Напомним, что при составлении диаграммы классов анализа атрибуты могут иметь содержательные названия без каких-либо ограничений, они должны задавать характеристики объектов этого класса. В качестве операции выберем обязанности, т. е. действия, которые этот класс должен предоставить при решении задач. Типы и структуры данных можно не уточнять. Установление отношения обобщения просто: выберем стрелку *Inheritance* и тянем ее от подчиненного класса к предку.

Рассмотрим отражение отношения ассоциации (точно так же устанавливаются и ее разновидности — агрегация и композиция). Вначале поступим, как описали выше при создании диаграммы вариантов использования (щелкнем на нужном отношении, затем установим курсор на начальный класс и протянем до конечного класса). Рассмотрим уточнение его характеристик. Выделим на диаграмме отношение ассоциации. В свойствах уточним его имя (поле *Name*), в нашем случае Обучение. Раскроем составные свойства: первая роль (*FirstRole*) и вторая роль (*SecondRole*). Там можем уточнить имена ролей и мощность (*Cardinality*). На стороне Университет оставим 1, но стороне Студент поставим 1..*. Наличие стрелки определим через свойство является перемещаемым (*IsNavigable*) (рис. 4.8).

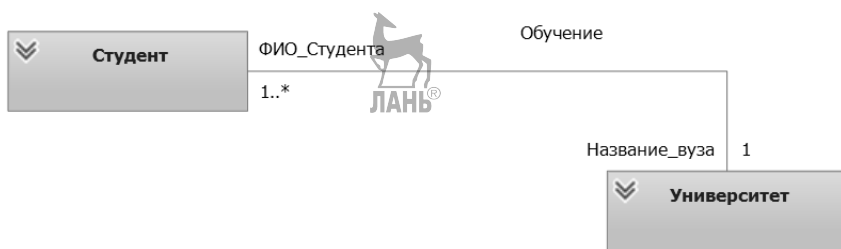


Рис. 4.8

Создадим диаграмму деятельности (*Activity*), как это делать, вы уже знаете. Перенесем на диаграмму нужные компоненты, в свойстве *Name* уточним их названия. Для их связывания поставим курсор на компонент, от которого должна идти связывающая линия, нажмем на правую клавишу мыши и выберем из контекстного меню *Add* — *Connection*, и фиксируем конец линии на нужном компоненте.

4.8. Создание диаграмм проектирования

Рассмотрим создание диаграммы классов проектирования и диаграммы последовательностей. Перед составлением диаграммы классов проектирования необходимо выбрать язык и среду реализации, чтобы при ее составлении можно было определить типы и структуры данных, а операции были оформлены по требованиям. В нашем случае используем язык C# в среде *Microsoft Visual Studio*.

Создаем вспомогательный класс *Dan*, который содержит данные о людях: фамилию (*Fam*), имя (*Imya*), рост (*Rost*), вес (*Ves*) и индекс массы тела (*MasInd*). Последний атрибут пусть будет свойство, остальные — поля. Для простых данных мы должны определить значения следующих свойств:

- имя — приведены выше;
- видимость (*public, protected, private*);
- тип данных (*int, double, bool, . . .*);
- стереотип (*field, property*);
- значение по умолчанию — задавать не обязательно.



Определим их значения путем набора с клавиатуры (имя, тип данных), выбора из выпадающего списка (видимость) или путем расставления флажков для стереотипа (поле или свойство, в принципе допустимы и оба одновременно).

Создадим еще класс *MyClass*, содержащий данные сложной структуры.

Массив *mas*, в качестве типа данных напомним *double[]*.

Список *lst1* из элементов класса *Dan*, в качестве типа данных напомним *List<Dan>*, значения остальных свойств выберем так, как было описано выше.

Коллекция общего вида *collect* из данных заданного типа. В качестве типа данных пишем *double*, поля имя, видимость, стереотип выберем как раньше, но обратим внимание на свойство кратность. По умолчанию оно равно единице. Можем дать там, в принципе, любое значение больше единицы (это — количество элементов) или писать знак *, что означает, что количество элементов не определено. Такому компоненту класса в программе соответствует стандартный интерфейс *IEnumerable*.

Приступаем к добавлению функций. Добавим в класс *Dan* функцию вычисления индекса массы тела *funMI*. Тип возвращаемого значения *double*, параметров она не имеет, стереотип метод C#. Свойства имя и видимость нам уже знакомы. У всех функций имеются свойства постусловия и предусловия, о них мы говорили выше в разделе о языке OCL. Можно заполнить и эти свойства, но они не влияют на генерацию программ и поэтому могут рассматриваться как комментарии.

Добавим в класс *MyClass* функцию *fun1*, у которой нет возвращаемого значения, но она имеет формальные параметры:

- массив *arr* типа *double*;
- *n* типа *int*, исходное значение для функции (передача по значению);
- *w* типа *double*, значение будет получено в функции;
- *t* типа *double*, значение которой должна быть задано при обращении к функции, но она может там измениться.

Добавим функцию *fun1*, определим ее свойства: имя, видимость, стереотип, возвращаемое значение, как было описано выше. Имеется свойство параметры, имеющее диалоговое окно для их определения. Откроем это окно. Видим там кнопки Добавить (*Add*) и Удалить (*Remove*). Нажмем на кнопку Добавить, в результате появится новый параметр с именем *Parameter1*. Справа имеется окно свойств параметра.

Имя *arr*. Тип данных *double[]*. Остальные значения можем оставить по умолчанию.

Имя *n*, тип данных *int*. Остальные значения можем оставить по умолчанию.

Имя *w*, тип данных *double*. Направление *out*, что означат, что его значение будет определено в функции и не должно быть определено при обращении. Как вы заметили, по умолчанию это свойство имеет значение *in*, что и требовалось в предыдущих случаях.

Имя *t* тип данных *double*. Направление *inout*, что означает, что этот параметр должен иметь значение при обращении к функции, но функция может его изменить.

Рассмотрим использование отношений между классами. Отношение обобщения означает наследование. Кроме него на диаграмме классов проектирования рекомендуется использовать только отношение агрегации (композиции) и избегать мощности «многие ко многим», как это делать — см. выше. Добавим на нашу диаграмму 3 класса для показа названных отношений. Обратите внимание на то, что у класса *ClassA* кратность отошения равна 1, а у класса *ClassB* задана *.

Результат представлен на рис. 4.9.

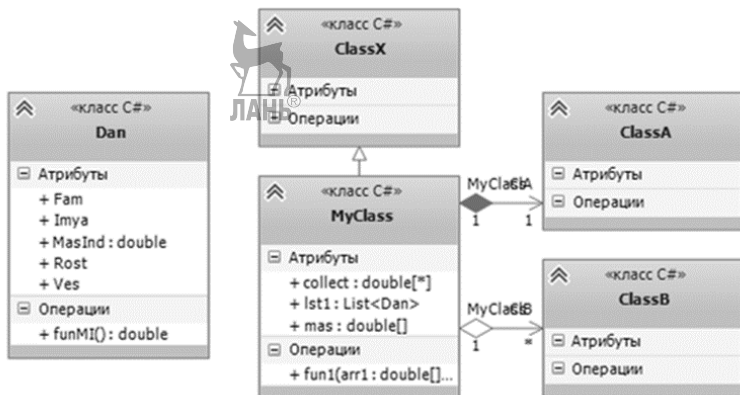


Рис. 4.9

Поставим курсор мыши на имя диаграммы *UML Class Diagram1*, нажмем на правую клавишу и выберем пункт Создать код (*Generate Code*). В обозревателе решения появится новый раздел *Mod48Lib* (*Modlib* — имя созданного нами проекта проектирования), где и появятся сгенерированные классы:

```
public class Dan
```

```

{
    public object Fam;
    public object Imya;
    public object Rost = 0;
    public object Ves;
    public virtual double MasInd
    {
        get;
        set;
    }
    public virtual double funMI()
    {
        throw new System.NotImplementedException();
    } }

//-----
public class MyClass : ClassX    //наследование
{
    public double[] mas;    // обычный массив
    public List<Dan> lst1;    // список из элементов класса Dan
    public IEnumerable<double> collect;
        //коллекция, кратность>1
    public ClassA x
    {
        get;
        set;
    }
    public virtual IEnumerable<ClassB>CIB
// свойство типа класс, кратность>1
// генерируется автоматически
    {
        get;
        set;
    }
    public virtual void fun1(double[] arr1, int n,
        out double w, ref double t)
// заголовок функции по правилам C#
    {
        throw new System.NotImplementedException();
    } }

```

При внесении изменений в диаграмму классов и повторной генерации изменения будут учтены. Однако при внесении изменений в тексты программ они не могут быть перенесены в диаграммы.

Таким образом, целесообразно создать первоначальную диаграмму классов проектирования. После генерации кода можно перейти к другой разновидности диаграммы классов, где возможности для работы с ними заметно шире. Для этого поставим курсор мыши на имя сгенерированного класса, нажмем на правую клавишу и выберем Перейти к диаграмме классов (*Show Class Diagram*). Результат показан на рис. 4.10, внесенные в диаграмму изменения автоматически передаются в текст программы и наоборот. Если на этой диаграмме классов поставить курсор мыши на символ класса и нажать на правую клавишу, то в контекстном меню предлагается добавить в класс поля, методы, свойства, конструктор, деструктор и т. д. Кроме того, там появился новый пункт «Сведения о классе» (*Class Details*) при выборе которого появится окно, приведенное на рис. 4.11. Как видно из рисунка, там подробно представлены все компоненты класса и их характеристики. Там можно их изменить, и эти изменения автоматически передаются на диаграмму классов и сами тексты сгенерированных программ. Обратите внимание на запись «Добавить ...(*Add*)»: нажав на нее можно увеличить количество всех составных частей. На самой диаграмме классов (рис. 4.10) можно также менять состав классов и их компонентов, как это было описано выше.

Рассмотрим создание диаграммы последовательностей. На диаграмме классов, приведенной на рис. 4.9 и 4.10, мы главное внимание уделили составу класса. Для рассмотрения диаграммы последовательностей нам потребуется несколько классов с функциями. Поэтому составим вспомогательную диаграмму классов, представленную на рис. 4.12. Обратите внимание, что между классами Class1 и Class2 имеется отношение наследования.

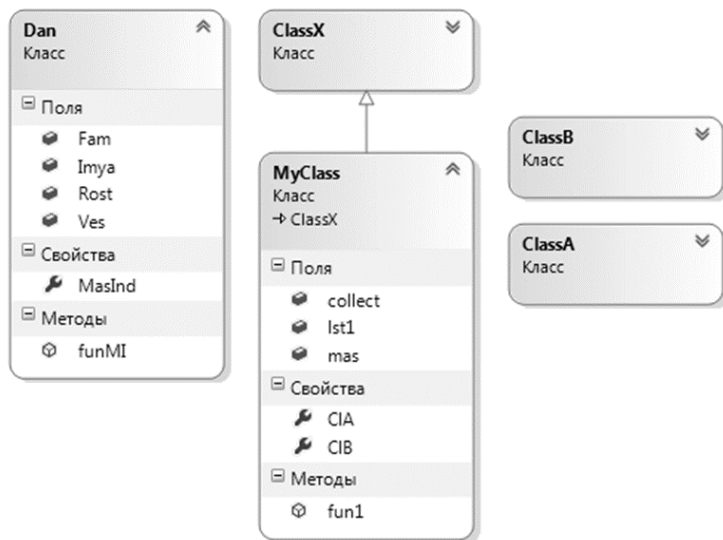


Рис. 4.10

Сведения о классах - MyClass

Имя	Тип	Модификатор	Сводка	Скрыть
Методы				
fun1	void	public		<input type="checkbox"/>
{ arr1	double[]	Her		
, n	int	Her		
, w	double	out		
, t	double	ref		
} <добавить параметр>				
<input type="checkbox"/> <добавить метод>				
Свойства				
ClaA	ClassA	public		<input type="checkbox"/>
ClaB	IEnumerable<ClassB>	public		<input type="checkbox"/>
<input type="checkbox"/> <добавить свойство>				
Поля				
collect	IEnumerable<double>	public		<input type="checkbox"/>
lst1	List<Dan>	public		<input type="checkbox"/>
mas	double[]	public		<input type="checkbox"/>
x	ClassA	public		<input type="checkbox"/>
<input type="checkbox"/> <добавить поле>				
События				
<input type="checkbox"/> <добавить событие>				

Рис. 4.11

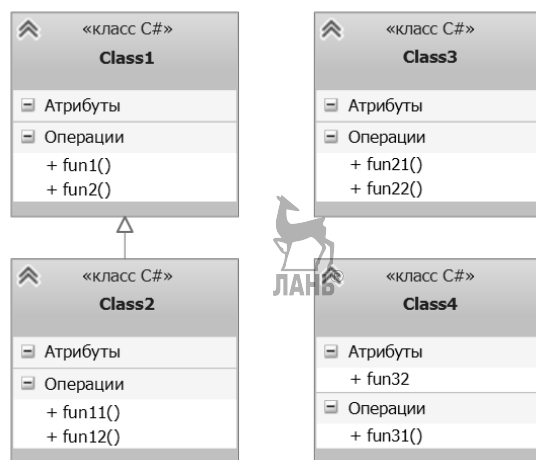


Рис. 4.12

Для создания этой диаграммы, как обычно, поставим курсор на имя проекта, нажмем на правую клавишу, выберем по очереди добавить — новый элемент — диаграмма последовательностей. Для первоначального заполнения только что созданной диаграммы имеются две возможности.

Можно поставить на диаграмме классов курсор мыши на класс и выбрать из контекстного меню пункт «Создать линию жизни», следует вопрос: на какую диаграмму линию жизни ставить, ответим на него.

Можно открыть пустую диаграмму последовательностей, из палитры инструментов перенести на нее компонент «Линия жизни» и заполнить свойства Имя и Тип. Тип можно выбрать из существующих классов.

Конечный результат не зависит от способа создания. Текст *Obj1:Class1* означает, что это объект *Obj1* класса *Class1*. Напомним, что на этой диаграмме время течет сверху вниз, и все сообщения должны быть размещены в такой последовательности. Сообщению всегда должен соответствовать метод в классе-адресате. Сообщения бывают синхронными (класс-отправитель ждет ответа от класса-адресата) и асинхронными (класс отправляет сообщение и продолжает работу). Сообщения наносим на диаграмму точно так, как мы и ранее наносили все связи: выберем нужную разновидность сообщения и протягиваем его от отправителя до адресата. Сообщениям будут присвоены имена *Message1*, ..., которые можно менять. Сообщения имеют свойство «Операция» с выпадающим списком, включающем функции класса-адресата (как собственные, так и наследованные). Диаграмма последовательностей считается корректной, если каждой линии жизни (объекту) соответствует класс и каждому сообщению функция. На нашей диаграмме сообщение *Message1* синхронное, остальные — асинхронные. Сообщению *Message1* соответствует функция *fun1()*, наследованная классом *Class2* от класса *Class1*. Составленная диаграмма представлена на рис. 4.13.

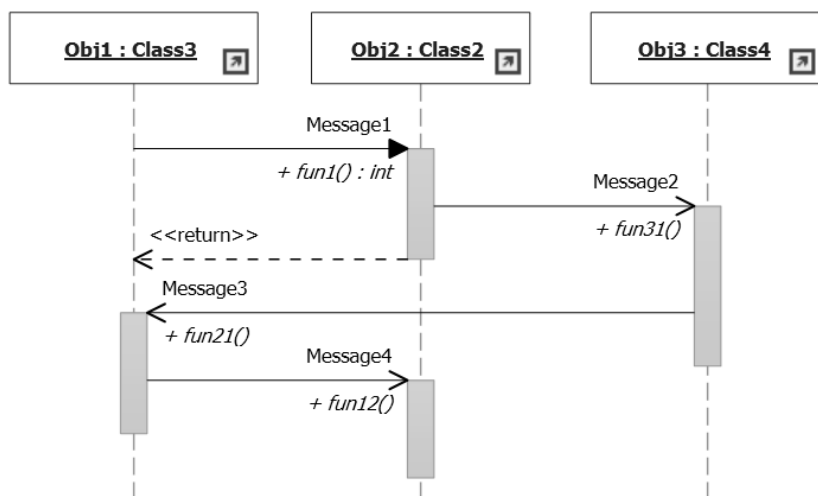


Рис. 4.13

При этом не следует забывать об одной тонкости: все объекты должны быть созданы до того, когда им можно будет передавать сообщения. По сути это означает, что должен быть запущен конструктор класса и инициирован минимальный для работы набор его данных. Кроме того, класс-отправитель должен иметь доступ к переменной типа класс-адресат. Можно рекомендовать два способа решения этой проблемы:

-
- первое сообщение запускает конструктор класса-адресата;
 - имеется «центральный» класс, из которого будут запущены конструкторы всех необходимых классов с последующей передачей полученных адресов объектов с учетом будущих передач сообщений.

Пример можно найти в разделе создания многооконных интерфейсов (п. 5.2.1 и 5.2.2): создание нового окна, определение его свойств и вызов методов.



5. Реализация на языке C# в среде *Microsoft Visual Studio*

5.1. Простейший пример

5.1.1. Постановка задачи

Визуальное программирование похоже на рассмотренное выше составление диаграмм. «Перетаскиваем» с палитры инструментов на создаваемую форму компоненты, уточним их свойства и определим связанные с ними события. Создаем в *Microsoft Visual Studio* проект *Windows Forms* для C#. Поставим перед собой цель — создать простейший пример, главная форма которого приведена на рис. 5.1.

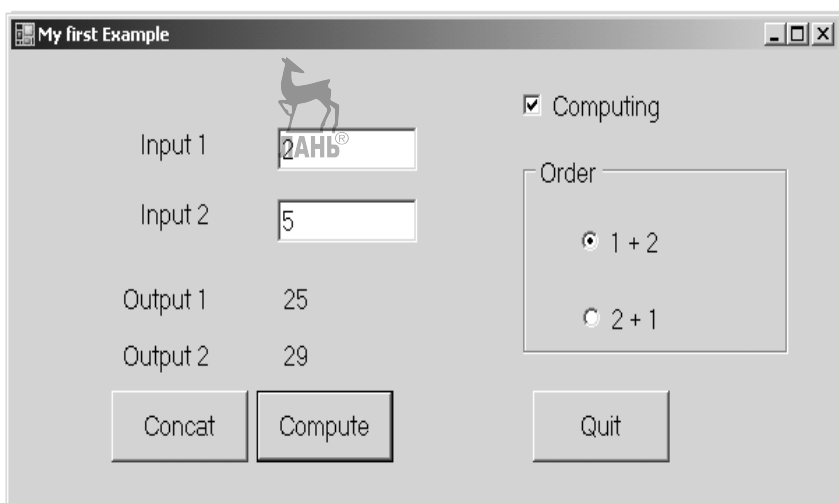


Рис. 5.1

Для начала поменяем свойство формы *Text* на *My first Example* и увидим, что изменился заголовок формы. Ограничимся рассмотрением компонентов *Label*, *TextBox* и *Button*. *Label* (метка) предназначена для нанесения на форму пояснительных текстов и вывода результатов. Во время работы приложения невозможно редактировать содержимое меток. *TextBox* (строка редактирования) предназначена для ввода/вывода, тип данных в нем — всегда *string*, и все преобразования должны выполняться программистом. Компоненту *Button* (командная кнопка) можно ставить в соответствие функцию, которая будет выполнена при нажатии на кнопку. Все названные компоненты имеют среди других свойство *Name* — имя, по которому можно в программе ссылаться на него, и *Text*, которое задает изображаемый на экране текст. Какие значения присвое-

ны свойствам *Text*, видно из рисунка, значения свойства *Name* по умолчанию видны из приведенных текстов функций.

Реализации командных кнопок:

```
private void button3_Click(object sender, EventArgs e)
{ //    завершение работы приложения    Quit
    Close();
}
private void button1_Click(object sender, EventArgs e)
{ //    сцепление введенных строк    Concat
    label5.Text = textBox1.Text + textBox2.Text;
}
private void button2_Click(object sender, EventArgs e)
{ // Compute вычисления, нужны явные преобразования типов
    int i, j, k;
    i = Convert.ToInt32(textBox1.Text);
    j = Convert.ToInt32(textBox2.Text);
    k = 2 * i + 5 * j;
    label6.Text = Convert.ToString(k); // 1
}
```

Примечание. Здесь и в дальнейшем: жирным шрифтом показано то, что вставляет программист; обычным то, что вставляет среда.

Вопрос читателю: можно ли вместо строки // 1 писать *label6.Text = "" + k;* Почему?

5.1.2. Средства управления работой программы

Добавим на нашу форму кнопку выбора (*CheckBox*) и две радиокнопки (*RadioButton*). Все кнопки выбора независимы друг от друга. Радиокнопки обычно объединяют в радиогруппы, и из каждой группы в любой момент времени может быть выбрана одна и только одна радиокнопка.

Пусть наша единственная кнопка выбора имеет имя *cB1* и показанное на рис. 5.1 значение свойства *Text*. Назначение этой кнопки: вычисления возможны лишь в случае, если она выбрана (свойство *Checked* имеет значение *True*).

Для создания радиогруппы необходимо сначала занести на форму компонент рамка — *GroupBox* (находится среди элементов управления *Containers*) и лишь после этого на него — требуемое количество (в нашем случае 2) радиокнопок. Пусть свойство *Text* для *GroupBox* имеет значение *Order*, свойство *Name* сохраняет значение по умолчанию. Для двух радиокнопок дадим свойству *Name* значения *rB1* и *rB2* соответственно. Значения их свойств *Text* видны на рис. 5.1. В нашем случае от выбора радиокнопки зависит очередность соединения введенных строк.

Поменяем реализацию двух командных кнопок:

```
private void button1_Click(object sender, EventArgs e)
{ // сцепление введенных строк Concat
    if (rB1.Checked)
        label5.Text = textBox1.Text + textBox2.Text;
    if(rB2.Checked)
        label5.Text = textBox2.Text + textBox1.Text;
}

private void button2_Click(object sender, EventArgs e)
{ // Compute вычисления
    int i, j, k;
    if (cB1.Checked)
    {
        i = Convert.ToInt32(textBox1.Text);
        j = Convert.ToInt32(textBox2.Text);
        k = 2 * i + 5 * j;
        label6.Text = Convert.ToString(k);
        // label6.Text = "" + k;
    }
    else
        MessageBox.Show("Режим вычислений не установлен ");
    // вывод сообщения пользователю
}
```

5.1.3. Создание меню

Для создания меню перенесем на форму компонент *MenuStrip*, который находится в группе компонентов *Menus & Toolbars*. Местоположение его на форме значения не имеет, поскольку меню всегда находится на одном и том же месте. После этого, двигаясь с помощью мыши по пунктам меню, набираем на клавиатуре их названия, которые автоматически становятся значениями свойств *Text*. Значения свойств *Name* выбираются по умолчанию, например, пункт меню *File* получит имя *fileToolStripMenuItem*. Если не предполагается в дальнейшем ссылка в функциях на пункты меню, то имена, естественно, можно такими и оставить. В противном случае желательно их поменять на нечто покороче. Например, делать их равными свойству *Text*.

Для внесения исправлений в уже созданное меню поставим курсор на тот пункт (неважно, горизонтального или выпадающего меню), перед которым необходимо вставить новый пункт, нажмем на правую клавишу и выберем пункт *Insert*. Для удаления поставим курсор на удаляемый пункт и выберем из выпа-

дающего меню пункт *Delete*. Для изменения свойств выделяем пункт меню и пользуемся традиционным способом — списком свойств.

Для реализации пунктов меню ставим на него курсор, делаем двойной щелчок и в полученную заготовку функции пишем необходимую программу.

5.1.4. Ввод/вывод массивов

Рассмотрим ввод/вывод массивов на примере формы рис. 5.2 (одномерный массив может быть представлен как в виде строки, так и в виде столбца). В обоих случаях используется знакомый нам компонент *TextBox*, но для представления массива столбцом необходимо свойству *Multiline* придать значение *True*. После этого можно «вытянуть» этот компонент по вертикали. Для представления двумерного массива используем этот же компонент, *Multiline* имеет значение *True*, а компоненту дадим необходимые для размещения массива высоту и ширину. Дадим компонентам следующие имена (свойство *Name*): *mas_row* для массива-строки, *mas_col* — для массива-столбца, *mas22* — для двумерного массива, *result* — для результата. Тип данных в компонентах *TextBox* по-прежнему *string*.

The screenshot shows a Windows form titled "Form1" with a light gray background. At the top left, there is a text box containing the string "12;23;45;96". Below it, on the left side, is a vertical list box containing the values "-8", "96", "-8", and "-3". In the center of the form, there are two buttons stacked vertically: "Ввод двумерного массива" (Input 2D array) and "Обработка и вывод двумерного массива" (Processing and output of 2D array). Below these buttons are two more buttons: "Массив - столбец" (Array - column) and "Массив - строка" (Array - row). To the right of the input text box, there is a larger text box containing a 3x3 matrix of numbers: "-156 192 -126", "-128 -190 150", and "30 912 -12650". Below this matrix, the text "Результат" (Result) is displayed. At the bottom right, there is a text box containing "Summa 2: -11966" and a button labeled "Выход" (Exit).



Рис. 5.2

Назначение кнопок видно из названий.

Ввод/вывод и обработка одномерного массива-строки

Для ввода одномерного массива-строки необходимо определить, какой разделитель будет использован между отдельными элементами. В нашем случае используем знак ;. Реализация кнопки «Массив-строка» приведена ниже:

```

private void button5_Click(object sender, EventArgs e)
{
    string[] temp;
    //массив для размещения вводимых и выводимых данных
    int k, pr = 1;
    string dan;
    dan = mas_row.Text;
    //присвоение набранной строки целиком переменной
    temp = dan.Split(';');
    // выделение отдельных элементов массива,
    // аргументом функции Split является наш разделитель ;
    k = temp.Length; //определим длину введенного массива
    while (temp[k - 1] == "") k--; // см. разъяснение 1
    int[] x = new int[k];
    //объявление массива чисел для введенных данных
    for (int i = 0; i < k; i++)
    //преобразование типов введенных данных
        x[i] = Convert.ToInt32(temp[i]);
    for (int i = 0; i < k; i++)
    { // обработка введенного массива
        x[i] = x[i] * x[i];
        pr *= x[i];
    }

    result.Text = "Proizv " + pr; // вывод результата
    dan = "";
    mas_row.Clear(); // очистка поля mas_row для вывода
    for (int i = 0; i < k; i++)
        dan += " " + x[i];
    /* Накопление полученных элементов числового массива в
       переменной символьного типа, в качестве разделителя -
       пробел */
    mas_row.Text = dan; //вывод результата
}

```

Разъяснение 1. Если закончить последовательность вводимых данных разделителем (в нашем случае ;), то последним элементом массива `temp` будет пустая строка, которая не может быть преобразована в число. Поэтому исключим эту возможность. Если при вводе поставить два разделителя подряд, то в массиве `temp` будет пустая строка в середине. Что с ней делать — решать программисту!

Ввод/вывод и обработка одномерного массива-столбца

При вводе одномерного массива в виде столбца необходимо просто набирать значения, нажав после каждого на Enter. Нажатие на Enter после последнего значения вставит пустую строку в качестве последнего элемента. Реализация кнопки «Массив-столбец»:

```
private void button4_Click(object sender, EventArgs e)
{
    string []st1;
    int []mas;//массивы для размещения данных
    int k, pr=0;
    st1=new string[mass_col.Lines.Length];
    k=mass_col.Lines.Length;
    mas = new int[k];
    /* Иницилируем массивы для принятия исходных данных,
    количество элементов равно количеству строк
    в компоненте mass_col */
    st1 = mass_col.Lines;
    /* Перенесем введенный массив из компонента mass_col
    в массив st1 */
    while(st1[k-1]=="") k--;//см. разъяснение 1 выше
    for(int i=0;i<k;i++)// преобразование типов данных
        mas[i]=Convert.ToInt32(st1[i]);
        for (int i = 0; i < k; i++)
            {// обработка массива
                mas[i] = 5 * mas[i];
                pr += mas[i];
                st1[i] = Convert.ToString(mas[i]);
            }
    mas_col.Clear();//очистка поля вывода
    mass_col.Lines = st1;// вывод результата: массива-столбца
    result.Text="Summa Col "+pr;
}
```

Ввод/вывод и обработка двумерного массива

Перед тем как приступить к вводу/выводу и обработке двумерного массива, введем некоторые уточнения. Очевидно, что на кнопку «Обработка и вывод двумерного массива» нет смысла нажимать, пока массив не введен. Поэтому изначально пусть эта кнопка будет «серой», для чего на этапе проектирования дадим ее свойству *Enabled* значение *False*. Во-вторых, ввод и обработка двумерного массива будут в разных функциях, поэтому его необходимо объявить

как переменную класса *Form1* (напомним, что каждой форме соответствует класс). Найдем раздел переменных этого класса и запишем туда:

```
private double[,] a;
```

```
int n, m;//количество строк и столбцов
```

Мы по существу уже знаем, как ввести двумерный массив: он представляет собой столбец с одномерными массивами-строками. Используем в качестве разделителя ; . Предположим, что двумерный массив прямоугольный.

Реализация кнопки «Ввод двумерного массива»:

```
private void button1_Click(object sender, EventArgs e)
```

```
{
```

```
    string[] mas;
```

```
    // массив для представления одной вводимой строки
```

```
    string s = "";
```

```
    string[] dan;
```

```
    //массив для представления всех введенных строк
```

```
this.button2.Enabled = true;// кнопка обработки откроется
```

```
n = mas22.Lines.Length;/* количество строк в двумерном массиве будет равно количеству введенных строк */
```

```
    dan = new string[n];
```

```
    dan = mas22.Lines;//введенные строки передаем в
```

```
    // массив символьных строк, одна строка - один элемент
```

```
    while (dan[n - 1] == "") n--;
```

```
    mas = dan[0].Split(';'); //Разделим первую строку на  
    //элементы с помощью разделителя
```

```
    m = mas.Length;
```

```
    //определим количество столбцов двумерного массива
```

```
a = new double[n, m];// инициализация двумерного массива
```

```
    for (inti = 0; i<n; i++)
```

```
{ // строка за строкой преобразуем типы данных и  
    // формируем числовой массив
```

```
        mas = dan[i].Split(';');
```

```
        for (int j = 0; j < mas.Length; j++)
```

```
            a[i, j] = Convert.ToDouble(mas[j]);
```

```
}
```

```
}
```

Реализация кнопки «Обработка и вывод двумерного массива»:

```
private void button2_Click(object sender, EventArgs e)
```

```
{
```

```
    string[] st1 = new string[n];// массив для
```

```
    // накопления двумерного массива после обработки
```

```
    double smm=0;
```

```
    mas22.Clear();//очистка поля для вывода
```

```

    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
        { // обработка массива
            smm += a[i, j];
            a[i, j] = 2 * a[i, j];
        }
    for (int i = 0; i < n; i++)
// цикл подготовки двумерного массива к выводу
        for (int j = 0; j < m; j++)
            st1[i] += " " + Convert.ToString(a[i, j]);
    mas22.Lines = st1; // вывод двумерного массива
    result.Text = "Summa 2: " + Convert.ToString(smm);
}

```

Примечание. На рис. 5.2 форма показана после нажатия кнопок «Ввод двумерного массива» и «Обработка и вывод двумерного массива».

5.1.5. Форматированный ввод/вывод двумерного массива

Приведенные выше примеры ввода массивов легки в реализации, но требуют от пользователя большой аккуратности при наборе данных. Имеется и возможность форматированного ввода/вывода массивов. Ограничимся рассмотрением двумерного массива, упростить приводимый пример для одномерного случая не составляет труда.

Рассмотрим две разновидности обработки форматированного двумерного массива. Первый вариант представлен на рис. 5.3.

The screenshot shows a Windows application window titled "Form1". Inside the window, there are two labels with text boxes: "Количество строк" with the value "3" and "Количество столбцов" with the value "4". To the right of these is a button labeled "Менять". Below these is a large, empty rectangular area, likely a data grid. At the bottom, there are three labels: "Сумма: 0", "Класс: 0", and "Строка: 0". To the right of each of these labels is a button: "Сумма" and "Класс".

Рис. 5.3

Компоненты *Label*, *textBox*, *Button* нам уже знакомы. В середины формы находится компонент *dataGridView* из палитры *Data*, на котором и будет представлен двумерный массив. В принципе существуют две возможности обработки двумерного массива: считывать его элементы прямо из формы или создать

обычный двумерный массив, записать туда элементы с формы и обрабатывать его. Последняя возможность предпочтительнее, если массив должен обрабатываться многократно, так можно избежать многократного преобразования данных. Нижнюю часть формы (метка и кнопка Класс, строку редактирования строка) рассмотрим ниже, в п. 5.6.

Реализация:

До начала работы программы необходимо задать количество строк и столбцов. Считаем, что начальные их значения даны при проектировании (в нашем случае 3 и 4). Во время работы программы их можно менять. Начальные значения дадим в конструкторе формы.

```
public partial class Form1 : Form
{
int n, m;//n-количество строк, m-количество столбцов
public Form1()
{
InitializeComponent();
// определим количество строк и столбцов
n = Convert.ToInt32(RowCnt.Text);
m = Convert.ToInt32(ColCnt.Text);
dataGridView1.RowCount = n;
dataGridView1.ColumnCount = m;
/* dataGridView1 имя поля представления двумерного массива
RowCount количество строк
ColumnCount количество столбцов */
}

private void Change_Click(object sender, EventArgs e)
{
// изменение количества строк и столбцов,
// считаем, что их минимальное значение равно 2
n = Convert.ToInt32(RowCnt.Text);
m = Convert.ToInt32(ColCnt.Text);
if (n < 2) n = 2;
if (m < 2) m = 2;
dataGridView1.RowCount = n;
dataGridView1.ColumnCount = m;
}

private void Sum2_Click(object sender, EventArgs e)
{
double x,sum = 0;
for(int i=0;i<dataGridView1.Rows.Count;i++)
for(int j=0;j<dataGridView1.Columns.Count;j++)
{
```

```

x = Convert.ToDouble(dataGridView1.Rows[i].Cells[j].Value);
// чтение значений из формы
    sum += x;
    x *= 2.5; // изменения значений
    dataGridView1.Rows[i].Cells[j].Value =
        x.ToString("F1");

// вывод значений на форму
}

Sum1.Text = sum.ToString("F2");

}

```

Во втором варианте обработки двумерного массива используем для внутреннего представления массива стандартный класс *DataTable*. Это класс в первую очередь предназначен для работы с таблицами баз данных, поэтому в нем столбцы могут иметь разный тип данных. В этом примере у нас все столбцы будут иметь одинаковый тип данных, поэтому мы можем их задать в цикле. Внешний вид представлен на рис. 5.4, верхняя строка представляет номера столбцов, первый столбец предназначен для номеров строк.

Занесем на форму компонент *dataGridView* из группы *Data* и дадим ему имя *Gr1*. Этот компонент служит для представления таблицы (по сути двумерного массива) на форме. Для внутреннего представления двумерного массива используем стандартный класс *DataTable* (объект *Tabel*).

В состав класса *Form1* добавим функцию *InitGrid* с двумя параметрами: количество строк и количество столбцов. Эта функция определяет свойства создаваемой таблицы и связывает ее с представлением на форме. Для начальной инициализации таблицы включим ее вызов в конструктор класса *Form1*.

The screenshot shows a Windows application window titled "Form1". On the left, there are two input fields: "Строк" (Rows) with the value "3" and "Столбцов" (Columns) with the value "2". To the right of these is a data grid with 4 rows and 3 columns. The first row has headers "First", "Col 1", and "Col 2". The subsequent rows contain numerical values: (0, 2, 0), (1, 0, 3), (2, 4, 0), and a row starting with an asterisk (*). Below the grid, there are four buttons: "Сумма" (Sum), "Преобразование массива" (Transform array), "Изменение размеров" (Change size), and "Выход" (Exit). To the right of these buttons, the text "Результат" (Result) is displayed with the value "9".

	First	Col 1	Col 2
0	2	0	
1	0	3	
2	4	0	
*			

Рис. 5.4

```

namespace WA1
{

    public partial class Form1 : Form
    {
        DataTable Tabel;
        void InitGrid(int n, int m)
        {
            Tabel = new DataTable();
             DataColumn x2 = Tabel.Columns.Add("First ",
                                                typeof(Int32));
            // Создание первого столбца, параметры: заглавие и тип
            // данных Первый столбец содержит имена строк
             DataColumn x1;
            for (int i = 0; i < m; i++)
            {
                //создание остальных столбцов - для представления данных
                x1 = Tabel.Columns.Add("Col " + (i + 1), typeof(Int32));
                x1.DefaultValue = 0; //Значение по умолчанию
            }
            for (int i = 0; i < n; i++)
            { //создание строк
                DataRow y = Tabel.NewRow();
                Tabel.Rows.Add(y);
                Tabel.Rows[i][0] = i+1;
            }
            x2.ReadOnly = true; // первый столбец для номеров, туда
                                // нельзя ввести данные
            Gr1.DataSource = Tabel;
            // связывает внутреннее и внешнее представления данных
        }

        public Form1()
        {
            InitializeComponent();
            InitGrid(3, 4);
            //создание начальной таблицы
        }

        private void In1_Click(object sender, EventArgs e)
        { // изменение количества строк, столбцов
            int k,n;

```

```

        k = Convert.ToInt32(textBox1.Text);
//Количество столбцов
        n = Convert.ToInt32(textBox2.Text);
//    Количество строк
        InitGrid(k, n);

}

private void First_Click(object sender, EventArgs e)
{
    //обработка данных из таблицы
    int sum = 0;
    for (int i = 0; i < Tabel.Rows.Count ; i++)
        for (int j = 1; j < Tabel.Columns.Count; j++)
            sum += Convert.ToInt32(Tabel.Rows[i][j]);
    result.Text=Convert.ToString(sum);

}

private void Second_Click(object sender, EventArgs e)
{ // изменение данных в таблице
    int y,z = 3;
    for (int i = 0; i < Tabel.Rows.Count ; i++)
        for (int j = 1; j < Tabel.Columns.Count; j++)
        {
            y = Convert.ToInt32(Tabel.Rows[i][j]);
            y += z;
            Tabel.Rows[i][j] = Convert.ToString(y);
        }

}

```

Примечания.

1. Нумерация строк и столбцов начинается с нуля. Нулевой столбец мы используем для номеров строк, поэтому при обработке минимальное значение номеров столбцов равно единице, а номеров строк — нулю. Первая строка считается стандартной, и она в нумерацию не входит.
2. На рис. 5.4 форма показана после нажатия кнопки «Сумма».

5.2. Создание многооконных приложений

Microsoft Visual Studio позволяет создавать две разновидности многооконных приложений: *SDI*- и *MDI*-приложения. *SDI*-приложения состоят из нескольких независимых форм (окон). По умолчанию будет создано *SDI*-приложение. В *MDI*-приложении имеется одна главная форма, остальные формы находятся в пределах главной; из главной формы можно управлять подчиненными формами. Единственное меню *MDI*-приложения находится в главном окне.

Перед созданием многооконного приложения его необходимо проектировать: продумать вопрос о том, какие окна нужны и что на них будет отображено. Форма — это разновидность класса. Экземпляры классов, как известно, необходимо создавать. Это правило распространяется и на формы: автоматически создается лишь одна форма — главная. Создание всех остальных форм лежит на программисте. Закрытие формы функцией *Close()*; или нажатием на кнопку **X** вызывает уничтожение формы, и в случае необходимости она должна быть создана заново.

В принципе, любую задачу можно решить как с помощью *SDI*-приложения, так и с помощью *MDI*-приложения. Пожалуй, создание *SDI*-приложения проще. *MDI*-приложение можно рекомендовать при необходимости создать и работать одновременно с несколькими одинаковыми формами. *Microsoft Visual Studio* является *SDI*-приложением.

5.2.1. Создание *SDI*-приложения

Пусть наше приложение содержит следующие формы:

- главная форма с меню;
- форма для определения режимов работы программы;
- форма «О программе» (*About*);
- форма для ввода исходных данных и вывода результата.

В момент запуска приложения на экране появляются первые две формы, остальные появляются при выборе соответствующего пункта меню.

Пусть меню имеет следующую структуру:

<i>File</i>	<i>Windows</i>	<i>Help</i>
<i>Quit</i>	<i>New</i>	<i>About</i>
<i>Open Dialog</i>		

Главная форма содержит только меню, и поэтому она здесь не приводится. Форма для определения режимов работы (*Form2*) программы приведена на рис. 5.5.

Две радиокнопки имеют имена *rB1* и *rB2*; две кнопки выбора — имена *cB1* и *cB2*; строка редактирования *textBox1*. Реализация единственной командной кнопки: *Hide()*; (убрать с экрана, но не уничтожить, в таком случае ее можно позже только открыть, нет необходимости ее заново создавать).

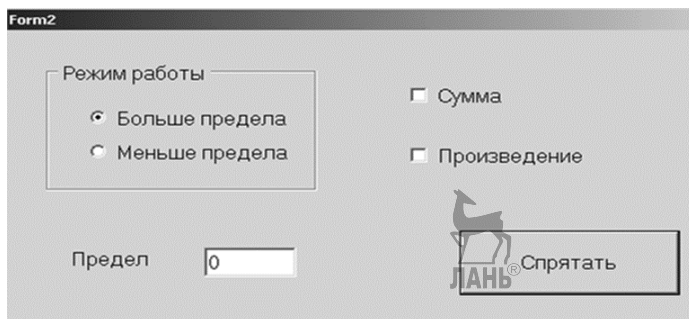


Рис. 5.5

Занесение на форму перечисленных компонентов ничем не отличается от рассмотренного выше, и мы на этом останавливаться не будем. В связи с тем, что эта форма участвует в многооконном приложении, требуется выполнить дополнительные действия.

1. Во избежание случайного закрытия этой формы уберем с нее кнопку **X**. Для этого дадим ее свойству *ControlBox* значение *False*. В нашем приложении мы предусмотрим **только** открытие этой формы во время работы приложения. Ее создание выполняется автоматически при запуске. Поэтому при случайном ее закрытии (с уничтожением) пользователь лишен возможности ее заново создать.
2. Необходимо обеспечить доступ к радиокнопкам, кнопкам выбора и строке редактирования извне. По умолчанию они имеют атрибут доступа *private* и являются переменными своего класса (в нашем случае *Form2*). Для этого их свойствам *Modifiers* дадим значение *public*.
3. Желаемое местоположение формы обеспечивается изменением значений двух ее свойств: для *StartPosition* выберем значение *Manual* и для *Location* задаем подходящие значения координат *X* и *Y* верхнего левого угла. Таким образом, можем подбирать местоположение любой формы, в том числе и главной.
4. Для обеспечения создания и открытия формы при запуске приложения в класс главной формы внесем следующие дополнения:
 - Объявление *Form2 f2*; в раздел переменных.
 - В конструктор *Form1* добавить

```
public Form1()
{
    InitializeComponent();
    f2 = new Form2();
    //запуск конструктора класса
    f2.Show();
    //Передача сообщения объекту f2
}
```

5. Реализация пункта меню *Open Dialog*: **f2.Show();**

Для создания формы «**О программе**» используем заготовку: поставим курсор мыши на имя нашего приложения, нажмем правую клавишу, из выпадающего меню выберем *Add — New Item*, затем из списка *About Box* и нажмем на кнопку *Add*. На экране появится заготовка формы «О программе». На ней имеются компоненты типа *Label*, поэтому внесение изменений в заготовку не вызывает трудностей. Реализация единственной кнопки этой формы *Close()*;

Реализация пункта меню *About*:

```
private void aboutToolStripMenuItem_Click(object sender, EventArgs e)
{
    AboutBox1 ab1;
    ab1 = new AboutBox1();
    ab1.ShowDialog();
}
```

Возникает вопрос: какая разница между двумя способами открытия форм *ShowDialog()* и *Show()*? При открытии формы через *ShowDialog()* заблокируется доступ к другим формам, пока эта форма не закрыта. При открытии через *Show()* на экране компьютера появится еще одна форма, но между открытыми формами можно переключаться.

Внешний вид формы (*Form3*) для ввода исходных данных и вывода результата показан на рис. 5.6. Для ввода одномерного массива используем строку. При работе с этой формой необходимо сослаться на переменные диалоговой формы (*Form2*). Но они являются переменными другого класса, поэтому ссылка на них возможна только через указатель на экземпляр соответствующего класса. Для обеспечения такой ссылки включим в число переменных *Form2*:

```
public Form2 f22;
// Реализация пункта меню New:
private void newToolStripMenuItem_Click(object sender,EventArgs e)
{
    Form3 f3;
    f3 = new Form3();
    f3.f22 = f2;
    //для обеспечения ссылки на диалоговое окно
    f3.Show();
}
```

Рис. 5.6

Реализация командной кнопки «Вычислить» (в зависимости от состояния кнопок выбора вычисляют или нет сумму/произведение; в зависимости от состояний радиогруппы обрабатывают числа больше/меньше заданного в диалоговом окне значения):

```
private void button2_Click(object sender, EventArgs e)
{
    string[] St1Arr;
    string r1;
```

```

double []x;
double z,sum = 0,pr = 1;
r1 = textBox1.Text;
St1Arr = r1.Split(';');
x = new double[St1Arr.Length];
z=Convert.ToDouble(f22.textBox1.Text);
for (int i = 0; i < x.Length; i++)
    x[i] = Convert.ToDouble(St1Arr[i]);
if (f22.cb1.Checked)
//найдем сумму, используем ссылку на диалоговую форму
{
    for (int i = 0; i < x.Length; i++)
    {
        if (f22.rb1.Checked && x[i] >= z) sum += x[i];
        if (f22.rb2.Checked && x[i] < z) sum += x[i];
    }
}
if (f22.cb2.Checked)//найдем произведение
{
    for (int i = 0; i < x.Length; i++)
    {
        if (f22.rb1.Checked && x[i] >= z) pr*= x[i];
        if (f22.rb2.Checked && x[i] < z) pr*= x[i];
    }
}
label3.Text = "" + sum;
label5.Text = "" + pr;
}

```

5.2.2. Создание MDI-приложения

В MDI-приложении существует одна главная форма и множество подчиненных. Чтобы окно *Form1* стала главным, поменяем значение его свойства *IsMdiContainer* на *True*. Разрешается и создание окон, которые открываются из главного окна, но не являются подчиненными. Но такими окнами нельзя управлять из главного окна. В нашем приложении будут:

- главное окно с меню;
- окно «О программе»;
- динамически создаваемое диалоговое окно;
- окно, которое можно создавать многократно из главного;
- окно, которое можно создать из главного в единственном экземпляре.

Убедительно рекомендуем: не размещайте на главном окне ничего, кроме меню! Пусть меню имеет следующую структуру:

Файл	Окно	Диалог	Помощь
Выход	Создать	Открыть	О программе
	Каскад		
	Мозаика горизонтально		
	Мозаика вертикально		
	Заккрыть текущее		
	Заккрыть все		
	Единственная форма		

Создание окна «О программе» ничем не отличается от рассмотренного выше, и мы на этом останавливаться не будем.

Переходим к рассмотрению диалогового окна, которое может создаваться и уничтожаться по ходу работы с программой. (Диалоговое окно, которое существует всегда и которое может лишь появиться на экране и исчезнуть с него, было рассмотрено выше и таким же образом можно с ним работать и в MDI-приложении.) Очевидно, что если диалоговое окно уничтожается, то все его компоненты потеряют свои значения, и, чтобы этого не случилось, следует их где-то хранить. Мы используем для этого переменные в классе главной формы. Внешний вид диалогового окна приведен на рис. 5.7.

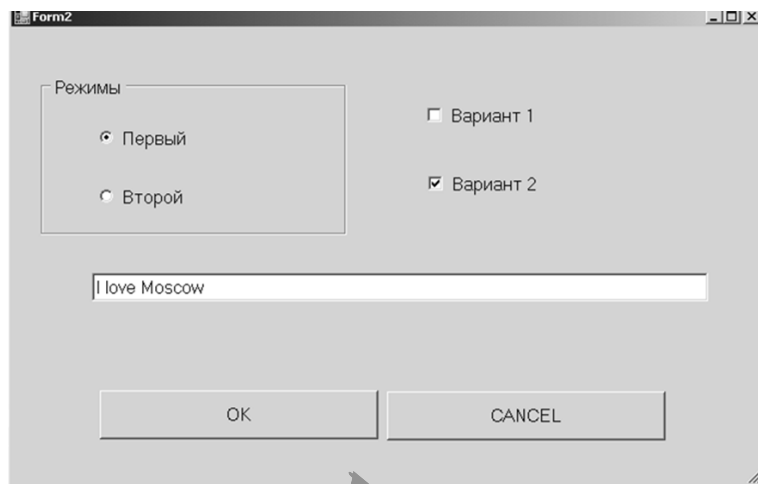


Рис. 5.7

Нанесенные на него компоненты нам знакомы, они имеют имена $rB1$, $rB2$, $cB1$, $cB2$, $tB1$. У всех атрибут доступа (свойство *Modifiers*) изменен на *public*. Кнопки *OK* и *CANCEL* по традиции позволяют закрыть окно с (без) сохранением внесенных изменений. Нам предстоит решить следующие задачи:

- открытие диалогового окна с присвоением текущих значений его компонентам;
- работа с диалоговым окном;

- закрытие окна с сохранением, в случае необходимости, внесенных изменений.

Для хранения внесенных в диалоговое окно значений, в то время когда само окно не существует, объявим в классе главного окна (у нас *Form1*) следующие переменные:

```
static public int rgi;
```

```
//для хранения номера выбранной радиокнопки
```

```
static public boolean b1,b2; //для кнопок выбора
```

```
static public string s1; // для строки редактирования
```

Атрибут доступа *public* необходим, чтобы можно из диалогового окна к ним обращаться. Атрибут *static* позволяет ссылаться на эти переменные через имя класса.

Для присвоения начальных значений этим переменным в момент запуска приложения свяжем с событием главного окна *Load* (возникает в момент загрузки формы в момент запуска приложения) следующую функцию:

```
private void Form1_Load(object sender, EventArgs e)
```

```
{  
    rgi = 0;  
    b1 = false;  
    b2 = true;  
    s1 = "I love Moscow"; }
```

Открытие этого окна (пункт меню Диалог — Открыть):

```
private void открытьToolStripMenuItem_Click(object sender, EventArgs e)
```

```
{  
    Form2 f2;  
    f2 = new Form2();  
    switch(rgi)  
    { case 0:  
        f2.rB1.Checked=true;  
        break;  
        case 1:  
        f2.rB2.Checked=true;  
        break;  
    }  
    f2.cB1.Checked = b1;  
    f2.cB2.Checked = b2;  
    f2.tB1.Text = s1;  
    f2.ShowDialog();  
}
```

Работа с окном ничего нового для нас не содержит, и мы на этом останавливаться не будем.

Закрытие окна кнопкой *CANCEL*: *Close()*;

Закрытие окна кнопкой *OK*: сохранение значений и закрытие.

```
private void button1_Click(object sender, EventArgs e)
{
    Form1.s1 = tB1.Text;
    Form1.b1 = cB1.Checked;
    Form1.b2 = cB2.Checked;
    if (rB1.Checked) Form1.rgi = 0;
    else Form1.rgi = 1;
    Close(); }
```

Множественно создаваемое окно представлено на рис. 5.8.

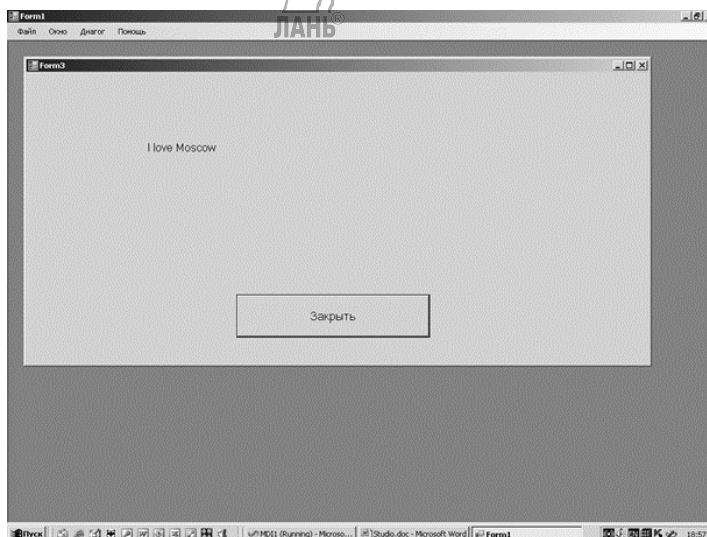


Рис. 5.8

Создание окна (создавать можно много подобных окон):

```
private void создатьToolStripMenuItem_Click(object sender, EventArgs e)
{
    Form3 f3;
    f3 = new Form3();
    f3.MdiParent = this; // задаем, что созданное окно
    // является подчиненным для окна, из которого оно создано
    f3.label1.Text = s1;
    f3.Show();}
```

Примечание. Использовать для показа подчиненных окон функцию *ShowDialog()* нельзя!

Управление подчиненными окнами осуществляется стандартными средствами.

```
private void каскадToolStripMenuItem_Click(object sender, EventArgs e)
{
this.LayoutMdi(MdiLayout.Cascade); //располагать каскадом
}
```

```
private void
мозаикаГоризонтальноToolStripMenuItem_Click(object sender, EventArgs e)
{ // горизонтальная мозаика
this.LayoutMdi(MdiLayout.TileHorizontal); }
```

```
private void мозаикаВертикальноToolStripMenuItem_Click(object sender,
EventArgs e)
{ // вертикальная мозаика
this.LayoutMdi(MdiLayout.TileVertical); }
```

```
private void закрытьТекущееToolStripMenuItem_Click(object sender,
EventArgs e)
{ // закрытие текущего (активного) окна с проверкой,
// существует ли такое
if (this.ActiveMdiChild != null) ActiveMdiChild.Close(); }
```

```
private void закрытьВсеToolStripMenuItem_Click(object sender, EventArgs e)
{ // два варианта закрытия всех подчиненных окон
/* for (int i = MdiChildren.Length - 1; i >= 0; i--)
    MdiChildren[i].Close();*/
while (this.ActiveMdiChild != null) ActiveMdiChild.Close();
}
```

ActiveMdiChild — системная переменная, указатель на активное в данный момент окно.

MdiChildren — массив указателей на подчиненные окна.

Может возникнуть необходимость создания подчиненного окна в единственном экземпляре. Для этого необходимо после его создания заблокировать пункт меню его открытия и разблокировать его при закрытии окна. Для этого используем пункт меню Окна — **Единственная форма**. Этому пункту меню дадим имя *FormA*, и он должен иметь атрибут доступа *public*. Создаем таким образом почти пустую форму *Form4*. В число ее переменных включим ссылку на главное окно: `public Form1 f1;`

Создание и открытие этого окна:

```
private void FormA_Click(object sender, EventArgs e)
{
    Form4 f4;
    f4 = new Form4();
    f4.MdiParent = this;
    // Установили значение свойства f4
    FormA.Enabled = false; //закроем пункт меню
    f4.f1 = this;
    // передаем адрес главного окна в Form4, для обеспечения
    // возможности передачи сообщения в обратном направлении
    f4.Show(); }

```

При закрытии пункт меню **Единственная форма** (его имя *FormA*) должен быть снова освобожден. Тонкость в следующем: форму можно закрыть несколькими способами: нажатием на командную кнопку, нажатием на кнопку X, через системное меню и т. д. В любом случае пункт меню должен быть разблокирован. Для этого используем событие, связанное с закрываемым окном: *FormClosed* (происходит после закрытия) или *FormClosing* (происходит до закрытия). В нашем случае можно использовать любое из них. Найдем нужное событие в списке, сделаем на его имени двойной щелчок и напишем реализацию:

```
private void Form4_FormClosed(object sender,
FormClosedEventArgs e)
{//Сообщение в обратном направлении
f1.FormA.Enabled = true;
}
```

В *MDI*-приложениях возможно изменение меню в главном окне при открытии того или иного подчиненного окна. Для этого создаем меню в подчиненном окне. При открытии этого окна оно соединяется с меню главного окна.

5.3. Реализация алгоритмов

Каждая программ состоит из интерфейса пользователя и логики решаемой задачи. В предыдущих параграфах мы рассмотрели создание интерфейса, логику решения задачи ставили в соответствие командным кнопкам и/или пунктам меню. Так можно поступить, если алгоритм решаемой задачи прост. В более сложных случаях рекомендуется разделить интерфейс и логику. Это упрощает разработку интерфейса и реализацию алгоритмов решаемых задач. Кроме того, такой подход упрощает сопровождение программы (можно отдельно модифицировать как интерфейс, так и алгоритмы). Также это упрощает повторное использование реализованных алгоритмов.

Разделение алгоритмов и интерфейса можно выполнить на трех уровнях:

- В состав стандартного класса, реализующего интерфейс, добавим свои функции, реализующие алгоритм.

- В эту область имен добавим собственные классы. Имейте в виду: собственные классы должны располагаться ПОСЛЕ стандартных.
- Создание отдельных библиотек классов, которые будут подключены.

Рассмотрим здесь использование дополнительного класса в существующем проекте. Третий подход рассмотрим в следующем параграфе. Дополнительный класс может быть создан «с нуля» в самом проекте или можно подключить к проекту ранее созданный класс. Рассмотрим второй подход и подключим к проекту, рассмотренному в п. 5.4.4, класс *MyClass*, созданный нами в п. 4.8. Для этого поставим курсор мыши на имя проекта и из контекстного меню выберем по очереди Добавить — Существующий элемент. После этого найдем проект, при выполнении которого требуемый класс был сгенерирован (в нашем случае *Mod48*), сгенерированные классы находятся там в папке *Mod48Lib*, а в этой папке имеется папка *GeneratedCode*, где мы и найдем требуемый класс и подключим его к проекту. Приведем текст добавленного класса. Часть класса, которая не используется в данном примере, закомментирована и здесь не приводится. Оставим только функцию суммирования одномерного массива.

```
public class MyClass : ClassX
{
    public double [] mas;
    public MyClass(double[] mas)
    {
        this.mas = new double[mas.Length];
        for (int i = 0; i < mas.Length; i++)
            this.mas[i] = mas[i];
    }
    /*
        Пропущенная часть
    */
    public double Sum12()
    {
        return mas.Sum();
    }
}
```

Интерфейс приложения был представлен на рис. 5.3. В п. 5.4.4 осталась нереализованной кнопка «Класс», реализация которой приведена ниже. С помощью прикрепленного класса находим сумму строки, номер которого задан.

```
private void ClCl2_Click(object sender, EventArgs e)
{
    int kol;
    double[] arr;
    double z;
    MyClass my;
```

```

//объявим переменную типа класс MyClass
    kol = Convert.ToInt32(NumRow.Text);
    if (kol < 0 || kol > n - 1) kol = 0;
//проверим, находится ли номер строки
//в допустимых пределах
    arr = new double[dataGridView1.Columns.Count];
//создаем массив для размещения строки исходного
    for (int i = 0; i < arr.Length; i++)
arr[i] = Convert.ToDouble(dataGridView1.Rows[kol].Cells[i].Value);
//заполняем массив элементами заданной строки
    my = new MyClass(arr);
//запускаем конструктор класса
    z = my.Sum12();
    Cl1.Text = z.ToString("F3");}

```

5.4. Работа с библиотекой классов

Рассмотрим создание отдельной библиотеки классов, которую можно подключить к любому проекту. Для этого создадим проект, который так и называется «Библиотека классов». Эту библиотеку можно создать «с нуля» или путем добавления туда заранее сгенерированных классов из проекта моделирования. Выберем первый путь (подключение сгенерированных классов было рассмотрено в предыдущем параграфе) и создадим библиотеку *ClassLibrary1*. При создании библиотеки классов надо соблюдать следующие требования:

- Классы, которые должны быть доступны за пределами создаваемой библиотеки, должны иметь атрибут доступа *public*.
- Для обеспечения универсальности создаваемой библиотеки желательно исходные данные передавать в функции через параметры, а результаты их работы получать тоже через параметры или в виде возвращаемых значений. Желательно избегать использование средств консольного ввода/вывода и обращения к формам (хотя это НЕ запрещено).

Пусть у нас имеется следующая библиотека:

```

namespace ClassLibrary1
{
    /// <summary>
    /// Это базовый класс
    /// </summary>
    public class Class1
    {
        protected double[] mas;
        /// <summary>
        /// Конструктор
        /// </summary>
        /// <param name="mas">Исходный массив</param>

```

```

public Class1 (double []mas)
{
    this.mas=new double [mas.Length];
    mas.CopyTo(this.mas,0);
}
/// <summary>
/// Это свойство - количество положительных элементов
/// </summary>
public int KolPol
{
    get {
        return mas.Where(p => p > 0).Count();
    }
}
public int KolNeg
{
    get {
        return mas.Where(p => p < 0).Count();
    }
}
/// <summary>
/// Это класс-наследник
/// </summary>
public class Class2:Class1
{
    protected double c1, c2;
    double sum = -10000;
    /// <summary>
    /// </summary>
    /// <param name="temp"></param>
public Class2(double []temp):base(temp)
{
    /// <summary>
    /// Ввод данных в класс
    /// </summary>
    /// <param name="c1">Нижняя граница</param>
    /// <param name="c2">
    /// Верхняя граница
    /// </param>
public void inpt(double c1,double c2)
{
    this.c1 = c1; this.c2 = c2;
}
    /// <summary>

```

```

        /// Функция с возвращаемым значением
        /// </summary>
    /// <returns>Сформированный массив</returns>
    public double [] NewMas()
    {
        return mas.Where(p => p > c1 && p < c2).ToArray();
    }

        /// <summary>
        /// Функция без возвращаемого значения
        /// </summary>
    /// <param name="d1">Граница</param>
        /// <param name="cnt">
        /// Результат
        /// </param>
    public void fun1(double d1,out int cnt)
    {
        cnt = 0;
        foreach (double x in mas)
            if (x > d1) cnt++;
    } } }

```



Структура классов и их функций предельно проста и не нуждается в разъяснениях. Желательно включить в библиотеку классов *XML*-комментарии, что сильно облегчает использование библиотеки в будущем (*XML* — *eXtensible Markup Language* — расширенный язык разметки). Напомним: признаком *XML*-комментария в *C#* является *///*. В библиотеке классов *XML*-комментарии целесообразно ставить перед классами и перед функциями (конструктор тоже функция). В таком случае будет автоматически создан каркас комментария: *summary* для общей характеристики класса или функции, *param* для каждого формального параметра при их наличии, *returns* для возвращаемого значения при его наличии. Остается лишь писать необходимый текст, при этом недопустимо нарушить структуру комментария: любой текст должен быть между знаками, например *<summary> . . . </summary>*, нарушение этого правила считается синтаксической ошибкой.

Для завершения создания библиотеки:

- Поставим курсор мыши на имя созданной библиотеки (*ClassLibrary1*) и выберем из контекстного меню пункт «Свойства», в открывающемся окне свойств пункт «Сборка (*Build*)». В нижней его части можно определить папку размещения создаваемой библиотеки (по умолчанию *bin\debug* в папке проекта).
- Ставим «галочку» при *XML*.
- После этого из меню *Visual Studio* выберем Сборка — Построить (*Build — Build Solution*) *ClassLibrary1*. Если в библиотеке были допущены синтаксические ошибки, то появятся диагностика, если их нет — библиотека будет создана в виде файла с расширением **.dll*.

Использование созданной библиотеки. Для простоты создадим консольное приложение *useClassLibr1*. Для подключения созданной библиотеки в обозревателе решения найдем пункт *References* и из его контекстного меню выберем добавление ссылки. Откроется диалоговое окно, с помощью кнопки Обзор найдем нужную папку (если все было сделано правильно, то там должен быть файл *ClassLibrary1.dll*) и добавим ссылку. Чтобы освободиться от необходимости писать при обращении к компонентам библиотеки полные имена, можно добавить

using ClassLibrary1;

После этого можем работать с классами из библиотеки точно так, как работаем с классами, созданными в текущем проекте:

```
static void Main(string[] args)
{
    double[] mas,res;
    int n,mpol,motr,k;
    Class2 my;

// ввод mas
    my = new Class2(mas);
    mpol = my.KolPol;
    k=my.fun1()
    my.inpt(1, 5);
    my.fun1(2.3, out k);

//и т.д.}
```

5.5. Реализация взаимодействия с базой данных

5.5.1. Структура базы данных

Практически во всех задачах необходимы условно-постоянные данные, которые используются многократно и которые должны храниться в базе данных. Рассмотрим в этом пункте использование баз данных в программах на C# в *Microsoft Visual Studio*. Пусть база данных создана с использованием системы управления базами данных *Microsoft SQL Server*. Созданию баз данных посвящена обширная литература, поэтому мы не будем на этом останавливаться. Пусть наша база данных состоит из двух таблиц.

Groups — учебные группы (номер группы — ключ и специальность).

Имя	Тип данных
GrNum	char(10)
Спец	varchar(50)

Students — данные о студентах (табельный номер — ключ, фамилия, имя, группа, рост, вес, дата рождения). Автор, конечно, отдает себе отчет о том, что рост и вес неважные показатели, но нужны простые числовые данные! Типы

данных видны из рисунка. Между таблицами установлена связь по полям *GrNum* и *Gr* с включением режима защиты ссылочной целостности: ни один студент не может учиться в несуществующей группе. Напомним, что в базах данных не допускается повторение значений в ключевом поле. Кроме того, существует представление *FamSpec*, созданное на основе двух таблиц и содержащее поля фамилия, имя, специальность. В программах на *C#* представления могут быть обработаны аналогично таблицам, за исключением внесения изменений в данные. Кроме того, в базе данных имеются хранимые процедуры и функции. Рассмотрим создание приложения «клиент — сервер», где данные, хранимые процедуры и функции хранятся и выполняются на сервере. Клиент должен осуществлять их вызов и пользоваться результатами, возможно, с дальнейшей обработкой.

Имя	Тип данных	Допустимы значения NULL
TabNum	char(10)	<input type="checkbox"/>
Fam	varchar(20)	<input type="checkbox"/>
Name1	varchar(20)	<input type="checkbox"/>
Gr	char(10)	<input type="checkbox"/>
Rost	smallint	<input type="checkbox"/>
Ves	smallint	<input type="checkbox"/>
DataR	datetime	<input type="checkbox"/>

Хранимые процедуры и функции написаны на языке *T-SQL*, ограничимся простейшими средствами этого языка (-- это комментарии на *T-SQL*).

Хранимые процедуры. Извлечение данных.

```
CREATE PROCEDURE dbo.CountStudents
(
    @par1 int = 150,
    -- входной параметр со значением по умолчанию
    @par2 int OUTPUT ) -- результат процедуры
AS
    SET @par2=( --найти количество студентов,
    -- ростом выше заданного
    Select COUNT(*)
    From Students
    Where Rost>@par1)
    RETURN )
```

Удаление данных. Повторение табельных номеров не допускается, поэтому будет удален один студент или ни одного.

```
CREATE PROCEDURE dbo.Del
( @par1 char(50))
--входной параметр без значения по умолчанию
AS--удалить студента с заданным табельным номером
```



```
DELETE FROM Students Where TabNum=@par1
RETURN
```

Добавление данных. Приводим простейший вариант этого оператора, в нем должны быть заданы значения для всех полей, и в INTO они должны быть в такой последовательности, как в таблице.

```
CREATE PROCEDURE dbo.INSRT
(    @ng char(10), -- номер группы
  @sp VarChar(50)  ) -- специальность
```

```
AS
INSERT INTO Groups Values(@ng, @sp)
Изменение данных
```


```
CREATE PROCEDURE dbo.Updte
(    @TabNum1 char(10), -- табельный номер
  @f1 varchar(50)      ) -- фамилия
```

```
AS
--поменять фамилию студента с заданным табельным номером
UPDATE Students SET Fam=@f1 WHERE TabNum=@TabNum1
RETURN
```

Все хранимые процедуры обновления данных будут приведены в исполнение немедленно, без какого-либо предупреждения, если они только не противоречат заложенным в базу данных ограничениям. В случае такой необходимости это следует предусмотреть в клиентской части.

Функции. Функция, возвращающая одно значение.

```
CREATE FUNCTION dbo.Scal(    @par1 int ) --входной параметр
RETURNS int--тип возвращаемого значения
AS
```



```
BEGIN
DECLARE @n int-- локальная переменная
SET @n=(SELECT COUNT(*) -- количество студентов,
        -- ростом выше заданного
FROM Students
WHERE Rost>@par1)
RETURN @n-- возвращать @n
END
```

Функции, возвращающие таблицу.

Функция, состоящая из одного оператора.

```
CREATE FUNCTION dbo.Function2
( @par1 int, @par2 int)
RETURNS TABLE-- возвращает таблицу
AS-- вывести фамилии и имена студентов,
```

```
-- массой тела в заданном интервале
RETURN SELECT Fam, Name1 FROM Students
WHERE Ves BETWEEN @par1 AND @par2
```

Функция общего вида. В функции задана структура таблицы, которая затем заполняется данными.

```
CREATE FUNCTION dbo.Function3
( @par1 DateTime )
RETURNS @tabl TABLE (Fam1 varchar(20),Nm Varchar(20))
AS-- Вывести фамилии и имена студентов,
--родившийся после заданной даты.
BEGIN
    INSERT INTO @tabl
    SELECT Fam,Name1 FROM Students
    WHERE DataR>@par1
RETURN
```



5.5.2. Структура проекта

В созданном для работы с базами данных проекте предусмотрены следующие возможности:

- Просмотр таблиц базы данных с возможностью изменения или без этого.
- Извлечение данных с помощью хранимых процедур и функций.
- Внесение изменений с помощью хранимых процедур.
- Использование хранимых данных в качестве исходных при решении задач.
- Запись в базу данных результатов решения задач.

Создаем новый проект *WindowsForms*. По умолчанию будет создано *SDI*-приложение. Структура меню (создание меню было рассмотрено выше).

Файл	Просмотр	Извлечение	Изменение	Обработка
Выход	Таблица	данных Скалярные	данных Добавление	Получение результата
	Редактирование	Табличные	Изменение значений Удаление	



Для связывания проекта с источником данных используем компонент *BindingSource* из палитры Данные (*Data*).

1. Определим значение его свойства *DataSource*. Для этого имеется серия диалоговых окон, в которых по очереди выбираем базу данных, набор данных, создадим новое подключение, выберем *SQLServer*, имя сервера и имя базы данных и подключим все компоненты, предлагаемые в диалоговом окне для этого. В нашем случае — все перечисленные выше таблицы, представление, хранимые процедуры и функции.

2. Определим значения его свойства *DataMember*, оно имеет выпадающий список, в который включены наши две таблицы, представление и функции, возвращающие таблицу. Всех их по очереди и подключим.

В результате внизу под формой появятся следующие компоненты:

Data0512Set — набор данных, который представляет в проекте всю базу данных; *studentstableAdapter*, *groupstableadapter*, *famSpectableAdapter*, *function2TableAdapter*, *function3TableAdapter* — все адаптеры обеспечивают связь с соответствующей частью базы данных. Поменяем у всех названных компонентов атрибут доступа (*Modifiers*) на *public*, чтобы обеспечить доступ к ним из других форм.

В программном модуле созданного окна внесем следующие дополнения:

```
public partial class Form1 : Form
{
    public static Data0512DataSet ds;
        //представляет базу данных
    public static
Data0512DataSetTableAdapters.QueriesTableAdapter qa;
    // представляет запросы, его назначение будет видно позже
    public Form1()
    {
        InitializeComponent();
        ds = new Data0512DataSet(); //запуск конструкторов
        qa = new Data0512DataSetTableAdapters.
            QueriesTableAdapter();
    }

    private void выходToolStripMenuItem_Click(object sender,
    EventArgs e)
    {
        Close(); //выход из приложения    }
    ... Класс Form1 продолжается!
```

После этих подготовительных действий займемся созданием и запуском форм.

5.5.3. Просмотр данных без изменений

Форма для просмотра таблиц базы данных (рис. 5.9) без возможности внесения изменений (точнее, на экране изменения ввести можно, но они не будут сохранены). Форма показана после нажатия кнопки «Студенты» и показывает эту таблицу.

TabNum	Fam	Name1	Gr	Rost	Ves	DataR
222	Kotova	Vera	02	178	60	14.02....
333	Ivanov	Petr	02	180	80	11.11....
444	Krotov	Leonid	03	175	79	29.02....
555	Liadova	Nina	03	183	75	31.01....
666	Frolov	Ivan	02	170	68	12.12....
777	Petrova	Ksenya	02	183	79	01.01....

Рис. 5.9

Программа этой формы

```
public partial class Form2 : Form
{
    public Form1 f1; //для связи с формой Form1
    public Form2()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        // извлечение данных из таблицы Students,
        //передача только в одну сторону
        dataGridView1.DataSource = f1.studentsTableAdapter.GetData();
    }

    private void button2_Click(object sender, EventArgs e)
    {
        dataGridView1.DataSource =
            f1.groupsTableAdapter.GetData();
    }

    private void button3_Click(object sender, EventArgs e)
    {
        dataGridView1.DataSource =
            f1.famSpecTableAdapter.GetData();
    }

    private void Form2_Load(object sender, EventArgs e)
    {
        // При каждом открытии этой формы адаптер загружается
        // заново, чтобы отражать последние изменения
        f1.studentsTableAdapter.Fill(f1.data0512DataSet.Students);
        f1.groupsTableAdapter.Fill(f1.data0512DataSet.Groups);
    }
}
```

Создание и открытие этой формы из главной формы, форма модальная для предотвращения внесения изменений в данные во время просмотра.

```
private void таблицаToolStripMenuItem_Click(object sender,
EventArgs e)
{
    Form2 f2;
    f2 = new Form2();
    f2.f1 = this; //Передача сообщения в другой класс
    f2.ShowDialog();
}
```

5.5.4. Просмотр и изменение данных

Для обеспечения изменений данных в таблицах *Students* и *Groups* необходимо организовать прямую связь между самими таблицами в базе данных и их представлениями на форме. Для этого нанесем на *Form1* еще два компонента *BindingSource*, свяжем их с базой данных, как было описано выше, но на последней стадии установления связи выберем из предложенных только по одной таблице. Получим компоненты *BindingSource2* и *BindingSource3* (атрибут доступа у обоих должен быть *public*), но в их свойствах *DataMember* выберем только *Students* и *Groups*. Форма для редактирования таблиц представлена на рис. 5.10, форма показана после нажатия кнопки «Показать студент».

TabNum	Fam	Name1	Gr	Rost	Ves	DataF
222	Kotova	Vera	02	178	80	14.02.
333	Ivanov	Petr	02	180	80	11.11.
444	Krotov	Leonid	03	175	79	29.02.
555	Liadova	Nina	03	183	75	31.01.
666	Frolov	Ivan	02	170	68	12.12.

Рис. 5.10

Кнопки изменения при открытии формы «серые» и откроются только после открытия соответствующей таблицы. Программный модуль этой формы:

```
public partial class Form3 : Form
{
    public Form1 f1; //для связис Form1
    public Form3()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
```

```
{ // связь с таблицей Students через bindingSource2
    dataGridView1.DataSource = f1.bindingSource2;
    button2.Enabled = true;
    button4.Enabled = false;    }
```

```
private void button2_Click(object sender, EventArgs e)
{ // сохранение внесенных изменений
    f1.studentsTableAdapter.Update
      (f1.data0512DataSet.Students);
}
```

```
private void button3_Click(object sender, EventArgs e)
{ // связь с таблицей Students через bindingSource3
    dataGridView1.DataSource = f1.bindingSource3;
    button4.Enabled=true;
    button2.Enabled=false;    }
```

```
private void button4_Click(object sender, EventArgs e)
{ // сохранение внесенных изменений
f1.groupsTableAdapter.Update(f1.data0512DataSet.Groups);
} } }
```

Открытие этой формы:

```
private void представлениеToolStripMenuItem_Click(object sender, EventArgs e)
{
    Form3 f3;
    f3 = new Form3();
    f3.f1 = this;
    f3.ShowDialog();    }
```

5.5.5. Извлечение данных

Реализация извлечения данных зависит от того, будет извлечено одно значение (с помощью хранимой процедуры или функции, возвращающей скалярное значение) или целая таблица (с помощью функции). Форму *Form4* мы не приводим ввиду ее простоты: на ней только поле для ввода одного числового значения, две командные кнопки и поле для вывода результата. Ее модуль приведен ниже:

```
public partial class Form4 : Form
{
```



```
public Form4()
{
    InitializeComponent();
}
```

```
private void button1_Click(object sender, EventArgs e)
{ // qa статическая переменная,
  // можем обратиться через имя класса
    int? k1, k2;
    k2=null;
    k1 = Int32.Parse(textBox1.Text);
    Form1.qa.CountStudents(k1, ref k2);
/* Вызов хранимой процедуры CountStudents, второй
   параметр выходной, поэтому аргумент должен иметь
   атрибут ref, но такие аргументы должны
   иметь начальное значение. */
    label1.Text = "" + k2; ;
}
```

```
private void button2_Click(object sender, EventArgs e)
{
    int? k1, k2;
    k1 = Int32.Parse(textBox1.Text);
    k2 = Form1.qa.Scal(k1);
// Вызов функции Scal, возвращающую одно значение
    label1.Text = "" + k2; ;
} } }
```

Для вывода извлеченных функцией таблиц предусмотрена *Form5* (рис. 5.11) форма показана после вызова функции *function2* (Верхняя кнопка).

The screenshot shows a Windows application window titled 'Form5'. It contains two input fields for mass ranges: 'Нижний предел массы' (Lower mass limit) with value 60 and 'Верхний предел массы' (Upper mass limit) with value 70. A 'Найти' (Find) button is to the right. Below these is a table with two columns: 'Fam' and 'Name1'. The table contains two rows: 'Kotova Vera' and 'Frolov Ivan'. At the bottom, there is a text input field for 'Дата рождения (День, месяц, год)' (Date of birth) with the value '15.12.1990' and another 'Найти' button. A watermark logo is visible in the bottom right corner of the form area.

Fam	Name1
Kotova	Vera
Frolov	Ivan

Рис. 5.11

Модуль этой формы:

```
public partial class Form5 : Form
{
```

```

public Form1 f1;
public Form5()
{
    InitializeComponent();
}
private void button1_Click(object sender, EventArgs e)
{ //вызов функции function2
    int k1,k2;
    k1 = Int32.Parse(textBox1.Text);
    k2 = Int32.Parse(textBox2.Text);
    dataGridView1.DataSource =
    f1.function2TableAdapter.GetData(k1,k2);
}
private void button2_Click(object sender, EventArgs e)
{ //вызов функции function3
    DateTime d1;
    d1 = Convert.ToDateTime(textBox3.Text);
    dataGridView1.DataSource =
    f1.function3TableAdapter.GetData(d1);} }

```

Открытие этой формы традиционное, и мы его не приводим.

5.5.6. Изменение данных хранимыми процедурами

В данном приложении новые значения будут введены с экрана. Таким же образом можно записать в базу данных и результаты вычислений. Рассмотрим три вида изменений:

- Добавление новой записи.
- Изменение полей записи.
- Удаление записи.

Для выполнения всех названных операций создана *Form6* (рис. 5.12).

Рис. 5.12

Его модуль:

```
public partial class Form6 : Form
```

```
{
```

```
public Form6()
```

```
{
```

```
    InitializeComponent();
```

```
}
```

```
private void button1_Click(object sender, EventArgs e)
```

```
{ // добавление записи в Groups
```

```
    string sgr, ssp;
```

```
    sgr=textBox1.Text;
```

```
    ssp=textBox2.Text;
```

```
try
```

```
{ //Проверим, не возникла ли исключительная ситуация.
```

```
    // Например, при повторении ключей.
```

```
Form1.qa.INSRT(sgr, ssp);//Запуск хранимой процедуры INSRT
```

```
    textBox1.Text = "";
```

```
    textBox2.Text = "";
```

```
}
```

```
catch
```

```
{
```

```
MessageBox.Show("Некорректные данные, добавление невоз-  
можно");
```

```
}
```

```
}
```

```
private void button2_Click(object sender, EventArgs e)
```

```
{
```

```
    string s1;
```

```
    DataRow dr;
```

```
    s1 = textBox3.Text;
```

```
try
```

```
{
```

```
dr = f1.data0512DataSet.Students.FindByTabNum(s1);
```

```
/* Проверим, имеется ли такой студент.
```

```
Если нет - возникает исключительная ситуация.
```

```
Если да - спросим "Надо ли удалить?" */
```

```
if (DialogResult.Yes == MessageBox.Show(this, "Удалить
```

```
     "+dr["Fam"]+" " +dr["Name1"]+"?",
```

```
    "Подтверждение удаления", MessageBoxButtons.YesNo))
```



```

        Form1.qa.Del(s1);
        textBox3.Clear();
f1.studentsTableAdapter.Fill(f1.data0512DataSet.Students);
        // Обновим представление таблицы в нашем приложении
    }
    catch
    {
        MessageBox.Show("Такого студента нет");
    }
}

```

```

private void button3_Click(object sender, EventArgs e)
{
    string s1, s2;
    DataRow dr;
    s1 = textBox4.Text;
    s2 = textBox5.Text;

    try
    {
dr = f1.data0512DataSet.Students.FindByTabNum(s1);
        /* Проверим, имеется ли такой студент
           Если нет - возникает исключительная ситуация
           Если да - спросим "Надо ли менять фамилию?" */

        if (DialogResult.Yes == MessageBox.Show(this,"Студент " +
dr["Fam"]+" "+dr["Name1"]+
" ". Подтверждение изменения фамилии на"+s2 ,
"Изменение фамилии" , MessageBoxButtons.YesNo))

            Form1.qa.Updte(s1, s2);
            textBox4.Clear();
            textBox5.Clear();
f1.studentsTableAdapter.Fill(f1.data0512DataSet.Students);
        // Обновим представление таблицы в нашем приложении
    }
    catch
    {
        MessageBox.Show("Такого студента нет");
    }
}

```

Создание и открытие этой формы традиционно, и мы его не приводим.

5.5.7. Использование данных из базы для вычислений

В принципе существуют два подхода к использованию данных из базы для вычислений в своей программе.

1. Существует стандартный класс *DataTable*, которым мы уже неоднократно пользовались. Можно записать нужные данные в него (таблицу базы данных, представление или результат функции) и выполнять обработку. Внесенные при обработке изменения в базу данных не передаются.
2. Можно работать непосредственно с таблицами базы данных. Тогда можно передать туда и изменения. Можно работать и с представлениями, но тогда изменения невозможны.

Кроме того, покажем на этом примере прямой доступ к данным по ключу: так как ключ не может повторяться, то либо будет найдена единственная запись, или ничего.

Форма для обоих случаев представлена на рис. 5.13.

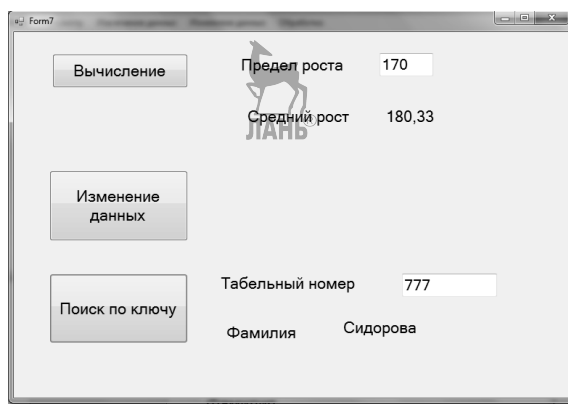


Рис. 5.13

Модуль этой формы:

```
publicForm1 f1;  
    public DataTable tabl1;  
public Form7()  
{  
    InitializeComponent();  
}  
  
private void button1_Click(object sender, EventArgs e)  
{  
    // Работаем с DataTable.  
    // вычисляем средний рост студентов, ростом выше с.  
    Double temp, sum = 0;  
    int kol = 0;  
    double c = Double.Parse(textBox1.Text);
```

```

for(int i=0; i<tabl1.Rows.Count; i++)
{
    temp = Convert.ToDouble(tabl1.Rows[i]["Rost"]);
/* Обращение к столбцам возможно как по имени,
   так и по номеру.
   Преобразование типа данных всегда обязательно! */
    if(temp>c)
    {
        sum += temp;
        kol++;
    }
}
if (kol != 0)
{
    sum = sum / kol;
    label1.Text = sum.ToString("F2");
}
else
    label1.Text = «Таких нет»;
// Решение этой же задачи с помощью языка LINQ
kol = f1.data0512DataSet.Students.Where(p =>
    p.Rost > c).Count();
if (kol != 0)
{
    //Функция Average() выдает ошибку при отсутствии данных
    label1.Text = (f1.data0512DataSet.Students.Where(p => p.Rost >
c).Select(q =>
    Convert.ToDouble(q.Rost)).Average()).ToString("F2");
    // Преобразование типа данных всегда обязательно!
}
else
    label1.Text = "Таких нет";
}

private void button2_Click(object sender, EventArgs e)
{
    // работа непосредственно с таблицей базы данных,
    // внесение изменений
    int temp;
    for(int i=0;i<f1.data0512DataSet.Students.Rows.Count;i++)
    {
        temp = Convert.ToInt32
(f1.data0512DataSet.Students.Rows[i]["Ves"]);
        temp -= 5;
        f1.data0512DataSet.Students.Rows[i]["Ves"]=temp;

```

```

}
f1.studentsTableAdapter.Update(f1.data0512DataSet.Students);
}

private void button3_Click(object sender, EventArgs e)
{ //Поиск данных по ключу.DataRow стандартный класс
  // для представления строки таблицы
    DataRow dr;
    try
    { // Если искомого ключа нет, то возникает исключение.
dr = f1.data0512DataSet.Students.FindByTabNum(textBox2.Text);
      label6.Text = dr[1].ToString();
      // вывод фамилии найденного студента
    }
    catch
    {
      label6.Text = «Нет в списке»;
    } }
}

```

Открытие этой формы:

```

private void получениеРезультатаToolStripMenuItem_Click(object sender,
EventArgs e)
{
    Form7 f77;
    f77 = new Form7();
    f77.f1 = this;
    f77.tabl1 = studentsTableAdapter.GetData();
    f77.Show();
}

```

5.6. Использование стандартных классов в реализации

На языке *C#* имеется набор классов для работы с динамическими структурами данных. Использование этих классов в практическом программировании желательно, особенно при работе с большими объемами данных: с одной стороны, таким образом можно повысить производительность труда программистов и, с другой стороны, повысить и эффективность создаваемых программ.

Ограничимся в данном пособии рассмотрением стандартных классов список — *List* и словарь — *Dictionary*.

5.6.1. Работа с классом *List*

Для начала проведем краткое сравнение двух способов агрегации данных: массивов и списков. Преимуществами массивов является быстрота адресации, потому что элементы расположены в памяти друг за другом и всегда имеют одинаковую структуру и одинаковый объем занимаемой памяти. Но при необходимости частого удаления и добавления элементов, особенно если требуется их упорядоченность, возникают существенные трудности. Элементы списков связаны друг с другом с помощью адресов связи, поэтому их удаление и добавление не требуют перемещения остальных элементов. Отсюда рекомендация: списки предпочтительны при высокой динамичности их состава. Рассмотрим работу с классом *List* в двух случаях:

- Список состоит из стандартных данных *C#*.
- Список состоит из данных, структура которых определена пользователем.

Список из стандартных данных

Приведем пример, в котором показаны все основные действия над классом *List*. Рассмотрим подробно функцию

```
static int c;  
static bool pred(int x)  
{ // условие поиска и/или удаления элементов  
    return(x>c);    }
```

Как видно из комментария, она предназначена для определения условий, которые потом будут использованы при поиске и удалении элементов списка. Атрибут *static* выбирается по обычным правилам. Интерфейс функции строго фиксирован:

```
bool имя_функции (параметр_типа_элемент_списка)  
{ ..... }
```

Глобальная переменная *c* необходима для внесения в функцию параметра, так как состав формальных параметров фиксирован, то другого пути нет. Разумеется, количество таких функции не ограничено.

namespace ListInt

```
{  
    class Program  
    {  
        static int c;  
        static bool pred(int x)  
        { // условие поиска и/или удаления элементов  
            if (x > c) return true;  
            else return false;  
        }  
    }  
}
```

```

static void Main(string[] args)
{
    List<int> lst1, lst2;
    int el, kol;
    bool b1;
    lst1 = newList<int>();
    // Для создания списка запускаем конструктор класса List
    for (int i = 0; i < 6; i++)
    {
        // Заполнение списка
        Console.WriteLine("Введите элемент " + i + " ");
        el = Convert.ToInt32(Console.ReadLine());
        lst1.Add(el); // добавление элемента в список
    }
    // Допустим, что введены элементы 3, 12, -4, 65, 9, 100
    // Начинаем обработку списка
    b1 = lst1.Contains(65);
    // Содержится ли в списке заданный элемент, ответ true
    c = 25;
    kol = lst1.Find(pred);
    /* Будет найден 1-й элемент, удовлетворяющий условию,
       в данном случае 1-й элемент больше 25, при его
       отсутствии ответ 0, в нашем случае ответ 65 */

    kol = lst1.FindLast(pred);
    /* Будет найден последний элемент, удовлетворяющий
       условию, в данном случае последний элемент больше 25, при
       его отсутствии ответ 0, в нашем случае ответ 100 */
    kol = lst1.FindIndex(pred);
    /* Будет найден индекс первого элемента,
       удовлетворяющего условию, в нашем случае ответ 3,
       при отсутствии искомого элемента ответ -1
       аналогично FindLastIndex() */
    kol = lst1.IndexOf(9);
    /* Будет найден индекс первого вхождения заданного
       элемента, в нашем случае ответ 4
       при отсутствии искомого элемента ответ -1 */
    lst2 = lst1.FindAll(pred);
    /* Все элементы lst1, удовлетворяющие условию,
       будут размещены в lst2. Так как их количество заранее
       неизвестно, то для приема ответа нужен список или массив */
    int[] res = lst1.FindAll(pred).ToArray();
    // запись результата в массив, ответ 65 100

```

```

        lst2 = res.ToList();
// существует и функция переноса массива в список
        lst1.Sort();
// Сортировка списка по возрастанию
        lst1.Reverse();
// Изменение очередности элементов на противоположное

        el = 125;
        lst1.Insert(3, el);
/* Добавление нового элемента в заданное место.
   В нашем случае 125 займет 4-е с начала списка место.
   Размещение нового элемента за пределами
   списка - ошибка!
   Расширение списка - см. выше!           */
        lst1.Remove(-4);
/* Удаление заданного элемента из списка,
   при его отсутствии ничего не делается,
   но это и не ошибка!           */
        lst1.RemoveAt(0);
// Удаление элемента по индексу.
//Попытка удаления элемента за пределами списка - ошибка!
        lst1.RemoveAll(pred);
// Удаление всех элементов, удовлетворяющих условию.

// Обработка элементов списка
    for (int i = 0; i < lst1.Count; i++)
        lst1[i] = 48 * lst1.ElementAt(i);
/* Для изменения значения элемента - только lst1[i]
   Для извлечения значения элемента - оба варианта
   lst1.Count - количество элементов в списке */
Console.WriteLine("*****");
// Второй вариант извлечения элементов списка.
// Не забудьте про ограничения на foreach!
    foreach (int x in lst1)
        Console.WriteLine("" + x);
    Console.ReadLine();
}    }    }

```

Список со своими элементами

Для создания и обработки списка со своими элементами следует сначала определить структуру элементов и функции для их обслуживания (минимальный набор — ввод и вывод). Структуру элементов списка можно задать с помощью двух конструкций языка C#: *struct* и *class*. Напомним, что классы

являются ссылочными данными, поэтому список из переменных типа класс — это по существу список указателей на элементы. Структуры не являются ссылочными переменными, и соответствующий список состоит из самих экземпляров структур.

Для задания условий на элементах списка используется описанная выше функция. Если требуется выполнить упорядочивание элементов — то условия сортировки придется определить самому. Как это делать — рассмотрим позже.

Использование *struct*.

```
namespace ListStruct
{
    struct elem
    { // Задаем структуру элементов списка
        public string s;
        public int k;
        public void inpt()
        {
            Console.WriteLine("String "); s = Console.ReadLine();
            Console.WriteLine("Number ");
            k = Int32.Parse(Console.ReadLine());
        }
        public void otpt()
        {
            Console.WriteLine(s + "    " + k);
        }
    }

    class Program
    {
        static int upor(elem x1, elem x2)
        { // функция задает условие упорядочения по s
            return x1.s.CompareTo(x2.s);
        }
        static int c;
        static bool udal(elem x)
        { // функция задает условие на элементах
            return (x.k >= c);
        }
        static void Main(string[] args)
        {
            List<elem> lst1, lst2;
            lst1 = new List<elem>();
            elem el;
```

```

    int kol;
    string st1;
// Создание списка
    el = new elem();
    for (int i = 0; i < 6; i++)
    {
        el.inpt();
        lst1.Add(el);
    }
    el.s = "aaa";
    el.k = 25;
    kol = lst1.IndexOf(el);
/* Работает ТОЛЬКО при использовании для элементов списка
   struct индекс будет найден при совпадении
   ВСЕХ компонентов элемента */
    bool b1 = lst1.Contains(el);
/* Работает ТОЛЬКО при использовании для элементов списка
   struct ответ будет true при совпадении
   ВСЕХ компонентов элемента */
    lst1.Remove(el);
/* Работает ТОЛЬКО при использовании для элементов списка
   struct удаление будет выполнено
   при совпадении ВСЕХ компонентов элемента
   Обработка элементов */
    for (int i = 0; i < lst1.Count; i++)
    {
        el = lst1[i];
        el.k = el.k + 100;
        lst1[i]=el;
    }

    lst1.Sort(upor);
// сортировка элементов списка по критерию,
// заданному в upor
// Обработка элементов списка
    foreach (elem x in lst1)
        x.otpt();

    c = 15;
    el = lst1.Find(udal);
    el.otpt();
/* Будет найден и выведен первый элемент, удовлетворяющий
   условию или 0

```


Аналогично работают и `FindAll`, `FindLast` и другие, рассмотренные в предыдущем примере `*/`

```
    Console.ReadLine();  
} } }
```

Рассмотрим в следующем примере использование класса для задания структуры элементов списка. Заодно покажем использование в элементах списка динамических структур данных и стандартного интерфейса *Comparable<elem>*. По правилам *C#* в нашем классе *elem* должна быть реализация функции *intCompareTo(elem)*. При этом эта функция должна обеспечить возвращение всех значений -1, 0 и 1.

```
namespace ListClass  
{  
    class elem : Comparable<elem>  
{ // элемент списка, структура и функции обслуживания  
        public string s;  
        public double[] mas;  
        int kol;  
        public elem(int n)  
        {  
            mas = new double[n];  
        }  
        public void inpt()  
        {  
            Console.Write("String ");  
            s = Console.ReadLine();  
            for (int i = 0; i < mas.Length; i++)  
            {  
                Console.Write(i + " ");  
                mas[i] = Double.Parse(Console.ReadLine());  
            }  
        }  
        public void otpt()  
        {  
            Console.WriteLine(s);  
            foreach (double x in mas)  
                Console.WriteLine(x.ToString("F2"));  
        }  
        public int KolPol()  
        {  
            kol = 0;  
            foreach (double x in mas)
```

```

        if (x > 0) kol++;
        return kol;
    }
    public int CompareTo(elem y)
    { // реализация унаследованной из интерфейса функции
        if (this.KolPol() == y.KolPol()) return 0;
        if (this.KolPol() > y.KolPol()) return 1;
        else return -1;
    }
}

```

class Program

```

{
    static bool pred(elem x)
    {
        return(x.KolPol() == 0);
    }
    static void Main(string[] args)
    {
        List<elem> lst1;
        elem el;
        int m;
        lst1 = new List<elem>();
        // Создание списка
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine("Количество чисел в массиве "+i+" ");
            m = Convert.ToInt32(Console.ReadLine());
            el = new elem(m); //должен всегда быть в цикле
            el.inpt();
            lst1.Add(el);
        }
        lst1.Sort();
        // сортировка элементов списка по заданному в
        // int CompareTo(elem y) критерию
        lst1.RemoveAll(pred);
        // удаление элементов по условию
        for (int i = 0; i < lst1.Count; i++)
        { // Обработка элементов в цикле
            el = lst1.ElementAt(i);
            if (el.s.Length > 5) lst1.RemoveAt(i);
        }
    }
}

```

```

    bool b1 = lst1.Exists(pred);
// Проверка наличия элементов по условию
    // Обработка элементов
    foreach (elem z in lst1)
    {
        z.otpt();
        Console.WriteLine("'" + z.KolPol());
    }
Console.ReadLine();
} } }

```

Обратим внимание еще раз на то обстоятельство, что список этого примера состоит из указателей на экземпляры класса. Поэтому оператор выделения памяти *el = new elem(m)* должен быть в теле цикла (Почему?). Кроме того, не работают функции класса *List*, аргументами которых являются элементы списка (*IndexOf*, *Remove*, *Contains* ...). Функции, аргументами которых являются индексы и/или оформленные функциями предикаты, работают без проблем.

5.6.2. Работа со словарем — классом *Dictionary*

Класс *Dictionary* состоит из пар «ключ — значение». Ключом может быть любой простой тип данных, значение может иметь сложную структуру, в таком случае должны быть предварительно созданы класс и/или структура.

Словарь из простых данных

Будем работать с классом *Dictionary*, где ключ имеет тип *string*, а значение *int*. По существу, это массив с символьными индексами, но возможностями обработки списка.

```

namespace DictInt
{
class Program
{
    static void Main(string[] args)
    {
        Dictionary<string, int> dic1 =
            new Dictionary<string, int>();
// Объявление словаря:
//ключ имеет тип string, значение тип int
        string s;
        int k;
        for (int i = 0; i < 5; i++)

```



```

{ // Создание словаря
    Console.WriteLine("Key "); s = Console.ReadLine();
    Console.WriteLine("Value "); k = Convert.
        ToInt32(Console.ReadLine());
    try
    {
        dic1.Add(s, k);
// Добавление элемента, повторение ключа
//вызывает прерывание
    }
    catch
    {
        Console.WriteLine("Duplicate Key");
        i--;
    }
}
dic1["abc"] = 25;
// Если такого ключа нет - добавит, если ключ уже
//имеется - меняет соответствующее ему значение
dic1.Remove("xyz");
// Элемент с заданным ключом удаляется, если такого
// ключа нет - ничего не делается и сообщение не выдается
s="aaa";
bool b1 = dic1.ContainsKey(s);
bool b2 = dic1.ContainsValue(25);
// Проверка наличия заданного ключа и/или значения
s = "bbb";
try
{
// Поиск значения по ключу,
//отсутствие ключа вызывает прерывание
    k = dic1[s];
    Console.WriteLine("Key " + s + " Value " + k);
}
catch (KeyNotFoundException)
{
    Console.WriteLine("Key not find");
}

// Организация цикла по ключам, например,
// для изменения значений
string[] keys1 = new string[dic1.Count];
dic1.Keys.CopyTo(keys1, 0);

```

```

        for (int i = 0; i < keys1.Length; i++)
        {
            dic1[keys1[i]] += 100;
        }
// Цикл по ключам, не забудьте об особенностях foreach!
        foreach (KeyValuePair<string, int> x in dic1)
            Console.WriteLine(x.Key + " " + x.Value);
        Console.ReadLine();
    } } }

```

С помощью класса *Dictionary* можно легко решить следующую классическую задачу: на вход поступает последовательность слов. Вывести список введенных слов с указанием того, сколько раз каждое из них встречалось. Таким же образом можно обрабатывать результаты социологического опроса: каждый опрашиваемый назвал 3 любимых животных (артистов, деревьев и т. д.). Вывести список названных животных с указанием количества полученных голосов. При создании такой программы обработки надо решить вопрос о вводе и проверке корректности данных, а это выходит за рамки данного пособия.

```

SortedDictionary<string, int> dic2;
dic2 = new SortedDictionary<string, int>();
// В SortedDictionary словарь всегда упорядочен
// по возрастанию ключей
for (int i = 0; i < 24; i++)
{
    Console.Write(" Input a name ");
    s = Console.ReadLine();
    if (dic2.ContainsKey(s)) dic2[s]++;
    else dic2[s] = 1;
// Добавили очередной голос или получили первый голос
}
foreach (KeyValuePair<string, int> x in dic2)
    Console.WriteLine(x.Key + " " + x.Value);

```

Словарь из данных со сложной структурой

Пусть ключ по-прежнему имеет тип *string*. В качестве данных используем массив и в качестве примера обработки — функцию нахождения суммы. Почти все остается без изменений, но обратите внимание на то, что выполнять какие-либо операции только со значением (*value*) невозможно, потому что оно имеет теперь внутреннюю структуру; необходимо всегда указывать имя функции, которая и будет выполнена на компонентах значения.

```

namespace DictClass
{
    class elem
    {
        public double[] mas;
        public elem(int n)
        {
            mas = new double[n];
        }
        public void inpt()
        {
            for (int i = 0; i < mas.Length; i++)
            {
                Console.Write(i + " ");
                mas[i]=Double.Parse(Console.ReadLine());
            }
        }
        public void otpt()
        {
            Console.WriteLine("    Output ");
            for (int i = 0; i < mas.Length; i++)
                Console.WriteLine(i + " " +
                    mas[i].ToString("F2"));
        }
        public double sum(double c)
        {
            double temp = 0;
            foreach (double x in mas)
                if (x > +c) temp += x;
            return temp;
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Dictionary<string, elem> dic1;
        dic1 = new Dictionary<string, elem>();
        elem el1;
        string kl;
        for (int i = 0; i < 5; i++)
        { // Создание словаря

```

```

        Console.Write("Key ");
        kl = Console.ReadLine();
        el1 = new elem(i + 2);
        el1.inpt();
        dic1.Add(kl, el1); //повторение ключей недопустимо
    }

        // Добавление элемента
        el1 = new elem(4);
        el1.inpt();
        dic1.Add("abc", el1);
// dic1["abc"] = el1; // Добавление или замена
try
{ // Поиск элемента по ключу
    el1 = dic1["xyz"];
    el1.otpt();
}
catch (KeyNotFoundException)
{
    Console.WriteLine("Not found");
}

//Использование функции на элементах словаря
Console.WriteLine("Summa");
foreach (KeyValuePair<string, elem> w in dic1)
    Console.WriteLine(w.Key + " " + w.Value.sum(1.4));

// Вывод словаря
Console.WriteLine("Dictionary");
foreach (KeyValuePair<string, elem> u in dic1)
{
    Console.WriteLine("Key " + u.Key);
    u.Value.otpt();
}
Console.ReadLine();
} } }

```

5.6.3. Список из списков

Возможен и случай, когда список тоже состоит из списков. Рассмотрим в качестве примера список, который, в свою очередь, состоит из списков из целых чисел.

```

namespace LstLst
{
class Program
{
    static void Main(string[] args)
    {
        List<List<int>> lst1=new List<List<int>>>();
        // Список, элементами которого являются списки
        List<int> temp;
        // Вспомогательный список
        int k1, k2, k3;
        for (int i = 0; i < 3; i++)
        { // Пусть во внешнем списке будет 3 элемента
            Console.WriteLine("Elements in list "+i+" ");
            k1=Convert.ToInt32(Console.ReadLine());
            // k1 - количество элементов в i-м подсписке
            temp = new List<int>();
            // Создаем вспомогательный список
            for (int j = 0; j < k1; j++)
            {
                Console.WriteLine("Enter a number for element " + i + " ");
                k2=Convert.ToInt32(Console.ReadLine());
                temp.Add(k2);
            }
            lst1.Add(temp);
            // Добавляем вспомогательный список в основной
            // в качестве элемента
        }
        // Вывод (или обработка) всего списка
        k3=0;
        foreach (List<int>x in lst1)
        { // Цикл по внешнему списку
            Console.WriteLine("Elements of List " + k3);
            k3++;
            foreach (int z in x)
            { // Цикл по внутреннему списку
                Console.WriteLine(" " + z);
            }
            Console.WriteLine();
            // Второй вариант цикла по списку
        }
        for(int i=0; i<lst1.Count;i++)
    }
}

```



```

{ // Цикл по внешнему списку

    Console.WriteLine("*****");
    /* Этот цикл по внутреннему списку тоже работает
    foreach (int y in lst1[i])
    Console.WriteLine(y);
    Второй вариант цикла по внутреннему списку */
    temp = lst1[i];
    for (int j = 0; j < lst1[i].Count; j++)
        Console.WriteLine("" + temp[j]);
    }
Console.ReadLine();
} } }

```



5.6.4. Использование диалоговых окон для работы со стандартными классами

Рассмотрим *SDI*-приложение, в котором выполняется создание и обработка класса *List* с использованием диалоговых окон. Создание *SDI*-приложений рассмотрено выше.

Элементы списка имеют следующую структуру:

```

public class elem
{
    public string s;
    public int[] mas;
    public elem(string s,int[]mas)
    {
        this.s = s;
        this.mas = new int[mas.Length];
        for (int i = 0; i < mas.Length; i++)
            this.mas[i] = mas[i];
    }
    public int sumpolr(int c)
    {
        return mas.Where(q => q > c).Sum();
    }
    public int KolVo()
    {
        return mas.Where(p => p > 0).Count();
    }
}

```



Пусть меню имеет следующую структуру:

Файл

Выход

Обработка

Создание списка

Обработка списка

Изменение списка

Помощь

О программе

Окно «О программе» имеет традиционную структуру, и мы на нем останавливаться не будем.

Создание списка

Для создания списка используется окно на рис. 5.14. Его открытие — реализация пункта меню «Создание списка»

```
private void созданиеВпискаToolStripMenuItem_Click(object sender, EventArgs e)
```

```
{  
    Form4 f2;  
    F2 = new Form2();  
    F2.ShowDialog();    }  
}
```

Реализация кнопки «Добавить»:

```
private void button1_Click(object sender, EventArgs e)
```

```
{  
    string s1, s2;  
    elem el;  
    string[] temp;  
    int[] arr;  
    s1 = textBox1.Text;  
    s2 = textBox2.Text;  
    temp = s2.Split(';');
```



```
//разделение строки на элементы массива
```

```
    temp = temp.Where(q => q != "").ToArray();
```

```
// удаление пустых элементов
```

```
    arr = newint[temp.Length];
```

```
    for (int i = 0; i < arr.Length; i++)
```

```
        arr[i] = Convert.ToInt32(temp[i]);
```

```
    el = new elem(s1, arr); //создание нового элемента
```

```
    Form1.lst1.Add(el); //добавление нового элемента
```

```
    textBox1.Clear(); //чистка полей
```

```
    textBox2.Clear(); }
```

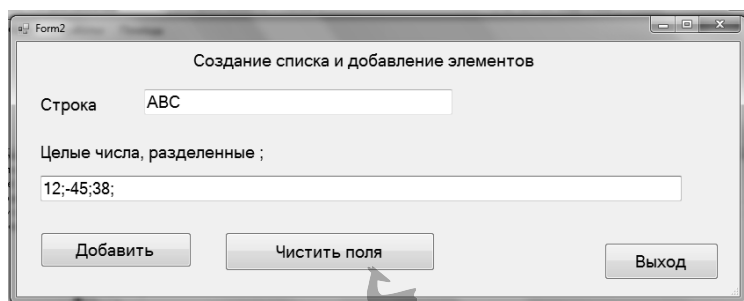


Рис. 5.14

Реализация кнопки «Чистить поля»:

```
private void button2_Click(object sender, EventArgs e)
{
    textBox1.Clear();
    textBox2.Clear();
}
```

Обработка списка

Для реализации пункта меню «Обработка списка» используется окно на рис. 5.15. Его открытие:

```
private void изменениеСпискаToolStripMenuItem_Click(object sender, EventArgs e)
{
    Form4 f4;
    F4 = new Form4();
    F4.Show();
}
```

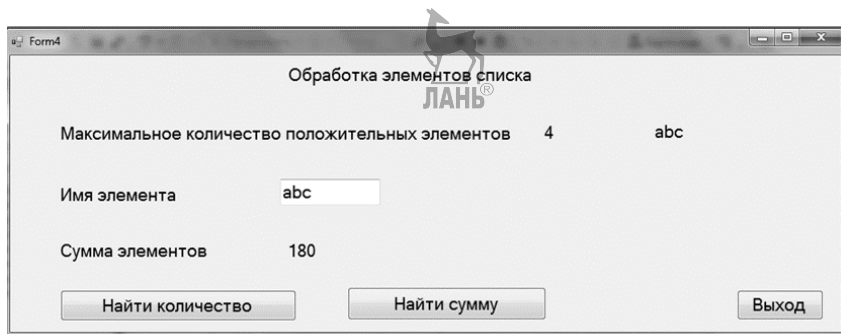


Рис. 5.15

Нажатие кнопки «Найти количество» выводит на экран максимальное количество положительных чисел в элементах списка и соответствующие им символичные строки. Ее реализация:

```

private void button1_Click(object sender, EventArgs e)
{
    int kol1, kol2;
    string[] s1;
    string st = "";
    kol1 = Form1.lst1.Select(p => p.KolVo()).Max();
    //реализация через функцию в составе elem
    kol2 = Form1.lst1.Select(p => p.mas.
Where(q => q > 0).Count()).Max();
    //Реализация только с помощью LINQ
    s1 = Form1.lst1.Where(q => q.KolVo() == kol2).
Select(p => p.s).ToArray();
    //В массив s1 будут собраны строки всех элементов,
    // имеющих max количество положительных
    label3.Text = " " + kol2;
foreach (string x in s1)
    //Подготовка вывода и вывод массива s1
    st += " " + x;
label4.Text = st; }
}

```

Реализация кнопки «Найти сумму». Будет найдена сумма чисел первого элемента списка с заданной символьной строкой или будет выдано сообщение об отсутствии.

```

private void button2_Click(object sender, EventArgs e)
{
    string s1 = textBox1.Text;
int[] sum1;
sum1 = Form1.lst1.Where(p => p.s == s1).
select(q => q.mas.Sum()).ToArray();
if (sum1.Length == 0)
MessageBox.Show("No element");
    else
        label7.Text = "" + sum1[0]; }
}

```

Изменение списка

Для реализации пункта меню «Обработка списка» используется окно на рис. 5.16. Его открытие:

```

private void изменениеСпискаToolStripMenuItem_Click(object sender, EventArgs e)
{
    Form3 f3;
}

```

```
f3 = new Form3();
f3.Show();    }
```

При нажатии кнопки «Показать» будет показан элемент списка с заданным номером (или выдано сообщение о его отсутствии). Реализация:

```
private void button1_Click(object sender, EventArgs e)
{
    elem el;
    int num = Int32.Parse(textBox1.Text);
    if (num < Form1.lst1.Count)
    {
        el = Form1.lst1[num];
        textBox2.Text = el.s;
string[] temp = new string[Form1.lst1[num].mas.Length];
        string s1 = "";
        foreach (int x in Form1.lst1[num].mas)
            s1 += x + ";";
        textBox3.Text = s1;
        button5.Enabled = true;
        button6.Enabled = true;
    }
    else
        MessageBox.Show("Всписке " + Form1.lst1.Count +
                        "элементов");    }
```

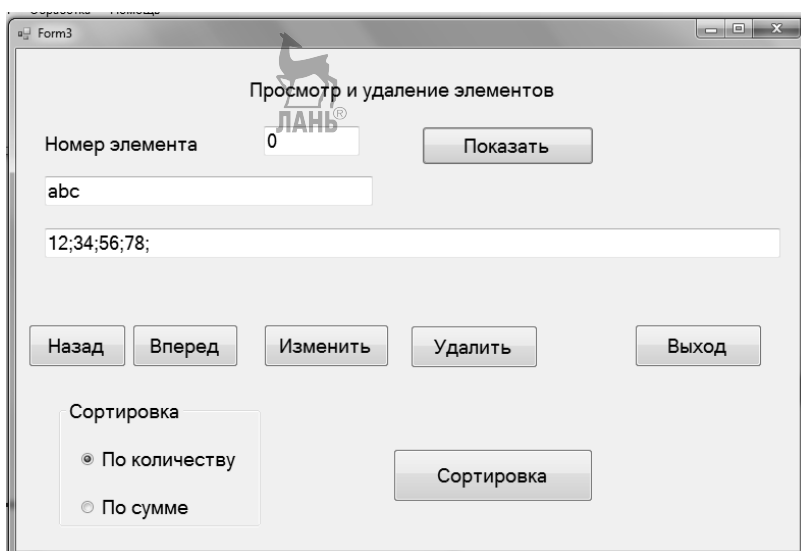


Рис. 5.16

Кнопки «Назад» и «Вперед» предназначены для передвижения по элементам списка. Их реализации:

```
private void button3_Click(object sender, EventArgs e)
```

```
{ // назад
    elem el;
    if (num == 0)
        MessageBox.Show("Первый элемент");
    else
    {
        num--;
        textBox1.Text = "" + num;
        el = Form1.lst1[num];
        textBox2.Text = el.s;
        string[] temp = new
        string[Form1.lst1[num].mas.Length];
        string s1 = "";
        foreach (int x in Form1.lst1[num].mas)
            s1 += x + ";";
        textBox3.Text = s1;
    }
}
```

```
private void button2_Click(object sender, EventArgs e)
```

```
{ // вперед
    elem el;
    if (num >= Form1.lst1.Count - 1)
        MessageBox.Show("Уже последний элемент");
    else
    {
        num++;
        textBox1.Text = "" + num;
        el = Form1.lst1[num];
        textBox2.Text = el.s;
        string[] temp = new
        string[Form1.lst1[num].mas.Length];
        string s1 = "";
        foreach (int x in Form1.lst1[num].mas)
            s1 += x + ";";
        textBox3.Text = s1;    }    }
```

Кнопка «Изменить» заменит текущий элемент списка элементом, набранным на экране. Это реализовано путем удаления старого текущего элемента и

добавления нового, потому что при этом может измениться количество чисел в массиве.

```
private void button5_Click(object sender, EventArgs e)
{
```

```
    string s1, s2;
    elem el;
    string[] temp;
    int[] arr;
```

```
    Form1.lst1.RemoveAt(num); // Удаление текущего элемента
```

```
    s1 = textBox2.Text;
```

```
    s2 = textBox3.Text;
```

```
    temp = s2.Split(';');
```

```
    temp = temp.Where(q => q != "").ToArray();
```

```
    arr = new int[temp.Length];
```

```
    for (int i = 0; i < arr.Length; i++)
```

```
        arr[i] = Convert.ToInt32(temp[i]);
```

```
// Добавление нового элемента
```

```
    el = newelem(s1, arr);
```

```
    Form1.lst1.Add(el);
```

```
}
```

Кнопка «Удалить» удаляет текущий элемент списка:

```
private void button6_Click(object sender, EventArgs e)
{
```

```
    Form1.lst1.RemoveAt(num); }
```

Кнопка «Сортировка» сортирует элементы списка по одному из двух критериев, заданному радиокнопками.

```
private void button7_Click(object sender, EventArgs e)
{ //sort
```

```
    elem el;
```

```
    string s1, s2;
```

```
    if (radioButton1.Checked)
```

```
// Сортировка по количеству положительных элементов
```

```
    Form1.lst1 = Form1.lst1.OrderBy(p =>
```

```
        p.mas.Where(q => q > 0).Count()).ToList();
```

```
    if (radioButton2.Checked)
```

```
        // Сортировка по сумме элементов
```

```
    Form1.lst1 = Form1.lst1.OrderBy(p =>
```

```
        p.mas.Sum()).ToList();
```

```
    textBox1.Text = "" + 0;
```

```
    num = 0;
```

```
    el = Form1.lst1[num];
```

```
    textBox2.Text = el.s;
```

```
//Покажем на экране первый элемент списка
//после сортировки
string[] temp = new string[Form1.lst1[num].mas.Length];
s1 = "";
foreach (int x in Form1.lst1[num].mas)
    s1 += x + ";";
textBox3.Text = s1;    }
```

5.7. Дополнительные средства создания интерфейса пользователя

Рассмотрим некоторые дополнительные возможности создания интерфейса пользователя, не использованные в приведенных примерах.

Текстовое поле с маской ввода *maskedTextBox*. Маска ввода накладывает ограничения на вводимые в такое поле символы и, таким образом, служит средством для уменьшения ошибок при наборе символов в такое поле. Для определения маски ввода можем использовать свойство *mask* или, нажав на маленький треугольник в правом верхнем углу поля, выбрать там пункт *SetMask*. В обоих случаях открывается диалоговое окно, в котором уже задан ряд масок.

1. Выберем короткую дату: тогда в это поле можно вести дату в виде дд.мм.гггг. Правда, проверку корректности даты сама маска не выполняет. При использовании приведенного ниже преобразования некорректные даты (например 29.02.2017) вызывают прерывание.

```
DateTime t1;
//стандартный класс представления даты и времени
t1 = Convert.ToDateTime(maskedTextBox1.Text);
label1.Text = t1.ToShortDateString();
```

2. Маска *aaaa* позволяет ввести до 4 произвольных символов. Таким образом можно обеспечить, чтобы пользователь не ввел больше символов, чем длина строки.
3. Маска *###.#* позволяет ввести только цифры, перед ними могут быть пробелы и знак числа. Приведенная маска позволяет ввести числа от -99.9 до 999.9. Маска *aa####aaa* требует ввода последовательности 2 символа, 2 цифры, 3 символа.

С возможными символами для масок можете ознакомиться через Help. Маски могут применяться и при выводе информации, но для управления внешнего вида записи при выводе можно использовать и функцию *ToString()*, которая была уже раньше нами применена.

Поле ввода/вывода одной даты *DateTimePicker*. Имеет выпадающий список, позволяющий выбрать дату из календаря.


```

{
    DateTime t2;
    t2 = dateTimePicker1.Value; //ввод даты
    label1.Text = t2.ToString();
                                //вывод даты в виде строки
    t2 = new DateTime(2000, 03, 21);
    dateTimePicker1.Value = t2; //вывод даты
}

```

Поле ввода диапазона дат *MonthCalendar*. Будет показан календарь, в котором пользователь может выделить диапазон дат (естественно, диапазон может состоять из одной даты).

```

{
    DateTime t1, t2;
    int delta;
    t1 = monthCalendar1.SelectionStart;
    t2 = monthCalendar1.SelectionEnd;
//Начало и конец выделенного диапазона
    delta = (t2 - t1).Days; //Длина интервала
    label1.Text = "" + delta;
//Представление дат на форме
    label2.Text = t1.ToLongDateString();
    label3.Text = t2.ToLongDateString();
}

```

При выборе лишь одной даты использование обоих свойств дает одинаковый результат. Приведем список некоторых свойств этого компонента:

- *MinDate* — допустимая самая ранняя дата при выделении диапазона.
- *MaxDate* — допустимая самая поздняя дата при выделении диапазона.
- *AnnuallyBoldedDates* — содержит набор дат, которые будут отмечены жирным в календаре для каждого года.
- *BoldedDates* — содержит набор дат, которые будут отмечены жирным (только для текущего года).
- *MonthlyBoldedDates* — содержит набор дат, которые будут отмечены жирным для каждого месяца.

Перечень дат для трех последних свойств может быть легко определен с помощью диалоговых окон.

Поле для выбора значения из списка реализуется с помощью инструментов *CheckedListBox*, *ListBox* и *ComboBox*. Их применение целесообразно при необходимости значения (особенно символьной строки) из числа заданных, и никакие другие значения кроме них не допускаются. Это позволит существенно снизить вероятность ошибки ввода символьных строк, особенно длинных. Например, направлений, по которым ведется подготовка в вузе. Свойства *CheckedListBox*, *ListBox* и *ComboBox* очень похожи, поэтому рассмотрим их вместе. Начинаем с отличий. Принципиальным отличием является то, что *ComboBox* и

ListBox могут получить перечень возможных значений из базы данных, *CheckedListBox* такой возможности не имеет. Все они могут работать и с перечнем возможных значений, заданном при проектировании. Подключение к базе данных нами рассмотрено в соответствующем разделе данного пособия. Отличаются они и по внешнему виду: *CheckedListBox* показывает значения в выпадающем списке, *ComboBox* и *ListBox* показывают их одновременно на экране, при нехватке места добавляют скроллинг. Вывод простой: при длинном перечне возможных значений предпочтительнее *ComboBox*.

Основные их свойства:

Items — перечень значений, задается по одному в строке.

Sorted — сортировать значения.

SelectedItem — выбранное значение.

SelectedIndex — выбранный индекс.

Задаем первоначально выбранные значения.

```
cLB1.SelectedIndex = 1;
```

```
cmbB1.SelectedIndex = 0;
```

```
{//Определение значения и номера выбранной строки
```

```
    string s1;
```

```
    int num;//CheckedListBox
```

```
    s1 = cLB1.SelectedItem.ToString();
```

```
    num = cLB1.SelectedIndex;
```

```
    label1.Text = s1+" "+num;
```

```
}
```

```
{
```

```
    string s2;
```

```
    int num; // ComboBox
```

```
    num = cmbB1.SelectedIndex;
```

```
    s2 = cmbB1.SelectedItem.ToString();
```

```
    label1.Text = s2 + " " + num;
```

```
}
```

```
{
```

```
    string s1;
```

```
    int num;
```

```
    s1 = listBox1.SelectedValue.ToString();
```

```
    num = listBox1.SelectedIndex;
```

```
    label1.Text = s1 + " " + num;}
```

SaveDialog* и *OpenDialog позволяют определить местонахождение файла при его создании или обработке, другими словами, они связывают файловую переменную в программе с файлом на внешнем носителе. В таких программах, как *Word* и/или *Excel*, открытие файла означает начало его обработки. На язы-

ках программирования можно лишь определить местонахождение файла, но его фактическое открытие и обработку придется самому запрограммировать. Их общие свойства:

- *DefaultExt*: устанавливает расширение файла, которое добавляется по умолчанию, если пользователь ввел имя файла без расширения, при открытии файла в первую очередь предлагает открывать файл с этим расширением. В нашем случае *.txt.
- *AddExtension*: при значении true добавляет к имени файла расширение при его отсутствии. Расширение берется из свойства *DefaultExt* или *Filter*.
- *CheckFileExists*: если имеет значение true, то проверяет существование файла с указанным именем.
- *FileName*: возвращает полное имя файла (включая путь), выбранного в диалоговом окне.
- *Filter*: задает фильтр файлов, благодаря чему в диалоговом окне можно отфильтровать файлы по расширению. Для получения фильтров текстовый файл *.txt и все файлы *.* придется писать:
Текстовый файл (*.txt)| *.txt|Все файлы(*.*)|*.*
- *InitialDirectory*: устанавливает каталог, который отображается при первом вызове окна, в нашем случае D:\Files.
- *Title*: заголовок диалогового окна, в нашем случае *OpenFile* и *SaveFile*.

Рассмотрим это на примере, форма которого во время создания файла приведена на рис. 5.17.

The screenshot shows a Windows application window titled "Form1". Inside the window, there are several controls: a "Create File" button at the top left; three input fields labeled "String", "Int", and "Double" with values "aaaaa", "25", and "12,3" respectively; an "Add" button to the right of the "Int" field; a "Close Add" button below the "Add" button; a "Read" button in the center; an "Open" button at the bottom left; and a "Close Read" button at the bottom right. A watermark logo with the text "ЛАНЬ" is visible in the center of the form.

Рис. 5.17

UsingSystem.IO; // Должно быть при работе с файлами



```
namespace Dialog1
{
    public partial class Form1 : Form
    {
        // Объявление переменных для работы с файлами
        FileStream fs1, fs2;
        BinaryWriter bw1;
        BinaryReader br1;
        public Form1()
        {
            InitializeComponent();

            private void button2_Click(object sender, EventArgs e)
            {
                // Определение местонахождения и имени
                // создаваемого файла.
                strings1;
                // Запуск диалогового окна SaveDialog и проверка,
                // закрыли ли его кнопкой OK или CANCEL
                if (saveFileDialog1.ShowDialog() ==
                    DialogResult.Cancel)
                    MessageBox.Show("Файл не открыт");
                else
                {
                    s1 = saveFileDialog1.FileName;
                    fs1 = new FileStream(s1, FileMode.Create);
                    button3.Enabled = true;
                    button4.Enabled = true;
                    bw1 = new BinaryWriter(fs1);
                }
            }

            private void button3_Click(object sender, EventArgs e)
            {
                // Запись данных в файл
                dan mydan;
                mydan.s = textBox1.Text;
                mydan.k = Convert.ToInt32(maskedTextBox1.Text);
                mydan.d = Double.Parse(maskedTextBox2.Text);
                bw1.Write(mydan.s);
                bw1.Write(mydan.k);
                bw1.Write(mydan.d);
            }
        }
    }
}
```

```

        textBox1.Clear();
        maskedTextBox2.Clear();
        maskedTextBox1.Clear();
    }

    private void button4_Click(object sender, EventArgs e)
    { //Заккрытие файла после создания
        fs1.Close();
        button3.Enabled = false;
        button4.Enabled = false;
    }

    private void button1_Click(object sender, EventArgs e)
    { //Открытие файла для чтения через OpenFileDialog.
        string s1;
        if (openFileDialog1.ShowDialog() == DialogResult.OK)
        {
            s1 = openFileDialog1.FileName;
            fs2 = new FileStream(s1, FileMode.Open);
            button5.Enabled = true;
            button6.Enabled = true;
            br1 = new BinaryReader(fs2);
            // Чтение первой записи из файла.
            label4.Text = br1.ReadString();
            label5.Text = "" + br1.ReadInt32();
            label6.Text = br1.ReadDouble().ToString("F1");
        }
        else
            MessageBox.Show("Файл не открыт");
    }

    private void button5_Click(object sender, EventArgs e)
    { //Чтение очередной записи и проверка,
        //не достигнут ли конец?
        dan mydan;
        mydan.s = br1.ReadString();
        mydan.k = br1.ReadInt32();
        mydan.d = br1.ReadDouble();
        label4.Text = mydan.s;
        label5.Text = "" + mydan.k;
        label6.Text = mydan.d.ToString("F1");
    }

```

```

        if (br1.PeekChar() == -1)
            MessageBox.Show("Данные кончились");
    }

private void button6_Click(object sender, EventArgs e)
{
    // Закрытие файла после чтения
    fs2.Close();
    button5.Enabled = false;
    button6.Enabled = false;
}
}
}
struct dan
{
    // Структура записи файла
    public string s;
    public int k;
    public double d;
}
}

```



5.8. Вывод на печать

Нередко возникает необходимость вывода результатов вычислений на печать. *Visual Studio* представляет для этого широкие возможности: можно печатать текст, рисунки, графики; управлять расположением текста по строкам и по страницам. Ограничимся рассмотрением вывода на одну страницу A4 (в книжном или альбомном формате) текстов, расположенных на форме в компонентах *textBox* (одно- и многострочный варианты) и *richTextBox*. Форма представлена на рис. 5.18.

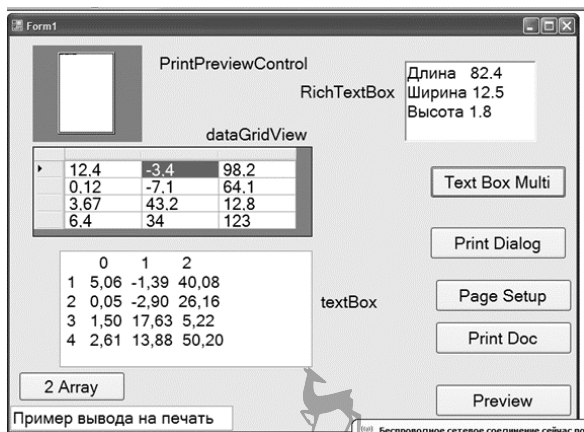


Рис. 5.18

Используемые визуальные компоненты видны на рисунке. Для организации вывода на печать требуются следующие компоненты *Visual Studio* (правда, как видно из дальнейшего рассмотрения, не обязательно ВСЕ они должны ВСЕГДА присутствовать):

printDocument — выполняет непосредственно вывод на печать, в реализации его события *PrintPage* определяется вывод. Его имя пусть останется *printDocument1*;

pageSetupDialog — позволяет уточнить параметры страницы и запускать печать. Его свойству *Document* дадим значение *printDocument1*;

printDialog — традиционное окно *Windows* для выбора принтера и уточнения некоторых параметров печати, позволяет запускать печать. Его свойству *Document* дадим значение *printDocument1*;

printPreviewDialog — традиционное окно *Windows* для предварительного просмотра выводимого на печать текста. Его свойству *Document* дадим значение *printDocument1*;

printPreviewControl — показывает на самой форме макет вывода. Его свойству *Document* дадим значение *printDocument1*.

В число подключенных библиотек добавим

```
using System.Drawing.Printing;
```

Программная реализация:

```
using System.Drawing.Printing;
```

```
namespace PrintDlg
```

```
{
```

```
    public partial class Form1 : Form
```

```
{
```

```
    public Form1()
```

```
    {
```

```
        InitializeComponent();
```

```
        //Задаем количество строк и столбцов
```

```
        //для двумерного массива
```

```
        dataGridview1.ColumnCount = 3;
```

```
        dataGridview1.RowCount = 4;
```

```
    }
```

```
    private void button1_Click(object sender, EventArgs e)
```

```
    { // Запускаем диалоговое окно PrintDialog
```

```
        printDialog1.ShowDialog();
```

```
    }
```

```
    private void button2_Click(object sender, EventArgs e)
```

```
    { //Запускаем диалоговое окно PageSetup
```

```
        pageSetupDialog1.ShowDialog();
```

```
    }
```

```

private void button3_Click(object sender, EventArgs e)
{
    // Print Document
    printDocument1.Print();
}

private void button4_Click(object sender, EventArgs e)
{
    //Вывод на печать PreviewPage
    printPreviewDialog1.ShowDialog();
}

private void printDocument1_PrintPage(object sender, PrintPageEventArgs e)
{
    //Релизация события printDocument1_PrintPage
    Graphics graph = e.Graphics;
    Font fn1 = this.Font;
    // задаем характеристики шрифта для печати,
    // совпадают с Form1
    graph.DrawString(richTextBox1.Text, fn1, Brushes.Black, 0, 80);
    // Вывод на печать, параметры: выводимое поле, шрифт,
    // цвет, координаты начала печати в пикселях
    graph.DrawString(textBox1.Text, fn1, Brushes.Black, 0, 0);
    graph.DrawString(textBox2.Text, fn1, Brushes.Black, 0, 500);
}

private void button5_Click(object sender, EventArgs e)
{
    // Подготовка многострочного текста для печати
    string[] mas = { "Длина 82.4", "Ширина 12.5", "Высота 1.8" };
    richTextBox1.Lines = mas;
}

private void button6_Click(object sender, EventArgs e)
{
    // Подготовка двумерного массива для печати
    double[,] mas = new double[dataGridView1.RowCount,
                                dataGridView1.ColumnCount];
    // Чтение массива из dataGridView1
    for (int i = 0; i <= mas.GetUpperBound(0); i++)
        for (int j = 0; j <= mas.GetUpperBound(1); j++)
            mas[i, j] =
                Convert.ToDouble(dataGridView1.Rows[i].Cells[j].Value);
    // Вычисления
    for (int i = 0; i <= mas.GetUpperBound(0); i++)
        for (int j = 0; j <= mas.GetUpperBound(1); j++)


```



```

        mas[i, j] = mas[i, j] / 2.45;
// Подготовка массива для печати,
// с номерами строк и столбцов
    string temp = " ";
    string []str=newstring [mas.GetLength(0)+1];
    for (int i = 0; i <= mas.GetUpperBound(1); i++)
        temp += (" " + i + " ");
    str[0] = temp;
    for(int i=1;i<=mas.GetUpperBound(0)+1;i++)
    {
        temp=" "+i+" ";
        for(int j=0;j<=mas.GetUpperBound(1);j++)
            temp+=(" "+mas[i-1,j].ToString("F2"));
        str[i] = temp;
    }
    textBox2.Lines = str;
}
}
}

```



Более подробно работа с двумерным массивом была рассмотрена выше.

5.9. Рефакторинг

Под рефакторингом понимают изменение кода программы с целью повышения его качества без изменения ее функционала. Не рекомендуют одновременно менять функциональные возможности и проводить рефакторинг, потому что тогда «ситуация может выходить из-под контроля». Вопросы рефакторинга подробно рассмотрены в [11], более сжатое изложение можно найти в [4].

Цели проведения рефакторинга:

1. Облегчение понимания программного комплекса в целом. В ходе разработки придется внести различные изменения и усовершенствования, в результате которых может теряться стройность системы в целом. рефакторинг позволяет исправить ситуацию.
2. Рефакторинг позволяет делать текст программы более понятным.
3. Рефакторинг способствует нахождению ошибок в результате повышения качества кода.

Рефакторинг является существенной составной частью при разработке программ по технологии экстремального программирования.

Для успешного рефакторинга необходимо:

1. Применять автоматизированное тестирование для проверки программ после каждого изменения.
2. Сохранить текст программы без изменений в первоначальном виде, в случае неудачи можно к нему вернуться.
3. Применять итерационную разработку.
4. Регулярно проводить рефакторинг.

Рассмотрим средства рефакторинга в среде *Microsoft Visual Studio*. Там предусмотрено следующее:

- переименование;
- упорядочение параметров;
- удаление параметра;
- инкапсуляция поля;
- создание интерфейса;
- извлечение метода.

Рассмотрим их по очереди.

Переименование. Можно переименовать все введенные программистом имена, независимо от того, что они обозначают (переменные, массивы, функции). Для этого выделим имя, которое необходимо менять, активизируем контекстное меню, далее пункт «Рефакторинг» и «Менять имя». В ответ предлагают ввести новое имя и спросят, надо ли менять это имя в комментариях и строках. Далее откроется диалоговое окно, где будет показано, где именно имя будет изменено, с помощью кнопок выбора можно часть переименований отменить. Естественно, это допустимо только для комментариев и строк, иначе один и тот же объект будет в разных частях программы назван по-разному. При подтверждении переименования старое имя везде в своей области действия, включая классы-наследники, будет заменено на новое. Пусть имеется следующий класс:

```
class clb
{
    double[] mas;
    double x, y;
    public clb(int m25)
    {
        mas = new double[m25];
        x = 3.5;
    }
    public void inpt()
    {
        for (int i = 0; i < mas.Length; i++)
        {
            // Ввод массива mas
            Console.WriteLine("mas["+i+"]=");
            mas[i] = Convert.ToDouble(Console.ReadLine());
        }
    }
    public int KolC(double n, int m)
    {
        return mas.Where(p => p > n).Count();
    }
}
```

```

    public void otpt()
    {
        Console.WriteLine("x=" + x);
    } }

```

Проведем по очереди два переименования: заменим имя переменной *x* на *x25* и имя массива *mas* на *arr*. В каком месте мы выделим переменную для переименования, не имеет значения. Обратите внимание, что оба имени присутствуют кроме текста программы в комментариях и символьных строках. Результат приведен ниже:

```

class clb
{
    double[] arr;
    double x25, y;
    public clb(int m25)
    {
        arr = new double[m25];
        x25 = 3.5;
    }
    public void inpt()
    {
        for (int i = 0; i < arr.Length; i++)
        {
            // Ввод массива arr
            Console.WriteLine("arr["+i+"]=");
            arr[i] = Convert.ToDouble(Console.ReadLine());
        }
    }
    public int KolC(double n, int m)
    {
        return arr.Where(p => p > n).Count();
    }
    public void otpt()
    {
        Console.WriteLine("x25=" + x25);
    }
}

```

Упорядочение параметров. При желании изменить очередность параметров функции можно выделить их в самой функции или в любом месте ее вызова и через контекстное меню выбрать «Рефакторинг» и «Очередность параметров». В открывшем диалоговом окне с помощью стрелок можно определить новую очередность параметров, после утверждения их очередность будет изменена везде, где эта функция встречается.

Удаление параметра. Выделим список параметров функции в самой функции или в любом месте ее вызова, откроем описанным выше способом диалоговое окно, где будут показаны все параметры. Выберем из них нужный и удалим его. Удаление будет автоматически распространено.

Инкапсуляция поля. Под этим термином понимается создание свойства, представляющее переменную и/или массив. При инкапсуляции по очереди переменной *y* и массива *arr* будут созданы следующие свойства:

```
public double[] Arr//имя свойства по умолчанию
{
    get { return arr; }
    set { arr = value; }
}
public double PropY
{//имя свойства выбрано программистом
get { return y; }
    set { y = value; }
}
```



Создание интерфейса. Для создания интерфейса выделим компонент класса, активизируем контекстное меню, дойдем до пункта контекстного меню «Создать интерфейс». Откроется диалоговое окно, в котором мы должны выбрать включаемые в создаваемый интерфейс компоненты класса и выбрать ему имя, если не устраивает имя по умолчанию. Будет создан интерфейс и класс, из которого он создавался, станет его наследником. В нашем случае может быть создан следующий интерфейс:

```
namespace Refactor1
{
interface Iclb
{
    double[] Arr { get; set; }
    void inpt();
    void otpt();
}
}
class clb : Refactor1.Iclb
{ .... }
```



Извлечение метода. Пусть в нашем классе имеется функция:

```
public void fun1(double c1,outdouble res)
{
    res = 0;
    for (int i = 0; i < mas.Length; i++)
        if (mas[i] > c1)
        {
```

```

        res += mas[i];
        mas[i] += 2.4;
    }
    else
        mas[i] -= 2.4;
}

```

Выделим целиком цикл и выберем рефакторинг «Извлечение функции». Результат приведен ниже:

```

public void fun1(double c1, out double res)
{
    res = 0;
    res = NewMethod(c1, res);
}

```

```

private double NewMethod(double c1, double res)
{ //Имя выбрано по умолчанию, его можно было изменить.
    for (int i = 0; i < mas.Length; i++)
        if (mas[i] > c1)
        {
            res += mas[i];
            mas[i] += 2.4;
        }
        else
            mas[i] -= 2.4;
    return res;
}

```

Рассмотрим некоторые другие методы рефакторинга, не реализованные в *Visual Studio*.

Подъем и/или спуск метода по иерархии классов. Если одинаковый метод используется в нескольких классах-наследниках, то может оказаться целесообразным его перенос в класс-предок. Наоборот, если метод класса-предка применяется только в одном классе-наследнике, то, скорее всего, следует его туда перенести.

Введение дополнительного класса-предка, куда собраны общие части (как данные, так и методы) классов-наследников. Введение дополнительного класса, куда собраны из исходного образующие единое целое данные и методы их обработки.

Замена условного оператора полиморфизмом. Вместо условного оператора в классе-предке иметь реализованные виртуальными функциями в классах-наследниках отдельные ветви этого условного оператора.

5.10. Работа в WPF

WPF (Windows Presentation Foundation) — это новая технология разработки пользовательского интерфейса. При разработке по технологии *Windows Forms* размеры форм строго заданы во время проектирования, и, хотя во время работы приложения мы можем их изменить, это ничего не даст, мы лишь увеличиваем пустое пространство формы. *WPF* использует усовершенствованную графику, что позволяет плавно менять размеры окон во время работы с пропорциональным изменением расположения нанесенных на нее компонентов интерфейса. Кроме того, набор средств создания интерфейсов пользователя в *WPF* и их свойств существенно шире. При создании многооконных приложений в *WPF* туда можно включить и классические формы. Уточним терминологию: при разработке по *WPF* мы говорим об **окнах** (а не о формах, как раньше при обсуждении технологии *Windows Form*). Подробно *WPF* рассмотрено в [12]. Создание нового проекта *WPF* ничем не отличается от ранее рассмотренного, только выберем нужный тип.

При создании проекта *WPF*, кроме знакомых нам частей, появится новый: описание интерфейса на языке *XAML*. Это дает возможность определить свойства компонентов тремя способами:

- через окно свойств;
- программным путем;
- в *XAML*.

При создании нового проекта его *XAML* имеет следующий вид:

```
<Window x:Class="WpfSimple.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="350" Width="525">
<Grid>

</Grid>
</Window>
```

Первые 3 строки содержат служебную информацию, мы их рассматривать не будем. Четвертая строка содержит заголовок окна (*Title="MainWindow"*) и его размеры, их можно здесь и менять. Приложение — это окно, содержащее пока одну клетку (*Grid*). Обратите внимание на структуру описания интерфейса, она полностью соответствует требованиям языка *XML* от открывающей скобки *<...>* до закрывающей *</...>*, их пересечение недопустимо. Язык *XAML* чувствителен к регистру, допущенные ошибки среда обрабатывает аналогично любым синтаксическим ошибкам в программе.

Заложенная в *WPF* возможность изменения размеров окна с пропорциональным изменением расположения компонентов достигается лишь при условии, что наша пока единственная клетка будет разделена горизонтально и вертикально на «мини-клетки», они видны только во время проектирования. Для их создания делаем щелчок мышью на нашем окне, убедимся, что в окне пока-

заны свойства *Grid* и определим значения его свойств *ColumnDefinitions* и *RowDefinitions*. Окно свойств содержит полосу поиска, что существенно облегчает их нахождение. Введем туда *col...*, и нужные нам свойства появятся. Они имеют собственные диалоговые окна, через которые добавим 4 столбца и 3 строки.

5.10.1. Простейший пример

Рассмотрим работу с компонентами интерфейса на примере командной кнопки *Button*, которая должна всегда быть расположена в правом нижнем углу окна. Занесем ее в нужную клетку и обратимся к ее свойствам. Первое отличие от *Windows Forms* — компоненты интерфейса не имеют имен по умолчанию. Если мы в своих программах на них ссылаться не будем, то это и не важно. Дадим нашей кнопке имя *Close1*. Видимый на экране на кнопке текст является теперь значением свойства *Content* (у нас *Close*). В окне поиска свойств введем *align...*, и откроется возможность определения выравнивания самой кнопки и текста в нем. Поставим выравнивание кнопки вниз и направо. Выравнивание текста менять не будем (останется в центре кнопки). Сбросим поиск выравнивания и наберем для поиска *Margin*, это свойство показывает расстояние до границ «мини-клетки», в которой находится *Button*. Так как мы задали выравнивание вниз и направо, то важны только значения расстояний до этих границ, зададим их равными нулю. Осталась лишь реализация кнопки. Переключаем окно ее свойств на события, выберем событие *Click*, делаем на нем двойной щелчок и в заготовку функции пишем знакомое *Close()*; запустим приложение и, изменяя размер окна, убедимся, что наша кнопка всегда будет в нижнем правом углу.

Поставим на наше окно поле *TextBox* и расположим его в верхнем левом углу, но так, чтобы он занимал 3 «мини-клетки» по горизонтали. Дадим ему имя *txt1*, видимый на экране текст задается, как и раньше, в свойстве *Text*. Мы уже знаем, как обеспечить его расположение. Дополнительно: свойство *Grid.Column* определяет номер столбца расположения его левой границы; *Grid.Row* — номер строки расположения его верхней границы; *Grid.ColumnSpan* — сколько столбцов оно занимает; *Grid.RowSpan* — сколько строк занимает. Значения всех свойств можно менять визуально на самом окне, в тексте на *XAML* или в окне свойств. Результат от этого не зависит.

Нанесем на окно:

- для вывода результатов компонент *Label*, имя *Out1*, видимый на экране текст задается в его свойстве *Content*;
- для запуска обработки массива-строки командную кнопку, имя *Row1*, текст на ней *Row*;
- для ввода массива-столбца еще один *textBox*, имя *txt2*, в кнопку выбора свойства *AcceptsReturn* ставим галочку;
- для запуска обработки массива-столбца создадим кнопку, имя *Col1*, текст на ней *Column*.

Кроме описанного выше, имеется еще один способ реализации командной кнопки. Найдем в *XAML* ее описание, поставим курсор мыши после любого свойства, нажмем на пробел и из выпадающего списка свойств и событий выбираем *Click* и ставим ему в соответствие новую функцию. Внешний вид формы представлен на рис. 5.19.

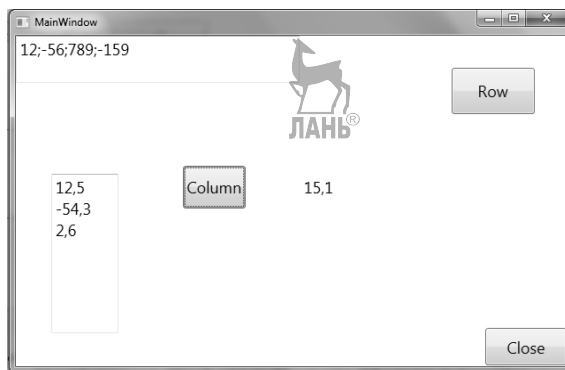


Рис. 5.19

Текст на *XAML*:

```
Window x:Class="WpfSimple1.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="400" Width="525" FontSize="18">
<Grid>
<Grid.RowDefinitions>
<RowDefinition/> //определение строк, интервалы равные
<RowDefinition/>
<RowDefinition/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
//определение столбцов, интервалы разные
<ColumnDefinition Width="126*"/>
<ColumnDefinition Width="132*"/>
<ColumnDefinition Width="130*"/>
<ColumnDefinition Width="129*"/>
</Grid.ColumnDefinitions>

<Button x:Name="close1" Content="Close" Grid.Column="3" Horizontal-
Alignment="Right" Height="42" Grid.Row="2" VerticalAlignment="Top"
Width="92" Margin="0,81,0,0" Click="close1_Click" HorizontalContentAlign-
ment="Center" RenderTransformOrigin="-1.236,-2.725"/> //свойства
Button
```



```
<TextBox x:Name="txt1" Grid.ColumnSpan="3" HorizontalAlignment="Left"
        Height="53" Margin="0" TextWrapping="Wrap" VerticalAlign-
ment="Top" Width="321"/> //свойства TextBox
```

```
<Label x:Name="Out1" Content="Answer" Grid.Column="2" Horizontal-
Alignment="Left" Height="49" Margin="10,29,0,0" Grid.Row="1" Vertical-
Alignment="Top" Width="110"/>
//свойства Label
```

```
<Button x:Name="Row1" Content="Row" Click="Row1_Click"
Grid.Column="3" HorizontalAlignment="Left" Height="52" Mar-
gin="25,36,0,0" VerticalAlignment="Top" Width="94"/>
```

```
<TextBox x:Name="txt2" HorizontalAlignment="Left" Height="179" Mar-
gin="40,31,0,0" Grid.Row="1" TextWrapping="Wrap" VerticalAlign-
ment="Top" Width="76" Grid.RowSpan="2" AcceptsReturn="True"/>
```

```
<Button x:Name="Col1" Content="Column" Grid.Column="1"
Click="Col1_Click" HorizontalAlignment="Left" Height="48" Mar-
gin="37,22,0,0" Grid.Row="1" VerticalAlignment="Top" Width="71" Render-
TransformOrigin="-3.455,0.049"/>
```

```
</Grid>
</Window>
```

Реализация двух командных кнопок:

```
private void Row1_Click(object sender, RoutedEventArgs e)
{ // Обработка массива-строки, описание приведено выше.
    string s;
    string[] dan;
    int[] mas;
    s = txt1.Text;
    dan = s.Split(';');
    dan = dan.Where(p => p != "").ToArray();
    mas = new Int32[dan.Length];
    for (int i = 0; i < mas.Length; i++)
        mas[i] = Convert.ToInt32(dan[i]);
    int kol = mas.Where(p => p > 0).Count();
```

```

    Out1.Content = "" + kol;
}

private void Col1_Click(object sender, RoutedEventArgs e)
{ //Описание массива-столбца
    List<string> lst1 = new List<string>();
    int m;
    string y;
    double[] arr;
    double summa;
    for (int i=0;;i++)
    {
        y = txt2.GetLineText(i);
        if(y=="\r\n")continue;
// Символы перевода строки входят в y,
//если строка состоит только из них, то она пуста.
        if (y == "") break;
        m=y.IndexOf("\r\n");
        if(m>-1) y=y.Substring(0,y.Length-2);
//символы "\r\n" должны быть удалены.
        lst1.Add(y);
        if (m < 0) break;
    }
    arr = new double[lst1.Count];
    for (int i = 0; i < arr.Length; i++)
        arr[i] = Double.Parse(lst1[i]);
    summa = arr.Where(p => p > 1.5).Sum();
    Out1.Content = summa.ToString("F1");
}

```

5.10.2. Работа с двумерным форматизированным массивом

Для представления двумерного массива в окне используем знакомый нам компонент *DataGrid*, дадим ему имя *dgl*. Для внутреннего представления используем класс *DataTable* с именем *tbl*. Необходимо добавить

```
using System.Data;
```

Для показа возможностей *WPF* изменим внешний вид формы, рис. 5.20. Оставим читателю поиск ответа на вопросы: «Какими свойствами можно это сделать? Какие еще возможности для изменения внешнего вида имеются?»

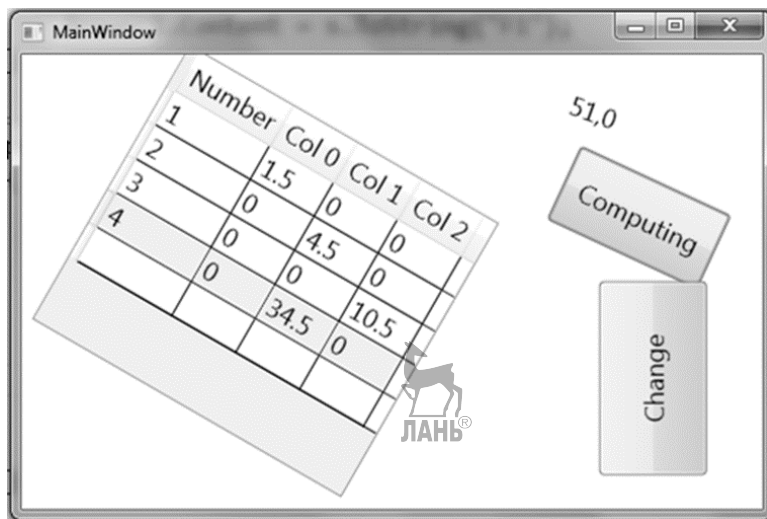


Рис. 5.20

Реализация обработки похожа на приведенную выше для *Windows Forms*, но имеются и отличия.

```
public partial class MainWindow : Window
```

```
{
```

```
    DataSet ds;
```

```
    DataTable tab1;
```

```
    void InitGrid(int n, int m)
```

```
{
```

```
        tab1 = new DataTable();
```

```
        DataColumn x1 = tab1.Columns.Add("Number",  
                                          typeof(Int32));
```

```
        DataColumn x2;
```

```
        for (int i = 0; i < n; i++)
```

```
{
```

```
            x2 = tab1.Columns.Add("Col " + i, typeof(Double));
```

```
            x2.DefaultValue = 0;
```

```
}
```

```
        DataRow dr;
```

```
        for (int i = 0; i < m; i++)
```

```
{
```

```
            dr = tab1.NewRow();
```

```
            tab1.Rows.Add(dr);
```

```
            tab1.Rows[i][0] = (Int32)(i + 1);
```

```
}
```

```
        x1.ReadOnly = true;
```

```

ds.Tables.Add(tab1);
//Добавим созданную таблицу в DataSet
}
public MainWindow()
{
    InitializeComponent();
ds = new DataSet();
InitGrid(3, 4);
dg1.ItemsSource = ds.Tables[0].DefaultView;
    // Связываем tab1 и dg1.
}

private void Comp1_Click(object sender, RoutedEventArgs e)
{ // Вычисление
    double s = 0;
    foreach (DataRow x in tab1.Rows)
        for (int j = 1; j < tab1.Columns.Count; j++)
            s += Convert.ToDouble(x[j]);
    Answer1.Content = s.ToString("F1");
}

private void Chng1_Click(object sender, RoutedEventArgs e)
{ // Изменение данных
    double x;
    for(int i=0;i<tab1.Rows.Count;i++)
        for(int j=1;j<tab1.Columns.Count;j++)
        {
            x = (double)tab1.Rows[i][j];
            x *= 1.5;
            tab1.Rows[i][j] = x;
        }
    }
}

```

5.10.3. Работа со списком

Создаем список с элементами следующей структуры (Наличие `{get;set;}` обязательно):

```

class Person    {
    public string name{get;set;}
    public int age {get;set;}
    public bool member {get;set;} }

```

на окно занесем знакомый нам компонент *dataGridView*, дадим ему имя *dgl* и выравниваем. Дополнения класса *MainWindow* и его конструктора:

```
public partial class MainWindow : Window
{
    List<Person>lst1;
    Person temp;
    public MainWindow()
    {
        InitializeComponent();
        lst1=new List<Person>();
        dg1.ItemsSource = lst1;
    } ...
```

Внешний вид окна представлен на рис. 5.21. В *WPF* меню может находиться в любом месте окна, для разнообразия пусть оно будет внизу.

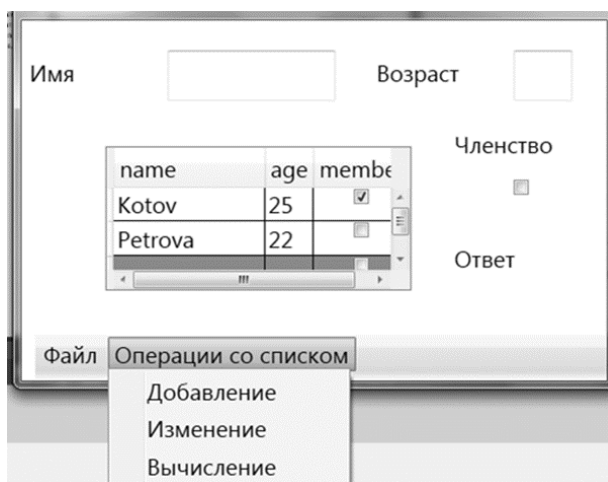


Рис. 5.21

Для создания меню перенесем на окно компонент *Menu* и расположим его внизу окна, его область на окне должна быть достаточна для размещения всех пунктов горизонтального меню. Для создания пунктов горизонтального меню выделим его на экране, выделим свойство *Items* и диалоговое окно создания меню. Определим тип пунктов меню *MenuItem*s («классическое» меню, имеются и другие варианты) и создаем горизонтальное меню. Тексты его пунктов пишем в свойство *Header*. Для создания пунктов выпадающего меню выделим нужный пункт горизонтального и повторим описанное выше и для него. Приведем описание меню на *XAML*:

```

<Menu Grid.ColumnSpan="3" HorizontalAlignment="Right"
Height="39" Margin="0" Grid.Row="3" VerticalAlignment="Bottom"
Width="497" FontSize="20"> //начинается описание меню
<MenuItem x:Name="File1" Header="Файл">
<MenuItem x:Name="Exit" Header="Выход" Click="Exit_Click"/>
//имя, текст на окне, функция реализации
</MenuItem>
<MenuItem x:Name="ListOper" Header="Операции со списком">
<MenuItem x:Name="Insert1" Header="Добавление"
Click="Insert1_Click" />
<MenuItem x:Name="Change1" Header="Изменение"
Click="Change1_Click" />
<MenuItem x:Name="Computing1" Header="Вычисление"
Click="Computing1_Click" />
</MenuItem>
</Menu>

```



По мнению автора, после приобретения первых навыков работы с меню проще его создать именно здесь. Правда, имеются широкие возможности выбора его внешнего вида, желаем читателю самому поэкспериментировать. Реализации пунктов меню:

Программное добавление новых элементов (в нашем случае их берут с экрана, но они могут быть получены и в программе):

```
private void Insert1_Click(object sender, RoutedEventArgs e)
```

```

{
    temp = new Person();
    dg1.ItemsSource = null;
    temp.name = name1.Text;
    temp.age = Convert.ToInt32(age1.Text);
    temp.member =(bool) member1.IsChecked;
    lst1.Add(temp);
    dg1.ItemsSource = lst1;
    name1.Clear();
    age1.Clear();
}

```



Обратите внимание, что при изменении списка программой надо отключить представление на окне, а после внесения изменений восстановить. Иначе изменения не будут показаны на экране.

Изменение списка программно, изменен будет выделенный на экране элемент:

```
private void Change1_Click(object sender, RoutedEventArgs e)
```

```

{
    temp = (Person)dg1.SelectedItem;

```

```

    dg1.ItemsSource = null;
    temp.age += 3;
    dg1.ItemsSource = lst1;
}

```

Выполнение вычислений; можно использовать внутреннее или внешнее представление списка:

```

private void Computing1_Click(object sender, RoutedEventArgs e)
{
    // цикл по dataGrid
    double sr = 0;
    for (int i = 0; i < dg1.Items.Count-1; i++)
        sr += ((Person)dg1.Items[i]).age;
    answer.Content =
        (sr/(dg1.Items.Count-1)).ToString("F2");
    // вычисление по списку
    sr = lst1.Select(p => p.age).Average();

}

```

Оставим на конец самое простое средство работы со списком: можем ввести новый элемент прямо в *DataGrid* и фиксировать его нажатием *Enter*. Таким же образом можно удалить элементы (*Delete*) и изменять их значения.



6. Тестирование программного обеспечения

6.1. Методы проверки программного обеспечения

Любая составленная программа и программный комплекс должны быть всесторонне проверены, чтобы убедиться в их работоспособности. К сожалению, никому еще не удавалась разработка программ без ошибок! Методы проверки программного обеспечения делятся на **статические** и **динамические**. Суть статических методов заключается в анализе текста программы с целью выявления ошибок. Наиболее известным методом статического контроля является **синтаксический анализ**, выполняемый любым транслятором. Цель синтаксического анализа: определить, соблюдены ли формальные критерии написания текста программы, не нарушен ли синтаксис языка программирования. Имеются и другие методы статического контроля: от систем для выявления частей текста программы, которые заведомо ошибочны, до доказательства корректности программ. Статические методы могут выполняться и на компьютере, но их суть именно в анализе текста, без запуска программы на выполнение.

Среди статических методов особое место занимают методы просмотра и анализа текста программ человеком, без применения компьютера. В свое время, когда программы сначала писались на бумаге, затем переносились на машинные носители (например, перфокарты) и передавались на выполнение, значимость предварительного анализа текста программы была огромной, чтобы экономить время на тестирование и отладку. При возможности постоянного запуска программ на выполнение в ходе их разработки значимость такого анализа снизилась, но не исчезла. Рассмотрим коротко основные методы такого анализа, подробно они изложены в [5].

Инспекция кода. Написанный код просматривается маленькой группой специалистов при участии автора. Члены группы имеют возможность предварительного ознакомления с текстом программы. На заседании группы автор расскажет идеи своего алгоритма, члены группы постараются определить, не забыл ли автор учесть некоторые «тонкие моменты», редко встречающиеся случаи в решаемой задаче, а также выявить ошибки в программе на основе контрольных списков возможных ошибок. Длинный перечень таких вопросов можно найти в упомянутой монографии.

Сквозной просмотр кода. Члены группы перед заседанием готовят тесты для проверки программы и попытаются их выполнять вручную, чтобы установить ее работоспособность.

Рецензирование кода. Просмотр текста программы опытным программистом для оценки стиля программирования и подготовки рекомендации начинающему.

При использовании динамических методов программу запускают в разных условиях и анализируют результаты ее работы. Наиболее распространенным динамическим методом контроля является **тестирование**. Тестирование —

это запуск программы на разных наборах исходных данных — тестах и анализ полученных результатов с целью обнаружения ошибок. С тестированием тесно связано понятие **отладка**. Отладка — это выявление и устранение причин ошибок в работе программ. Необходимость в отладке, очевидно, возникает при обнаружении ошибок при тестировании.

В классическом жизненном цикле тестирование проводится после завершения разработки. При *XP*-программировании рекомендуется тесты разработать до написания кода. Очень хорошей можно считать практику, при которой на этапах анализа думают и о тестах. Помните, мы при рассмотрении анализа говорили о выявлении альтернативных путей выполнения вариантов использования, более подробно об этом говорилось при разработке диаграмм деятельности и последовательностей. Это все является основой для будущих тестов, и совсем неплохо думать о тестах уже на этих этапах жизненного цикла.

Искусство тестирования заключается в составлении набора тестов, **минимальных по количеству**, но обеспечивающих **максимально возможную вероятность обнаружения ошибок**. Теоретически можно с помощью тестирования доказать правильность программы: для этого ее работу необходимо проверить на ВСЕХ возможных значениях исходных данных. Это называется **полное тестирование**. К сожалению, такое тестирование на практике невозможно из-за колоссальной трудоемкости. Можно говорить и так: тестирование проверяет, выполнены ли все функциональные требования к программе. Существуют разновидности тестирования, называемые иногда испытаниями, целью которых является проверка соблюдения нефункциональных требований. Такие испытания можно проводить лишь после достижения программой работоспособности.

Тестирование, кроме проверки того, что программа делает, должно проверить и то, чтобы программа НЕ делала ничего недопустимого. Например, функция может испортить передаваемые по ссылке данные, которые она должна была использовать лишь как исходные.

Начальное тестирование проводит автор программы, но для обеспечения должного уровня проверки заключительное тестирование должен проводить НЕ автор. Это имеет как объективные, так и субъективные причины. Объективная причина заключается в том, что если программист в свое творение «вложил душу», то он часто действительно не может видеть в нем недостатки. Субъективный фактор заключается в том, что обнаружить ошибку в программе — значит признать в своей ошибке (я что-то не учел, не увидел...). Кроме того, может случиться так, что сроки поджимают, работу надо сдать, но еще не все в порядке. Отладку в любом случае должен выполнять автор — кто же, кроме него, знает, где искать ошибки...

Говорят об **альфа-** и **бета-тестировании**. Альфа-тестирование — это проверка программного обеспечения в стенах фирмы-разработчика, но специалистами, не принявшими участия в его разработке (группа тестировщиков, группа качества). После завершения альфа-тестирования проводится бета-тестирование. О бета-тестировании говорят обычно при разработке программного обеспечения для выхода на рынок. Разработчик передает бесплатно бета-

версию программы потенциальным пользователям, с которыми у него сложились хорошие партнерские отношения. Те используют новое программное обеспечение в своей работе и сообщают авторам о выявленных ошибках и вносят предложения по усовершенствованию. Ошибки придется исправить, предложения обсудить. При большом количестве пользователей могут быть и противоречивые предложения! Если разрабатывается что-то совсем новое, то проводят несколько итераций бета-тестирования. Целью первых таких итераций является выявление необходимости в таком продукте и уточнение принципов его построения. При разработке программного обеспечения конкретному заказчику вместо бета-тестирования обычно говорят об опытной эксплуатации. Но цели одинаковые: ошибки должны быть устранены, предложения приняты во внимание!

Рассмотрим процесс тестирования в этой главе в следующей последовательности:

1. Тестирование примитивных программ. Программу называем примитивной, если она не содержит вызовов функций (независимо от сложности ее логики!).
2. Тестирование программных комплексов, построенных методом функциональной декомпозиции.
3. Тестирование программных комплексов, построенных по объектно-ориентированной методике.
4. Средства тестирования *Microsoft Visual Studio*.

6.2. Тестирование примитивных программ

Примитивными считаем программы, не содержащие вызовов функций. Рекомендуется, чтобы объем такой программы не превышал 1–2 экрана. Существуют два подхода их тестирования:

- Тестирование «черного ящика», равнозначные термины: функциональное тестирование, тестирование по данным.
- Тестирование «белого ящика», равнозначные термины: тестирование по управлению, структурное тестирование.

Некоторые авторы говорят еще о тестировании «серого ящика», подразумевая под этим совместное применение обоих подходов.

- Метод функциональных диаграмм.
- Предположение об ошибке.

6.2.1. Тестирование «черного ящика»

При таком тестировании структура программы неизвестна или не используется при составлении тестов (рис. 6.1).

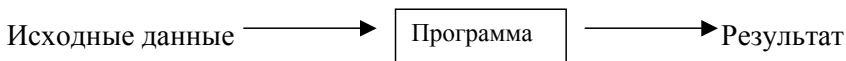


Рис. 6.1

Методы тестирования «черного ящика»:

- Метод эквивалентного разбиения.
- Метод граничных значений.
- Метод функциональных диаграмм.

Рассмотрим их по очереди.

Метод эквивалентного разбиения. Множество значений исходных данных разделяют на непересекающиеся подмножества таким образом, чтобы внутри каждого подмножества все значения были равносильны в качестве теста, но значения из разных подмножеств — нет. Простой пример: в математике модуль $|x|$ числа определяется по правилу $|x|=x$ при $x \geq 0$ и $|x|=-x$ при $x < 0$. Получили два класса эквивалентности. Разделяют **правильные** и **неправильные** классы эквивалентности. Правильный класс эквивалентности — это допустимые значения исходных данных, неправильный класс эквивалентности — недопустимые значения. Например, для функции $y = \ln(x)$ допустимым классом эквивалентности является $x > 0$, недопустимым $x \leq 0$. Программа должна быть протестирована как на правильных, так и на неправильных классах эквивалентности. В первом случае она должна дать правильный ответ, во втором случае сообщить о невозможности получения ответа с указанием причины. Конечно, множество недопустимых значений эквивалентности может быть очень большим, поэтому на практике ограничиваются наиболее вероятными случаями. Не проверять же случай, когда аргументом математической функции является символьная строка! При решении реальных задач надо учитывать и то, что любое значение всегда находится в «разумных пределах» и значения за этими пределами следует признать неправильными классами эквивалентности даже тогда, когда выполнение вычисления над ними вполне возможно. Не может же рост человека быть 5 см или 5 м; в системе продажи билетов в театр правильными можно считать лишь даты, когда состоится спектакль и нельзя продавать билеты на вчера.

Рекомендуют определить классы эквивалентности и для выходных данных и убедиться в том, что если все входные данные находятся в допустимых классах эквивалентности, то ни один результат не выходит за границы допустимых классов для выхода. Если это условие не выполняется, то в программе, скорее всего, серьезная ошибка.

Тестирование по этому методу состоит из двух этапов:

- Определение классов эквивалентности, как правильных, так и наиболее вероятных неправильных. При этом надо предусмотреть и случаи, когда исходные данные отсутствуют или их количество превышает максимально допустимое.
- Составление тестов таким образом, чтобы все классы были покрыты тестами, но их общее количество было минимальным.

Проиллюстрируем сказанное на простом примере. Дан массив. Найти сумму его элементов, удовлетворяющих условию $c1 \leq x \leq c2$. Исходные данные: $c1$, $c2$ и массив. Могут быть заданы пределы, в которых элементы массива должны находиться, допустим, от x_{min} до x_{max} .

Классы эквивалентности для $xmin$ до $xmax$, $xmin \leq xmax$. При $xmin=xmax$ задача бессмысленна. В принципе можно в программе поменять их местами. Классы эквивалентности для $c1$, $c2$:

Правильный класс эквивалентности: $xmin \leq c1$; $c1 \leq c2$; $c2 \leq xmax$. Задача может быть решена без проблем.

Неправильные классы эквивалентности: $c2 < c1$; $c1$ или $c2$ меньше $xmin$; $c1$ или $c2$ больше $xmax$. В принципе все названные случаи можно решить и алгоритмически в тестируемой программе: значения границ меньше или больше допустимых заменить на соответствующие крайние значения. Если $c2 < c1$, то поменять их местами. Но реакции программы на все эти случаи должны быть проверены.

Классы эквивалентности массива.

При правильных значениях $c1$ и $c2$ проверить принадлежность элементов заданного массива допустимому интервалу смысла не имеет: элементы, значениями за допустимыми пределами, на результат не влияют. Если массив введен не только для решения этой задачи, то такая проверка необходима. Классы эквивалентности:

1. $x_i < c1$ ответ — сумму найти невозможно.
2. $x_i > c2$ ответ — сумму найти невозможно.
3. $c1 \leq x_i \leq c2$ сумма, при вычислении учтены только удовлетворяющие условию.

В специфике программы должно быть задано, какие исходные данные требуются и каким ограничениям их значения должны удовлетворять. Допустим, в тестируемой программе $xmin$ и $xmax$ заданы константами, требуется ввести $c1$ и $c2$, количество элементов N и сами элементы. Тогда минимальное покрытие выделенных классов эквивалентности:

1. $c1 < c2$, $N=0$.
2. $c1 < c2$, $N>0$, имеются элементы $x_i < c1$ и элементы $x_i > c2$, но нет элементов $c1 \leq x_i \leq c2$.
3. $c1 < c2$, $N>0$, имеются элементы, как принадлежащие интервалу, так и не принадлежащие.
4. $c1 > c2$, $N=0$.
5. $c1 > c2$, $N>0$, имеются элементы $x_i < c1$ и элементы $x_i > c2$, но нет элементов $c1 \leq x_i \leq c2$.
6. $c1 > c2$, $N>0$, имеются элементы, как принадлежащие интервалу, так и не принадлежащие.

Крайне маловероятно на практике, что все решение содержится в одной главной функции (столь простых задач не бывает!), поэтому как продолжение начатого примера рассмотрим для случая, когда задача решена в виде функции для дальнейшего использования в составе более крупной программы. Пусть формальными параметрами функции являются исходный массив, границы диапазона $c1$ и $c2$ и сигнальная переменная. Функция не выполняет проверку принадлежности элементов массива допустимому интервалу, считаем, что эта задача, в случае необходимости, решена в вызывающей программе. Сигнальная

переменная свидетельствует о случаях, когда получение ответа невозможно. Реализация функции:

```
static double Sum1(double[] mas, double c1, double c2, out int sig)
{ // sig - сигнальная переменная
  double temp;
  if (mas == null)
  { // Проверка случая, когда массиву
    // не соответствует указатель
    sig = 1;
    return 0;
  }
  if (mas.Length==0 )
  { // Массив инициирован, но в нем нет элементов
    sig = 1;
    return 0;
  }

  sig = 2;
  // Массив задан, но в нем нет искомых элементов
  if(c1>c2)
  {
    temp=c2; c2=c1;c1=temp;
  }
  temp = 0;
  for (int i = 0; i < mas.Length; i++)
  {
    if (c1 <= mas[i] && mas[i] <= c2)
    {
      temp += mas[i];
      sig = 0; //Найден хотя бы один элемент
    }
  }
  return temp;
}
```

Ниже приведена вспомогательная главная функция для тестирования. Такая функция называется драйвером, ее задача: вызывать тестируемую функцию с разными тестами и помогать тестировщику анализировать результаты. К драйверам мы еще вернемся при обсуждении тестирования программных комплексов. Желательно исходные данные для тестов не вводить, а присваивать в самой программе. За исключением случаев, когда их количество большое, в таком случае можно для их хранения использовать файлы. Такой подход

позволяет легко выполнять одну из важнейших рекомендаций проведения тестирования: учет использованных тестов.

Пример функции-драйвера:



```
static void Main(string[] args)
```

```
{
    int kp;
    double res;
        // Test1
    /*
    double[] arr = new double[0];
    res=Sum1(arr, 10, 20, out kp);
    */
        // Test 2
    /*
    double[] arr = {-1.2,3.5,9.9,20.1,43 };
    res = Sum1(arr, 10, 20, out kp);
    */
        // Test 3
    /*
    double []arr={1.2,5.6,9.9,10.0,
10.1,13.4,19.9,20,20.1,22.5};
    res = Sum1(arr, 10, 20, out kp);
    */
        // Test4
    /*
    double[] arr = new double[0];
    res=Sum1(arr, 20, 10, out kp);
    */
        // Test 5
    /*
    double[] arr = {-1.2,3.5,9.9,20.1,43 };
    res = Sum1(arr, 20, 10, out kp);
    */
        // Test 6
    /*
    double[] arr = { 1.2, 5.6, 9.9,
10.0, 10.1, 13.4, 19.9, 20, 20.1, 22.5 };
    res = Sum1(arr, 20, 10, out kp);
    */
        // Test7
```

```
double[] arr = null;
```

```
/* Случай скорее всего теоретический, использование
фактического параметра без начального значения
```

```

    синтаксическая ошибка                                */
    res = Sum1(arr, 10, 20, out kp);
    //----- Вывод результата
    if(kp==0)
        Console.WriteLine("Sum=" + res.ToString("F1"));
    if(kp==2)
        Console.WriteLine("Нет искомых элементов");
    if(kp==1)
        Console.WriteLine("Массив пуст");
Console.ReadLine(); } }

```

В случаях 1 и 4 должно выдаваться сообщение об отсутствии элементов, эти случаи можно не разделять. Случаи, когда $c1 > c2$, программа могла бы правильно отработать без выдачи сообщения. В случаях 2 и 5 должен появиться ответ о невозможности вычисления суммы. В случаях 3 и 6 должна выдаваться сумма, в которой учтены нужные элементы и не учтены ненужные. Хороший тон программирования требует, чтобы ответ был понятен пользователю без дополнительных умозаключений. Поэтому ответ нуль для 3 и 6 следует считать неудачным. Действительно, если $c1 > 0$ и $c2 > c1$ ответ нуль означает отсутствие нужных элементов, но от пользователя требуются дополнительные рассуждения. При $c1 < 0$ и $c2 > 0$ ответ действительно может быть равен нулю.

В нашем случае исходные данные ($c1$, $c2$ и массив) независимы, поэтому мы могли проанализировать их в отдельности. В общем случае допустимые и недопустимые классы эквивалентности могут возникнуть и на их сочетаниях (20-летний человек никак не может иметь трудовой стаж 25 лет). Пусть среди исходных данных имеются переменные x и y . Пусть $0 \leq x \leq 10$ и $25 \leq y \leq 50$, но должны соблюдаться ограничения $x * y \leq 450$ и $x + y > 30$.

Рекомендации:

1. Проанализировать вопрос: какие исходные данные независимы друг от друг и между которыми имеется зависимость?
2. Какие классы эквивалентности (как правильные, так и неправильные) возникают на сочетаниях классов эквивалентности разных данных?

Метод граничных значений. Этот метод требует использовать в качестве тестов значения на границах и около границ классов эквивалентности. Если имеем дело с целыми числами, то понятно, что понимать под «около границы». При вещественных данных можно рассуждать так. Любая реально существующая величина измеряется и поступает на обработку в программу с какой-то точностью, например 2 знака после запятой. Тогда под «около» можно понимать $\pm 0,001$. Для рассмотренной выше задачи о сумме элементов в заданном интервале среди них в тесте можно рекомендовать $c1 - \Delta$, $c1$, $c1 + \Delta$, \dots $c2 - \Delta$, $c2$, $c2 + \Delta$. О том, как выбирать Δ , мы только что поговорили.

Для уменьшения количества тестов рекомендуют использовать эти два метода совместно: сначала составить тесты по эквивалентному разбиению, а затем подкорректировать их таким образом, чтобы соблюдались рекомендации второго метода. В нашем примере можно рекомендовать следующие:

1. $c1=10, c2=20$; $x: 1, 3.4, 9.99, 10, 10.1, \dots, 19.9, 20, 20.1, 22, \dots$
2. $c1=10, c2=20$; $x: 1, 3.4, 9.99, 20.1, 22, \dots$

В качестве упражнения рекомендуем читателю разработать тесты для следующих задач.

1. Решение квадратного уравнения $ax^2 + bx + c = 0$. Какие тесты необходимы, чтобы программа обрабатывала все (!) случаи и выдала либо ответ, либо сообщение о невозможности решения?
2. Дано 3 числа. Могут ли они быть сторонами треугольника, а если да — то какого: тупоугольного, прямоугольного или остроугольного?
3. И очень сложная задача. Дана дата: день — месяц — год. Допустим, что она корректная, т. е. такая дата действительно существует. Задано целое число M (возможно отрицательное). Какая дата будет через M дней (при $M > 0$) или была M дней назад (при $M < 0$)? Для начала возьмите ограничение $|M| \leq 365$.

6.2.2. Тестирование «белого ящика»

Введем понятие «граф управления программой». В любой программе присутствуют три базовых конструкции: последовательность, выбор и повторение (цикл). Последовательности в графе управления можно ставить в соответствие одну вершину или столько вершин, сколько действий имеется в последовательности. Как мы скоро увидим, это ни на что не влияет. Разветвлению ставим в соответствие столько вершин, сколько элементарных условий в нем имеется. Элементарным называем условие вида переменная — знак отношения — переменная, перед отношением может быть знак отрицания (вместо одной переменной может быть константа). Например, $x >= 5.4$. Сложное условие получается путем соединения элементарных условий знаками операции И, ИЛИ. Например, $(5.4 <= x) \&\& (x <= 9.3)$.

Пример 1. Последовательность.

Пусть имеем отрывок программы:

$K=3$;

$X=2.5$;

$Y=K*X$.

Его граф управления (рис. 6.2):

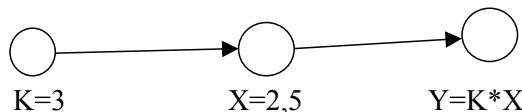


Рис. 6.2

Цикломатическая сложность программы определяется по формуле $\text{Количество_ребер} = \text{количество_вершин} + 2$, в нашем случае $2 - 3 + 2 = 1$. Нетрудно заметить, что добавление новых операторов в последовательность не меняет цикломатическую сложность. Другими словами, цикломатическая сложность последовательности всегда равна единице. Поэтому можно при со-

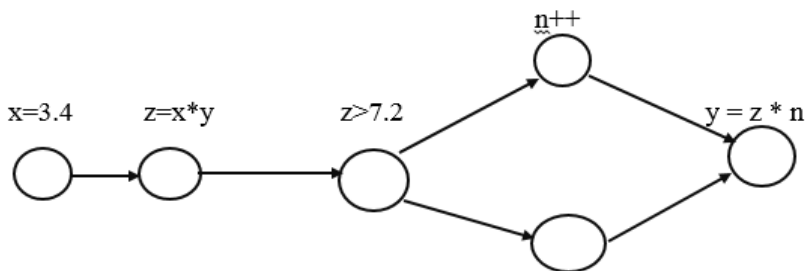
ставлении графа управления программой последовательность представить одной вершиной. Интерпретация: цикломатическая сложность — это количество путей от первой вершины до последней.

Пример 2. Выбор. Простое условие (рис. 6.3).

```
x=3.4;
z=x*y;
if ( z>7.2) n++;
else n++;
y=z*n.
```



Цикломатическая сложность равна



2.

Рис. 6.3

Пример 3. Выбор. Сложное условие.

Сложному условию на графе управления соответствует столько вершин, сколько в нем элементарных условий (рис. 6.4).

```
k=4;
if(2.3<=x && x<=6.4) z=5.2;
else z=-5.2;
y=z+k.
```

Цикломатическая сложность $7 - 6 + 2 = 3$.

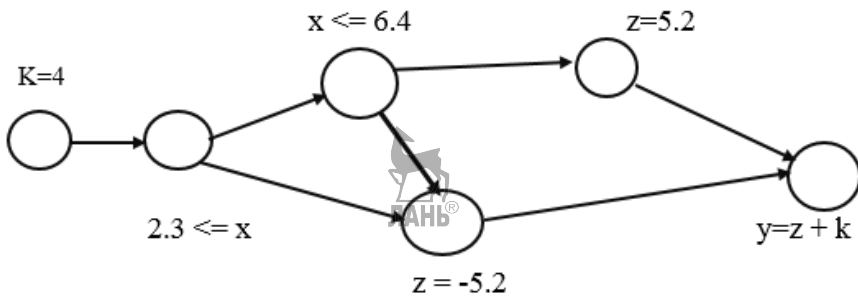


Рис. 6.4

Любой повтор (цикл) может быть реализован через выбор. Поэтому и составление графа управления аналогично случаю с выбором. При цикле с предусловием вершина выбора предшествует телу цикла, при цикле с постусловием — тело цикла предшествует выбору. Вершина(ы) выбора в обоих случаях одинаковая: оператору цикла поставим в соответствие столько вершин графа, сколько условий продолжения (прерывания) цикла. *for (inti=0; i<25; i++) {...}* одна вершина. *while (i<=25 && n!=0)* две вершины.

Вернемся к тестированию. Для тестирования «белого ящика» необходимо пропускать столько тестов, сколько путей имеется на графе управления. Для циклов с предусловием это означает, что должен быть проверен случай, когда цикл вообще не запускается. Для цикла с постусловием такой случай исключен.

При этом возникают следующие сложности.

1. При высокой сложности логики программы (большое количество условных операторов) количество путей может достигнуть астрономических величин. В таком случае рекомендуется разделить программу на части, каждая часть с одним входом и одним выходом, и протестировать эти части в отдельности.
2. Возможно наличие противоречивых путей: невозможно подбирать данные для их прохода. Элементарный пример:

```
if ( x>5.2)n++;  
if (x< -1.5) m--;
```

Очевидно, что если $x > 5.2$, то он никак не может удовлетворять условию $x < -1.5$. Поэтому не существует значения x , при котором будут выполнены как $n++$, так и $m--$. Это обстоятельство уменьшает объем тестирования.

Рекомендуем читателю составить граф управления для программы решения рассмотренной при обсуждении метода «черного ящика» задачи поиска суммы. Имеются ли там нереализуемые пути?

Возникает важный вопрос: как на практике использовать предложенные два подхода к тестированию? Можно рекомендовать следующую последовательность действий, получится подход, называемый иногда тестированием «серого ящика»:

1. На основе спецификации программы составим тесты методами «черного ящика». Допустим, получили N тестов.
2. Пусть на графе управления существует M путей.

Возможны следующие случаи:

1. $N=M$ и составленные тесты обеспечивают прохождение всех путей по меньшей мере по одному разу. Все хорошо!
2. $N=M$, но обеспечивается прохождение лишь K путей ($K < M$). Если оставшиеся $M-K$ путей противоречивые, то два или более теста будут пройдены по одному и тому же пути. Если это так и должно быть, то можно, в принципе, уменьшить количество тестов, но вполне возможно, что этого не стоит делать! Если так не должно быть (два теста пройдут по одному пути, но на самом деле они разные) — значит в программе ошибка!

3. $N=M$, но обеспечивается прохождение лишь K путей ($K < M$), среди оставшихся $M-K$ путей имеются реальные, то не все протестировано и количество тестов надо увеличить.
4. $N < M$. Если оставшиеся $M-N$ путей противоречивые, то все хорошо, можем приступить к тестированию, но если это не так, то придется составить и для них тесты.
5. $M < N$, значит, два или более теста пройдут по одному и тому же пути. Если так и должно быть, то некоторые случаи будут протестированы повторно. Если нет, то два разных случая пройдут по одному и тому же пути и в программе ошибка

6.2.3. Метод функциональных диаграмм

Другое название этого метода: метод причинно-следственных связей. Названный метод весьма эффективен, если результат зависит от сочетания входных условий. Кроме составления тестов применение этого метода позволяет проверить спецификацию программы и выявить случаи неопределенности и неоднозначности. Правда, составление функциональной диаграммы и ее анализ для задач реальной размерности весьма трудоемки и трудно поддаются автоматизации. В основе функциональной диаграммы лежит и/или граф. На этом графе вершинам соответствуют события, вершина принимает значение 0 или 1 в зависимости от того, событие имеет место или нет. Проиллюстрируем это на рис. 6.5.

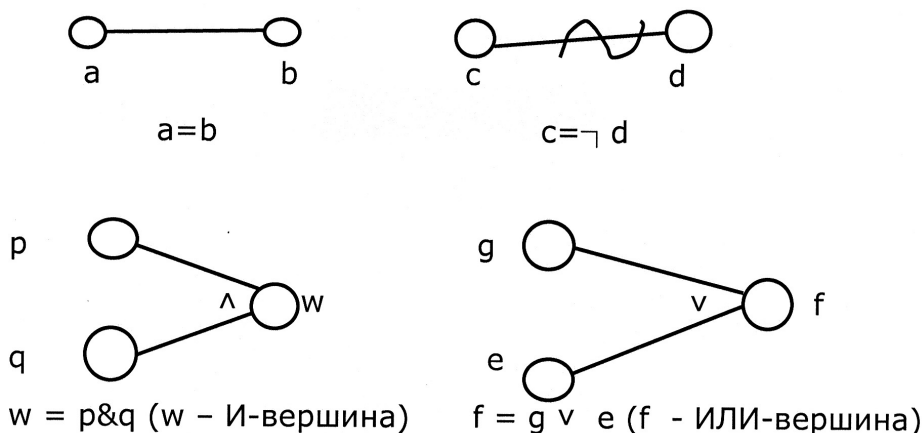


Рис. 6.5

Событие a имеет место, когда имеет место b , и наоборот. Событие c имеет место, если d не имеет место, и наоборот. Событие w имеет место, если имеют место p и q (количество аргументов можно увеличить) и т. д. На диаграмме могут быть заданы дополнительные условия (рис. 6.6). Если a , то b , обратное неверно; если вместо R писать M , то это означает: если a , то НЕ b . E означает, что c и d не могут одновременно иметь значения 1, но могут одновременно иметь значение 0. I означает, что, по меньшей мере, одно из e , f или g должно

иметь значение 1. O требует, чтобы в любой момент одно и только одно из p , q или r имело значение 1.

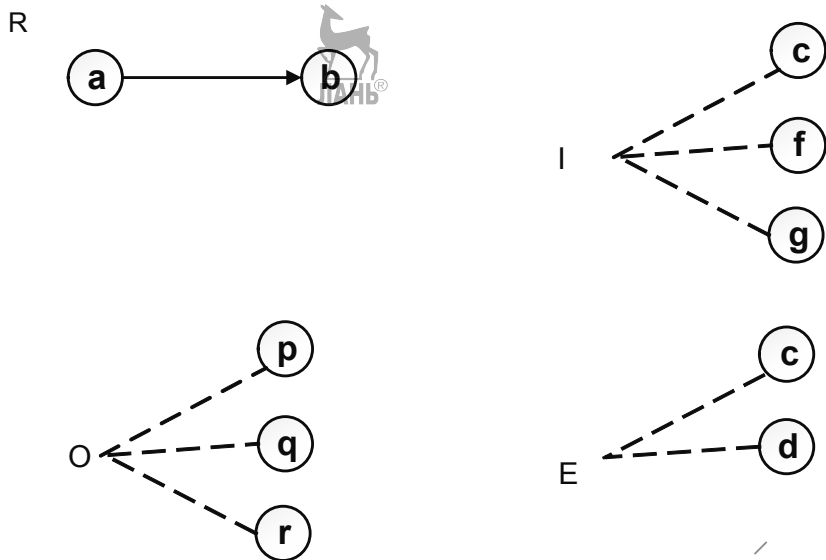


Рис. 6.6

Рассмотрим простой пример. При отправлении сотрудника в командировку начальник должен определить, ехать ему на поезде или самолете. Если расстояние не больше 400 км, то на поезде. Если расстояние от 401 до 1000 км, то поезд, если на самолет нет билетов эконом-класса, если такие билеты имеются, то самолет. Если расстояние свыше 1000 км, то самолет. Естественно, что поездка может состояться, если на самолет имеются билеты, если нет никаких билетов на самолет, то поездка откладывается. Считаем, что на поезд билеты имеются всегда. Граф представлен на рис. 6.7. Применение этого метода начинается с определения входов и выходов (относительно их обозначения никаких ограничений нет). В нашем случае:

Входы:

$R1$ — расстояние до 400 км.

$R2$ — расстояние от 401 до 1000 км.

$R3$ — расстояние свыше 1000 км.

$B1$ — имеются билеты в эконом-класс.

$B2$ — имеются билеты в бизнес-класс.

Очевидно, что из $R1$, $R2$ и $R3$ одно и только одно может иметь значение истинно. На $B1$, $B2$ ограничений нет.

Выходы:

$P1$ — поездка на поезде.

$P2$ — поездка на самолете.

$P3$ — поездка откладывается.

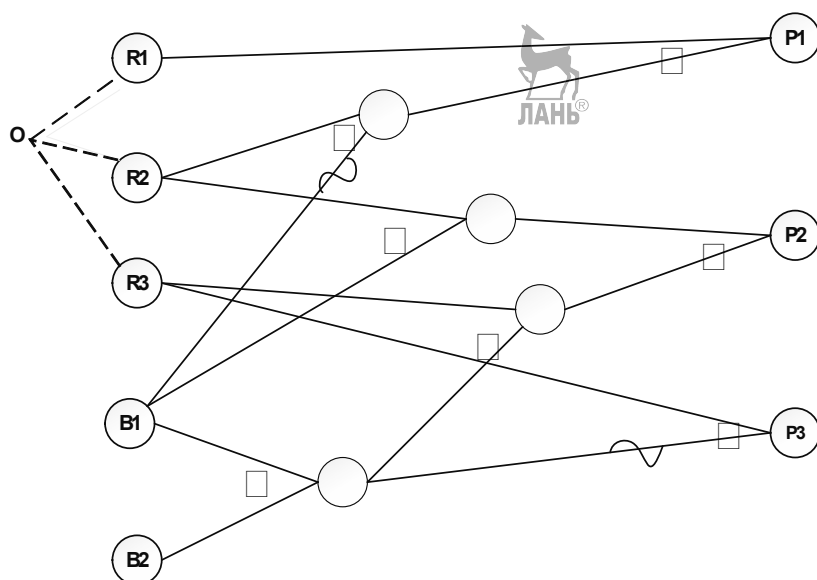


Рис. 6.7

И-ИЛИ граф нагляден, но вместо него можно использовать и функции алгебры логики, которые показывают зависимость выходов от входов. В нашем случае:

$$P1 = R1 \vee (R2 \& \neg B1);$$

$$P2 = (R2 \& B1) \vee (((B1 \vee B2) \& R3);$$

$$P3 = \neg (B1 \vee B2) \& R3 = \neg B1 \& \neg B2 \& R3.$$

Теоретически количество комбинаций входов равно 2^5 , но с учетом ограничений на входы R получим таблицу 6.1.

Таблица 6.1

№ п/п	Входы					Выходы		
	R1	R2	R3	B1	B2	P1	P2	P3
1	0	0	1	0	0			1
2	0	0	1	0	1		1	
3	0	0	1	1	0		1	
4	0	0	1	1	1		1	
5	0	1	0	0	0	1		
6	0	1	0	0	1	1		
7	0	1	0	1	0		1	
8	0	1	0	1	1		1	
9	1	0	0	0	0	1		
10	1	0	0	0	1	1		
11	1	0	0	1	0	1		
12	1	0	0	1	1	1		

С помощью этой таблицы легко выполнять проверки:

1. Каждый выход должен иметь значение 1 хотя бы один раз.
2. При любом сочетании входов только один выход должен иметь значение 1, если противоположное не предусмотрено условиями.

6.2.4. Предположение об ошибке

Это трудно назвать полноценным методом тестирования, невозможно дать конкретные рекомендации по нему. Можно лишь сказать, что на основе опыта (или интуиции) тестировщик может предполагать, где какие ошибки могут быть, и нацеливает проверки именно туда или обращает внимание на алгоритмически сложные программы, где наиболее вероятно, что такой-то случай не учтен.

6.3. Тестирование программных комплексов, построенных методом функциональной декомпозиции

Метод функциональной декомпозиции был рассмотрен в п. 2.1 и представлен на рис. 2.1. Напомним, его суть в том, что имеется главная программа, которая вызывает другие программные модули и управляет их взаимодействием в зависимости от решаемой задачи. Мы рассмотрели в предыдущем разделе тестирование примитивных программ, под которыми понимали программы, не вызывающие другие. Рассмотрим в этом разделе, как тестировать такой комплекс целиком. Существуют два подхода:

1. **Монолитное тестирование.** Все составные части тестируются независимо друг от друга, и после завершения этого выполняется сборка всего комплекса. Такую сборку иногда называют «большой скачок».
2. **Пошаговое (или инкрементное) тестирование,** где протестированные модули тут же подключают к комплексу, тестирование и сборка выполняются параллельно. Пошаговое тестирование может выполняться сверху — вниз, когда первым тестируется главный модуль, затем вызываемые им модули и так далее до нижнего уровня. Для замены пока не тестируемых модулей используют **заглушки**. Тестирование также может выполняться снизу — вверх, когда начинают с модулей самого нижнего уровня и постепенно доходят до верхнего. Для тестирования модулей (кроме главного) разрабатывают **драйверы**, о которых мы уже говорили. Возможны и другие тактики тестирования. Например, одновременно сверху — вниз и снизу — вверх, и в середине встречаются.

Тестирование каждого модуля выполняется рассмотренными нами выше методами «черного» и «белого» ящиков, но вызов функции более нижнего уровня декомпозиции рассматривается как один оператор, который при заданных исходных данных выдает требуемый результат.

Рассмотрим преимущества и недостатки этих подходов. При монолитном тестировании приходится почти для всех модулей разрабатывать как драйверы, так и заглушки, но благодаря этому можно каждый модуль всесторонне про-

тестировать и можно распараллелить работу. Принципиальный недостаток: совсем не проверяются межмодульные интерфейсы, поэтому и сборка тестируемых таким способом модулей называется «большой скачок», и найти места неправильно работающих интерфейсов часто весьма трудно.

При пошаговом тестировании при подключении нового модуля кроме его самого проверяется и его интерфейс. В ходе тестирования вновь подключенных модулей дополнительно протестируются и ранее подключенные. При этом тестирование не обязательно проводить уровень за уровнем, можно выделить часть модулей, которые обеспечивают решение значительной части, например, 75% задач, и начинать их внедрение, предупредив пользователя, что пока часть задач не решается. В ходе использования программный комплекс проходит дополнительную проверку по многим критериям.

Из сказанного ясно, что более предпочтительным является пошаговое тестирование, но полностью забыть о монолитном не следует, мы к этому еще вернемся. Вопрос о том, как лучше проводить пошаговое тестирование сверху — вниз или снизу — вверх, не имеет однозначного ответа.

Тестирование сверху — вниз.

Преимущества:

- Тестирование всего комплекса упрощается при подключении средств ввода/вывода, которые часто находятся в главном модуле и вызываются непосредственно из него.
- Рано формируется каркас всего комплекса, это позволяет проверить и правильность проектирования всего комплекса; результаты можно показать заказчику.
- Легко тестировать логику модулей верхних уровней.

Недостатки:

- Трудно обеспечить тестирование логики модулей нижних уровней. Это может создать условия для недобросовестных разработчиков и тестировщиков при приближении сроков сдачи работы (в каком-то «хитром» случае модуль нижнего уровня не работает, но он редко встречается). Правда, современные среды диалоговой разработки позволяют в какой-то степени снять эту проблему. При тестировании сверху — вниз можем поставить в модули точки прерывания и проверить, какие исходные данные он получил и какой дал ответ: правильный или нет. Трудность обеспечения прохождения всех необходимых тестов модулями нижних уровней сохраняется.
- Требуется разработать заглушки.

Тестирование снизу — вверх.

Преимущества:

- Легко протестировать логику модулей нижнего уровня, но там часто и реализованы самые сложные алгоритмы.

Недостатки:

- Долго программа не существует как единое целое, ошибки проектирования могут проявляться слишком поздно. Кроме того, это плохо психологически.
- Требуются драйверы.

Рекомендации по разработке драйверов были рассмотрены выше. Поговорим и о заглушках. Задача заглушки — имитировать работу заменяемого программного модуля, она должно выполнять такое же преобразования входных данных в выходные, как и «настоящая» программа. Согласитесь, в общем случае реализовать это совсем не просто, особенно когда значения исходных данные заменяемого ею модуля будут определены лишь в ходе работы других модулей. Кроме того, алгоритм работы самой заглушки должен быть настолько прост, чтобы не возникла в принципе задача ее тестирования. Конечно, могут быть случаи, когда полученный от заглушки ответ в ходе дальнейшего решения ни на что не влияет, и мы можем возвращать произвольное значение.

Ниже приведен вариант заглушки для программы нахождения суммы элементов массива в заданном интервале. Действительно, ввиду простоты задачи ее длина сопоставима с длиной самой функции. Принцип следующий. В заглушку ставим точку прерывания. Когда программа на ней остановится, ставим курсор мыши по очереди на исходные данные функции и на этой основе определим правильный ответ, который затем и будет введен. Некоторые среды позволяют после останова на точке прерывания менять данные в самой программе, тогда задача ввода ответа упростится, но принцип один: человек определяет правильный ответ, который затем и будет использован вызываемыми в дальнейшем программами. Повторяем, так целесообразно поступить, если проще получить требуемый ответ невозможно.

```
static double SumD(double[] mas, double c1, double c2,  
out int sig)
```

```
{  
    double temp;  
    if (mas.Length == 0)  
    {  
        temp = 0;  
        sig = 1;  
        Console.WriteLine("Массив пуст.  
                           Ответы sum=0 и sig=1");  
        return 0;  
    }
```

```
// На следующий оператор ставим точку прерывания.
```

```
    Console.WriteLine(  
        "Если массив не пуст и искомые элементы имеются -  
        вводите 0, если массив не пуст,  
        но их нет - вводите 2 "    );
```



```

sig = Int16.Parse(Console.ReadLine());
    if (sig != 0 && sig != 2)
    {
        Console.WriteLine("Ошибка. Повторите ввод!");
        Console.WriteLine("Если массив не пуст
и искомые элементы имеются - вводите 0,
если массив не пуст, но их нет —
вводите 2 " );
        sig = Int16.Parse(Console.ReadLine());
    }
    if (sig == 2) return 0;
    else
    {
        Console.WriteLine("Введите сумму ");
        temp = Double.Parse(Console.ReadLine());
    }
return temp;
}

```

Подведем итоги. В общем случае сказать, какой подход лучше: сверху — вниз или снизу — вверх, сложно. Рекомендуется тестировать сверху — вниз, но каждый модуль ДО подключения к комплексу подвергать монолитному тестированию, особенно это касается модулей со сложными алгоритмами. При постепенном подключении прошедших монолитное тестирование модулей снимается проблема «большого скачка». В средах диалоговой разработки создание драйверов и заглушек не такая уж и проблема! В первую очередь, тестировать и подключить модули, обслуживающие ввод/вывод, это упрощает тестирование остальных. Очередность подключения модулей целесообразно выбрать по функциональному признаку, чтобы комплекс в целом стал функционировать для решения хотя бы части задач. Это позволит проверить удобство использования, а также подвергать уже внедренные модули дополнительной проверке. Конечно, нельзя в таком случае забыть об ограничениях на применение, ведь не все еще закончено!

6.4. Тестирование программных комплексов, построенных по объектно-ориентированной методике

Программный комплекс, построенный по объектно-ориентированной методике, представляет иерархическую структуру классов, и его функционирование заключается в создании переменных типа класс — объектов и передаче сообщений между ними. С программистской точки зрения передача сообщения означает вызов функции или свойства своего и/или другого класса. Функции бывают синхронными — с возвращаемым значением или асинхронными, где возврат значения отсутствует. ЛАНЬ®

Как мы неоднократно повторяли выше, основной моделью жизненного цикла программного обеспечения, разрабатываемого по объектно-ориентированной методике, является спираль. Согласно этой модели, программное обеспечение разрабатывается итерационно, версия за версией, и каждая новая версия добавляет новые функциональные возможности. Поэтому важно проверить при добавлении новых возможностей, выполняются ли корректно те задачи, которые уже были реализованы в предыдущих версиях. Это называется регрессионным тестированием. Регрессионное тестирование заключается в том, что перед проверкой работы новых возможностей мы запустим тесты, на которых проверили предыдущую версию. Разумеется, не запрещено проведение регрессионного тестирования и при разработке по другим методикам.

Различают:

- Тестирование примитивных классов.
- Тестирование взаимодействия классов.
- Тестирование иерархии классов.

Тестирование примитивных классов. Класс называется примитивным, если он только принимает сообщения и не создает динамически экземпляры других классов. Класс имеет обязанности: функции, вызываемые извне. Они образуют интерфейсную часть, имеют открытый доступ. Если они сложные, то они могут быть подвергнуты функциональной декомпозиции, таким образом, возникают внутренние (закрытые) функции класса. Тестирование класса в таком случае мало чем отличается от рассмотренного выше, за исключением одного весьма существенного момента.

В построенном по методике функциональной декомпозиции программном комплексе каждая функция получает исходные данные при вызове и передает свой результат при возвращении. Правда, имеются еще глобальные данные, но их рекомендуют использовать ограниченно, лишь для данных, которые должны с одинаковыми значениями быть доступными во многих модулях. При объектно-ориентированном подходе функции — члены класса кроме значений, полученных при вызове, могут обрабатывать и данные класса, как в качестве исходных, так и результата. Так и должно быть, ведь класс — это сочетание данных и методов их обработки! В классе могут существовать и функции, которые лишь записывают данные в класс и/или изменяют их, не возвращая результата при вызове. Методика объектно-ориентированного программирования рекомендует ограничить доступ к данным извне. Выходом из ситуации является открытие доступа к данным, за значениями которых надо следить во время тестирования и отладки, а при их завершении поставить окончательные атрибуты доступа. Аналогичная ситуация возникает при тестировании функций, не входящих в интерфейс класса. Их тестирование аналогично тестированию функций нижних уровней при тестировании сверху — вниз. При необходимости более подробного тестирования можно их для этого временно открыть.

Функции класса, как правило, получают исходные данные, хотя бы частично, при вызове от класса-интерфейса или от другой функции или класса. Но любая функция может быть выполнена при определенных предусловиях, и по-

сле ее завершения должны выполняться постусловия. Мы об этом выше поговорили!

Существуют два подхода к построению объектно-ориентированных программ: **контрактное** и **защитное** программирование.

При контрактном подходе выполнение предусловий обеспечивают вместе отправитель и получатель. Отправитель должен обеспечить принадлежность исходных данных допустимым классам эквивалентности. Но в общем случае это является достаточным, но не необходимым условием успешного выполнения функции. Например, требуется 2 билета на определенный рейс. Все данные корректные, но билетов может не быть. Если корректность исходных данных достаточное условие для выполнения функции — то этот вопрос не стоит. Вывод для тестирования: при контрактном подходе тестирование на неправильных классах эквивалентности не требуется.

При защитном программировании гарантии корректности исходных данных нет, и функция сама должна обеспечить, чтобы «мусор на входе не превратили мусором на выходе». Допустимы оба подхода, но тесты должны разрабатываться с учетом применимого подхода.

Тестирование взаимодействия классов. Речь идет о тестировании класса, получающего и отправляющего сообщения. Мы рассматриваем только пассивные классы, поэтому любой класс может отправить сообщение после получения другого сообщения извне. Сообщения, получаемые таким классом, можно разделить на две категории: сообщения, полностью обрабатываемые этим классом, и сообщения, которые вызывают отправку сообщения другому классу. Тестирование первых ничем не отличается от рассмотренного выше. Тестирование сообщений второй категории похоже на тестирование модулей промежуточных уровней при функциональной декомпозиции: функция класса вызывается, и она, в свою очередь, вызывает другую функцию. При применении контрактного программирования должно быть установлено, что никакая функция при допустимых классах эквивалентности на входе не порождает недопустимых классов на выходе. Хорошим помощником при составлении тестов для тестирования взаимодействия классов являются диаграммы последовательностей, которые показывают реализацию вариантов использования со всеми возможными альтернативными путями. Все возможные пути их реализации должны быть протестированы. При длинном пути реализации варианта использования (через много функций разных классов) возникают те же проблемы, как при тестировании иерархической структуры программных модулей методом сверху — вниз. И решение их аналогично: использование драйверов, заглушек, а также точек останова, с проверкой получаемых и передаваемых значений.

Тестирование иерархии классов выполняется всегда сверху — вниз, от класса-предка к наследникам. Класс-наследник содержит все данные класса-предка (кроме закрытых) и может пользоваться функциями-членами класса-предка (кроме закрытых). При отправлении сообщения классу-наследнику существуют два варианта: оно будет обработано функцией-членом класса или наследованной функцией. Если наследованная функция уже протестирована при тестировании класса-предка, то таким образом можно уменьшить объем тести-

рования. Но при этом надо обратить внимание на виртуальные функции: имеют ли они новую реализацию в классе-наследнике или нет. Принцип простой: следует тщательно следить за тем, чтобы ВСЕ реализации виртуальных функций были полностью протестированы. Трудности их тестирования в свое время были одним из аргументов противников виртуальных функций. Придется согласиться, что доля истины здесь есть, и при их использовании требуется повышенная внимательность.

Другая проблема заключается в тестировании абстрактных классов. Напомним, что класс называется абстрактным, если он содержит хотя бы одну функцию, для которой задан только интерфейс, а реализации будут в наследниках (абстрактные функции на *C#* и *Delphi*, чисто виртуальные функции на *C++*). Можно использовать два подхода:

1. Заменить на время тестирования такого класса абстрактную функцию заглушкой.
2. Закомментировать абстрактную функцию на время тестирования и проводить тестирование ее реализации только при тестировании классов-наследников.

6.5. Отладка программ

Отладкой называют выявление и устранение причин неправильной работы программы. Очевидно, что необходимость проведения отладки возникает при обнаружении ошибок при тестировании. Отладку всегда придется проводить автору программы,

В [5] рекомендуют следующие принципы отладки:

- Думайте. Без применения компьютера постарайтесь пройти в мыслях по проблемному участку своей программы и найти ответ на вопрос «Почему она работает не так, как надо?»
- Если зашли в тупик — делайте паузу!
- Если окончательно зашли в тупик, поговорите с коллегой.
- Используйте инструментальные средства отладки в последнюю очередь.

В этой же книге рекомендуют применять индуктивный и дедуктивный подходы к отладке. При индуктивном подходе собирайте всю информацию о выявленной проблеме и на основе этого находите их причину. При дедуктивном подходе исходите из общих причин возникновения подобных ошибок и постарайтесь их применить к своему частному случаю.

Но если ничего не помогает, придется перейти к применению «грубой силы»: во всех современных средах программирования имеются точки останова и средства трассировки.

Если программа завершилась аварийно, то обычно указывается место, где это произошло. Причиной чаще всего являются некорректные данные для этой операции и/или их отсутствие. Поставьте точку останова перед операцией, вызвавшей аварийный останов, запустите снова программу и при останове на этой точке просматривайте значения участвующих в ней данных, постарайтесь най-

ти, что не так. Рекомендуется предусмотреть обработку исключительной ситуации для операторов, при выполнении которых она наиболее вероятна (см. пример поиска записи в базе данных по ключу, приведенный выше).

При завершении программы с неправильным ответом частой причиной является неправильное выполнение условных операторов. Для их проверки ставьте точку прерывания перед ними и проверьте, правильно ли выполняется разветвление.

И самое главное: помните, что невнимательность — злейший враг программиста, часто ошибки просто от невнимательности очень трудно обнаружить, потому что представить себе невозможно такой ошибки!

6.6. Средства тестирования *Microsoft Visual Studio*

Рассмотрим тестирование библиотеки классов. Пусть она для начала состоит из одного класса, имя проекта — библиотеки классов *CLibrSmpl*:

```
public class Cla
{
    double[] mas1;
    double c1, c2;
    public Cla(double[] mas1)
    { // Конструктор, он и источник части исходных данных
        this.mas1 = new double[mas1.Length];
        for (int i = 0; i < mas1.Length; i++)
            this.mas1[i] = mas1[i];
    }
    public void inpt(double c1, double c2)
    { // Пример функции, которая сохраняет результат
      // в переменных класса
        this.c1 = c1; this.c2 = c2;
    }
    public int cnt1()
    { // функция, возвращающая значение
        return mas1.Where(p => p > c1).Count();
    }
    public void cnt2(out int kol)
    { // Функция, возвращающая результат через параметр
        kol = mas1.Where(p => p < c2).Count();
    }
    public double[] arr1(ref double y)
    { // Функция, возвращающая массив.
        y += mas1.Sum();
        return mas1.Where(p => p > c1).ToArray();
    }
}
```

```

public void arr2(ref double y, out double[] res)
{ // Переменная ref
  // Функция, возвращающая массив через параметр
  y += mas1.Sum();
  res = mas1.Where(p => p > c1).ToArray();
}
void chg(double p1)
{ // Функция изменения данных класса,
  // не входит в интерфейс
  mas1 = mas1.Select(p => p = p * p1).ToArray(); ;
}
public int cnt3(double p1)
{ // Вызов внутренней функции
  chg(p1);
  return mas1.Where(p => p < c2).Count();
}
}

```

Конечно, библиотеку классов можно тестировать и традиционными методами путем создания драйверов и при необходимости и заглушек. Рассмотрим здесь использование специальных средств тестирования. Для этого необходимо создать проект ТЕСТ. В *Visual Studio* решение (*Solution*) может состоять и из нескольких проектов (*Project*). Добавим в существующее решение с нашим классом проект тестирования. Для этого поставим курсор мыши на имя решения (не проекта), активизируем контекстное меню, выберем Добавить — Новый проект — Тест. Дадим проекту тестирования имя *TestCLibrl*. Добавим ссылку на тестируемую библиотеку классов. Откроется заготовка теста:

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace UnitTestProject2
{
  [TestClass]
  public class UnitTest1
  {
    [TestMethod]           // 1
    public void TestMethod1() // 2
    {                       // 3
    }
  }
}

```

Части 1, 2 и 3 повторяем для каждой тестируемой функции. Имя функции можем менять по своему усмотрению.

После составления тестов для их запуска выберем из главного меню Тест — Запуск. Тесты будут выполнены, и откроется окно с их результатами. *Passed* — тест прошел. *Failed* — тест не прошел. Ставив курсор на имя тестовой функции, можно подробнее смотреть ее результаты. При наличии в нем оператора вывода появится текст Вывод (*Output*), активизация которого позволит посмотреть выведенное. Можно изменить данные и запустить тесты

заново, разрешается иметь и несколько тестов для одной функции, так будут хорошо видны все использованные для нее тесты. Повторно можно запустить не все тесты, а только не пройденные.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;  
using System.Linq;  
using CLibrSmpl;
```

```
namespace UnitTestProject1  
{
```



```
    [TestClass]  
    public class UnitTest1  
    {  
        [TestMethod]  
        public void TestConstructor()  
        {
```

```
            Cla my;  
            double []mas={-2.5,5.4,9.1,6.2,-7.1,15.7};  
            my = new Cla(mas);  
            int k1, k2;  
            k1 = mas.Length;  
            k2 = my.mas1.Length;  
            Assert.AreEqual(k1, k2);  
        }
```

```
    }  
    [TestMethod]  
    public void TestCnt1()  
    {
```

```
        Cla my;  
        double[] mas = { -2.5, 5.4, 9.1, 6.2, -7.1, 15.7 };  
        my = new Cla(mas);  
        my.inpt(1, 8);  
        int k1, k2;  
        k1 = 4;  
        k2 = my.cnt1();  
        Assert.AreEqual(k1, k2);  
    }
```

```
    }  
    [TestMethod]  
    public void TestCnt2()  
    {
```

```
        Cla my;  
        double[] mas = { -2.5, 5.4, 9.1, 16.2, -7.1, 15.7 };  
        my = new Cla(mas);  
        my.inpt(1, 8);  
    }
```

```

    int k1, k2;
    my.cnt2(out k1);
    k2 = 3;
    Assert.AreEqual(k1, k2);
}

```

При возвращении массива ситуация с проверкой усложняется: в *C#* переменная типа массив — это указатель на массив, и два отдельно инициализированных указателя никак не могут быть равны, и тестировщику придется самому писать отрывок программы, проверяющий правильность возвращенного массива, предварительно уточнив, что понимается под правильностью (по составу элементов, по составу и очередности...). Имеется возможность вывода во время выполнения теста. Видеть это можно через контекстное меню результатов теста: поставить курсор на имя теста, и при наличии такого вывода появится запись *Output*, при нажатии на нем откроется новое окно с выведенным текстом. Дополненный тест мог бы выглядеть следующим образом:

```

[TestMethod]
Publicvoid TestArray1()
{
    Cla my;
    double[] mas1 = { 1, 3, 5, 7, 9 };
    double[] mas2;
    double[] mas3 = { 5, 9, 7 };
    my = new Cla(mas1);
    my.inpt(4, 10);
    double y = 0;
    mas2 = my.arr1(ref y);
    for (int i = 0; i < mas2.Length; i++)
        Console.WriteLine(i + " " + mas2[i].ToString("F2"));
    mas3 = mas3.OrderBy(p => p).ToArray();
    mas2=mas2.OrderBy(p=>p).ToArray();
    bool b1 = true;
    if(mas2.Length!=mas3.Length)b1=false;
    else
        for(int i=0;i<mas2.Length;i++)
            if(mas2[i]!=mas3[i])
            {
                b1=false;
                break;
            }
    double res=25;
}

```



```

    Assert.AreEqual(y,res);
    Assert.IsTrue(b1);
}

[TestMethod]
public void TestArray2()
{
    Cla my;
    double[] mas1 = { 1, 3, 5, 7, 9 };
    double[] mas2;
    double[] mas3 = { 5, 9, 7 };
    my = new Cla(mas1);
    my.inpt(4, 10);
    double y = 0;
    my.arr2(ref y, out mas2);
    mas3 = mas3.OrderBy(p => p).ToArray();
    mas2 = mas2.OrderBy(p => p).ToArray();
    bool b1 = true;
    if (mas2.Length != mas3.Length) b1 = false;
    else
        for (int i = 0; i < mas2.Length; i++)
            if (mas2[i] != mas3[i])
            {
                b1 = false;
                break;
            }
    double res = 25;
    Assert.AreEqual(y, res);
    Assert.IsTrue(b1);
}

```

Для тестирования функции *chg()* можно описанным выше способом создать тест только после изменения ее атрибута доступа. В таком случае генерируется следующая заготовка.

```

[TestMethod]
public void TestChg()
{
    Clamy;
    double[] mas1 = { -2.5, 5.4, 9.1, 6.2, -7.1, 15.7 };
    my = new Cla(mas1);
    double[] mas2 = mas1.Select(p =>
        p = p * 6.2).ToArray();
    my.chg(6.2);
}

```

```

    bool b1=true;
    for(int i=0;i<mas1.Length;i++)
        if(mas2[i]!=my.mas1[i])
        {
            b1 = false;
            break;
        }
    Assert.IsTrue(b1);
}
[TestMethod]
public void TestCnt3()
{
    Cla my;
    double[] mas = { -2.5, 5.4, 9.1, 16.2, -7.1, 15.7 };
    my = new Cla(mas);
    my.inpt(-10, 100);
    int k1, k2;
    k1 = my.cnt3(10);
    k2 = 4;
    Assert.AreEqual(k1, k2);
}
}

```



Рассмотрим тестирование двух совместно используемых классов.

Примитивный класс содержит личные данные (для обеспечения тестирования они открыты) и вычисляемые на их основе возраст (целых лет) и индекс массы тела:

```

public class Person
{
    public string Fam;
    public string Imya;
    public DateTime dr;
    public int Rost;
    public int Ves;
    public void inpt(string Fam, string Imya, DateTime dr,
        int Rost, int Ves)
    {
        this.Fam = Fam;
        this.Imya = Imya;
        this.dr = dr;
        this.Rost = Rost;
        this.Ves = Ves;
    }
    public int vozr()
    {
        DateTime t1=DateTime.Today;

```



```

        if(t1.Month>dr.Month)
            return t1.Year - dr.Year;
        else
            return t1.Year-dr.Year-1;
    }
    public double IndVes()
    {
        returnVes / Math.Pow(Rost / 100.0, 2);
    }
}

```



Придется протестировать:

- Правильно ли установлены значения данных класса. Считаем, что их корректность обеспечивает вызывающая программа.
- Правильно ли функции вычисляют требуемые характеристики. Видно, что они вычисляются исключительно на основе данных класса.

Примеры тестов. Вопрос читателю: какие тесты следовало бы добавить для всесторонней проверки функций этого класса?

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using TestInh1;
using System.Diagnostics;

namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestFamily()
        {
            Sec target = newSec();
            string Fam = "Ivanov";
            string Imya = "Vladimir";
            DateTime dr = newDateTime(1995, 06, 17);
            int Rost = 180;
            int Ves = 70;
            target.ishdan(Fam, Imya, dr, Rost, Ves);
            string expected = "Ivanov";
            string actual;
            target.obr();
            actual = target.Family();
            Assert.AreEqual(expected, actual);
        }
    }
}

```



```

[TestMethod]
public void TestInpt()
{
    Person target = new Person();
    string Fam = "Ivanov";
    string Imya = "Vladimir";
    DateTime dr = new DateTime(1995, 06, 17);
    int Rost = 180;
    int Ves = 70;
    target.inpt(Fam, Imya, dr, Rost, Ves);
    Console.WriteLine(target.Fam + " " + target.Imya + " " +
target.dr + " " + target.Rost + " " + target.Ves);
    Debug.WriteLine("Это вывод во время тестирования");
    bool actual = false;
    if (target.Fam == Fam
        && target.Imya == Imya && target.dr == dr &&
        target.Rost == Rost && target.Ves == Ves)
        actual = true;

    Assert.IsTrue(actual);
}

[TestMethod]
public void TestIndVes()
{
    Person target = new Person();
    string Fam = "Ivanov";
    string Imya = "Vladimir";
    DateTime dr = new DateTime(1995, 06, 17);
    int Rost = 180;
    int Ves = 70;
    target.inpt(Fam, Imya, dr, Rost, Ves);
    double expected = 21.6;
    double actual;
    actual = target.IndVes();
    Assert.AreEqual(expected, actual, 0.01);
    // При проверке на равенство вещественных чисел третий
    // параметр задает требуемую точность при сравнении

}

[TestMethod]
public void TestVozr()
{
    Person target = new Person();

```



```

string Fam = "Ivanov";
string Imya = "Vladimir";
DateTime dr = new DateTime(1995, 06, 17);
int Rost = 180;
int Ves = 70;
target.inpt(Fam, Imya, dr, Rost, Ves);
int expected = 22;
int actual;
actual = target.vozr();
Assert.AreEqual(expected, actual);
    } }

```

Класс, использующий *Person*. Его тестирование проведем при условии, что *Person* протестирован.

```

public class Sec
{
    Person temp=new Person();
    int age;
    double ind;
    public void ishdan(string Fam, string Imya, DateTime dr,
                      int Rost, int ves)
    {
        temp.inpt(Fam,Imya, dr, Rost,ves);
    }
    public void obr()
{
    // Функция, изменяющая данные класса
    age=temp.vozr();
    ind=temp.IndVes();
}
    public string Family()
    {
        if(age>20&&age<40)
            return temp.Fam;
        else
            return"No Family";
    }
    public string FamInd()
    {
        if(ind<25)
            return temp.Fam;
        else
            return"NoFamily";
    }
}
}

```

Ограничимся одним тестом, оставим читателю вопрос: какие тесты еще необходимы? Не забудьте, что проверить надо и корректность исходных данных:

```
[TestMethod]
```

```
public void TestIndVes()
```

```
{
```

```
    Sec target = new Sec();  
    string Fam = "Ivanov";  
    string Imya = "Vladimir";  
    DateTime dr = new DateTime(1995, 06, 17);  
    int Rost = 180;  
    int Ves = 70;  
    target.ishdan(Fam, Imya, dr, Rost, Ves);  
    string expected = "Ivanov";  
    string actual;  
    target.obr();  
    actual = target.Family();  
    Assert.AreEqual(expected, actual);
```

```
}
```

6.7. Современный подход к проверке

Проверка объектно-ориентированных программ должна обеспечить их высокое качество: предотвратить появление дефектов и устранить дефекты, которые вкрались в программный продукт. При рассмотрении процесса создания программного продукта с позиции жизненного цикла становится очевидным, что результаты каждой его фазы нуждаются в проверке, потому что ошибки могут быть допущены везде, начиная с фазы определения требований. Поэтому фундаментальными принципами проверки объектно-ориентированных программ являются следующие:

- проверять как можно раньше;
- проверять часто;
- проверять в полном объеме.

Из сказанного вытекают следующие виды проверок:

- проверка моделей;
- тестирование классов;
- тестирование взаимодействия и иерархии классов;
- системное тестирование.

Основной недостаток стандартных проверок заключается в том, что проверяется то, что уже имеется (модели, программы), а не то, что там должно быть.



Рис. 6.7

При **целенаправленной проверке** все модели проверяют по мере их создания на предмет соответствия требованиям проекта. Модель целенаправленной проверки представлена на рис. 6.7. Целенаправленная проверка требует больших затрат, но исследования показывают, что затраты на обнаружение и устранение дефектов на ранней стадии разработки по сравнению с затратами по их обнаружению и устранению на заключительных стадиях существенно ниже.

Проверка моделей. Каждая фаза жизненного цикла программного продукта должна заканчиваться проверкой, при которой надо получить ответы на вопросы:

- корректна ли составленная модель;
- является ли она полным представлением информации, на основе которой она составлена;
- является ли она внутренне непротиворечивой и согласуется ли она с другими моделями?

Корректность — это мера точности модели. На фазе анализа — это точность описания задачи. В нашем случае — это качественное выполнение работ, описанных в разделе 4.2. На этой фазе проверка может быть осуществлена только экспертным путем с привлечением квалифицированных специалистов по предметной области. Перечень вопросов, по которым можно проверять модели, приведен в [7].

Полнота — это мера наличия в модели необходимых элементов. Проверка показывает, существуют ли тестовые случаи, которые не могут быть представлены элементами, включенными в модель.

Непротиворечивость — это мера присутствия противоречивости внутри модели или между текущей и ранее составленными моделями. Другими словами, в разных моделях не должно быть различных представлений подобных тестовых случаев. Проверка позволит определить, имеют ли место противоречия и/или конфликты между разными моделями.



В процессе проверки модели принимают участие **специалист по предметной области, тестировщик и разработчик**.

Специалист по предметной области рассматривается как источник истины, он определяет ожидаемый результат системы на конкретный вход. Квалифицированные разработчики могут быть и специалистами по предметной области, но основную роль в проверке должны играть внешние экспертные оценки.

Тестировщик проводит анализ для выбора эффективных тестовых случаев и на этой основе составляет тесты.

Разработчик представляет свои модели и предоставляет информацию, которая осталась «за моделью».

Целенаправленная проверка начинается с проверки «за столом». Каждый участник процесса проверки составляет контрольный список, характерный для типаверяемой модели.

Конечной целью создания любого продукта является получение высокого качества созданного. Показатели качества программных продуктов определены стандартом [14]. Среди показателей качества имеются легко измеряемые (временная и ресурсная эффективность), трудно измеряемые (надежность, защищенность) и чисто качественные показатели (удобство установки и использования).

Кроме того, разными авторами проведены исследования по числовым характеристикам программных продуктов с рекомендациями, в каких пределах они должны находиться и в какую сторону их желательно изменить с целью повышения качества. Подробный анализ метрик можно найти в [13]. Среды программирования позволяют вычислить значения многих числовых характеристик. При открытой библиотеке классов в *Microsoft Visual Studio* выберем из меню пункты Анализ – Вычислить метрики, будет определен набор числовых характеристик для всех классов и их функций. Правда, набор характеристик весьма скуден. Из распространенных сред богатый набор вычисляемых характеристик с разъяснением их сути имеется в среде *Embarcadero RAD Studio 10*, но только для программ на языке *Delphi*.

Заключение

В пособии были изложены базовые средства выполнения этапов анализа и проектирования, даны рекомендации по реализации в среде *Microsoft Visual Studio*. Изложены методы тестирования и рефакторинга и их инструментальная поддержка. Автор надеется, что полученные начальные сведения позволят читателю успешно работать с книгами, где эти вопросы рассмотрены более подробно. Часть из них можно найти в библиографическом списке.



Библиографический список

1. ГОСТ Р ИСО/МЭК 12207-2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств.
2. ISO/IEC 14764:2006. Разработка программного обеспечения. Процессы жизненного цикла программного обеспечения. Сопровождение.
3. Manifesto for Agile Software Development [Электронный ресурс]. — Режим доступа: www.agilemanifesto.org.
4. Назаров С. В. Архитектура и проектирование программных систем. — М. : ИНФРА-М, 2015. — 351 с.
5. Майерс Г. Искусство тестирования программ / Г. Майерс, Т. Баджетт, К. Сандлер. — М. ; СПб. : Диалектика, 2015. — 272 с.
6. The Unified Modeling Language [Электронный ресурс]. — Режим доступа: <http://www.uml-diagrams.org>. — Загл. с экрана.
7. About the Unified Modeling Language Specification Version 2.5 [Электронный ресурс]. — Режим доступа: <http://www.omg.org/spec/UML/2.5>. — Загл. с экрана.
8. Арлоу Д. UML 2 и унифицированный процесс / Д. Арлоу, А. Нейштадт. — СПб. : М. : Символ-Плюс, 2008. — 624 с.
9. Стеллман Э. Постигая Agile. Ценности, принципы, методологии / Э. Стеллман, Д. Грин. — М. : Изд-во «Манн, Иванов и Фербер», 2017. — 445 с.
10. Андерсон Д. Канбан. Альтернативный путь в Agile. — М. : Изд-во «Манн, Иванов и Фербер», 2017. — 336 с.
11. Фаулер М. Рефакторинг. Улучшение существующего кода. — СПб., 2016. — 430 с.
12. Мак-Дональд М. WPF: Windows Presentation Foundation в .NET 4.5 с примерами на C# 5.0 для профессионалов. — М. : Вильямс, 2017. — 1018 с.
13. Орлов С. А. Программная инженерия: технологии разработки программного обеспечения. — СПб. : Питер, 2017. — 640 с.
14. ГОСТ Р ИСО/МЭК 9126-93. Информационная технология. Оценка программной продукции. Характеристики качества и руководства по их применению.



Михкель Михкелевич МАРАН
ПРОГРАММНАЯ ИНЖЕНЕРИЯ
Учебное пособие
Издание второе, стереотипное

Зав. редакцией литературы
по информационным технологиям
и системам связи *О. Е. Гайнутдинова*

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 11.06.21.
Бумага офсетная. Гарнитура Школьная. Формат 70×100 ¹/₁₆.
Печать офсетная. Усл. п. л. 15,93. Тираж 30 экз.

Заказ № 738-21.

Отпечатано в полном соответствии с качеством
предоставленного оригинал-макета в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.