Е. А. Черткова

ПРОГРАММНАЯ ИНЖЕНЕРИЯ. ВИЗУАЛЬНОЕ МОДЕЛИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

УЧЕБНИК ДЛЯ СПО

2-е издание, исправленное и дополненное

Рекомендовано Учебно–методическим отделом среднего профессионального образования в качестве учебника для студентов образовательных учреждений среднего профессионального образования

Книга доступна в электронной библиотечной системе biblio-online.ru

Москва ▪ Юрайт ▪ 2019

УДК 004.4’24(075.32)
ББК 32.973.26-018.2я723
Ч-50

***Автор*:**

Черткова Елена Александровна — профессор, доктор технических наук, профессор Департамента программной инженерии факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики».

***Рецензент*:**

Ретинская И. В. — доктор технических наук, профессор кафедры прикладной математики и компьютерного моделирования Российского государственного университета нефти и газа имени И. М. Губкина.

Черткова, Е. А.

Ч-50 Программная инженерия. Визуальное моделирование программных систем : учебник для СПО / Е. А. Черткова. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2019. — 147 с. — (Серия : Профессиональное образование).

ISBN 978-5-534-09823-5

В учебнике изложены ключевые понятия программной инженерии — методы, технологии, модели процесса разработки программного обеспечения. Рассмотрены инструментальные средства программной инженерии, предназначенные для автоматизации процессов разработки программного обеспечения. Описаны методы и инструментальные средства визуального моделирования программных систем с использованием объектно-ориентированного подхода. Приведены упражнения на построение диаграмм программного обеспечения с помощью IBM Rational Rose на языке моделирования UML.

Соответствует актуальным требованиям Федерального государственного образовательного стандарта среднего профессионального образования и профессиональным требованиям.

Учебник предназначен студентам образовательных учреждений среднего профессионального образования, преподавателям и всем интересующимся.

УДК 004.4’24(075.32)
ББК 32.973.26-018.2я723

ISBN 978-5-534-09823-5

Оглавление

Часть II
ПРАКТИКУМ ПО ВИЗУАЛЬНОМУ МОДЕЛИРОВАНИЮ ПРОГРАММНЫХ СИСТЕМ

Введение

Разработка программных систем состоит из трех последовательных этапов: анализа, проектирования и реализации.

Развитие современных информационных технологий определяет постоянное повышение уровня сложности программного обеспечения. Попытка улучшения существующих систем для их адаптации к новейшим технологиям приводит к возникновению ряда технических и организационных проблем.

Современные крупные проекты в области программного обеспечения характеризуют следующие особенности:

— сложность описания (большое количество функций, элементов данных и взаимосвязи между ними), требующая тщательного моделирования и анализа процессов;

— совокупность взаимодействующих информационных и программных компонентов (модулей) системы, имеющих локальные задачи и цели функционирования (например, модуль генерации учебно-тренировочных заданий для компьютерной обучающей системы и т. п.).

Создание программных систем — логически сложная и трудоемкая работа, требующая высокой квалификации всех участников разработки. Однако до настоящего времени разработка программных систем нередко осуществляется на интуитивном уровне неформализованными методами, включающими элементы искусства, практический опыт и отсутствие экспериментальной проверки качества функционирования.

Решение задачи качественной разработки следует искать в применении современных методов программной инженерии. Качество — это цель инженерной деятельности; разработка качественных программных систем — цель программной инженерии.

Различные стили программирования требуют разных подходов к разработке программного обеспечения. При разработке традиционного программного обеспечения хорошо зарекомендовал себя структурный подход. Современные системы

на основе графического интерфейса, в том числе и информационные системы, требуют объектного программирования, и поэтому объектный подход является наилучшим способом проектирования таких систем.

Объектно-ориентированный подход подразумевает разделение системы на компоненты с разной степенью детализации. В основе этой декомпозиции лежат классы объектов. Классы связаны отношениями и обмениваются сообщениями, вызывающими операции над объектами.

Несмотря на то, что объектно-ориентированные языки программирования существовали еще в начале 1970-х гг., объектно-ориентированный подход к проектированию систем получил распространение лишь в 1990-х гг.

Исходными принципами программной инженерии, лежащими в основе современных технологий разработки, являются итеративность, модульная архитектура системы и визуальное моделирование. Реализация этих принципов должна осуществляться с использованием систем автоматизированного проектирования. Данный подход обеспечивает преимущества объектно-ориентированных методов и является платформой для создания инфраструктуры разработки информационных систем.

В соответствии с принципами программной инженерии один из основных этапов проектирования программного обеспечения — визуальное моделирование систем.

Модель программной системы — это формальное описание системы, в котором выделены основные объекты, составляющие систему, и отношения между этими объектами. В описаниях представляется замысел, или смысл системы. Модель всегда представляет собой некоторый уровень абстракции, охватывая лишь существенные черты.

Графические языки моделирования уже продолжительное время широко используются в программной индустрии. Основная причина их появления состоит в том, что языки программирования не обеспечивают нужный уровень абстракции, способный облегчить процесс проектирования систем.

Объектно-ориентированная технология визуального моделирования программных систем является преобладающей для построения систем любой степени сложности в самых различных областях.

Унификация множества объектно-ориентированных языков графического моделирования, применяемых в конце 1980-х —

начале 1990-х гг., привела к появлению унифицированного языка моделирования UML (*Unified Modeling Language*) открытого стандарта. UML — это семейство графических нотаций, в основе которых лежит единая метамодель, которая выражает семантику и структуру для моделей. Метамодель определяет характеристики этих элементов, возможные связи между ними, а также выявляет, что означают эти взаимосвязи.

Модели на языке UML содержат два основных аспекта: семантическую информацию (семантику) и визуальное представление (нотацию). С точки зрения семантики программная система — это набор взаимосвязанных логических конструкций: классов, ассоциаций, состояний, сообщений и т. п. Семантические элементы определяют содержание модели, ее смысл. Семантика используется для создания программного кода, контроля правильности модели, измерения ее сложности.

Нотация — это визуальное представление семантики модели. Визуальное представление отображает семантическую информацию, дает возможность непосредственно просматривать и редактировать модель.

Язык UML стал базовой технологией визуализации и разработки программных систем, он применяется для современного интегрированного инструментария автоматизированной разработки программного обеспечения — CASE-средств (*Computer Aided Software Engineering*). Объектно-ориентированное средство проектирования, реализующее CASE-технологии, — *IBM Rational Rose* — поддерживает визуальное моделирование на унифицированном языке UML.

UML позволяет создать своеобразный чертеж, подробно описывающий архитектуру системы, поскольку за каждым графическим символом стоит хорошо определенная семантика. С помощью такого описания (или модели) упрощается разработка и обновление программной системы. Спецификация UML не определяет конкретный процесс разработки, однако использовать этот язык моделирования удобнее всего в итеративном процессе.

В учебнике приводится описание UML-диаграмм для сквозного примера моделирования программной системы. Книга содержит упражнения на построение диаграмм программного обеспечения с помощью инструментального средства *IBM Rational Rose* на языке моделирования UML. Выполнение упражнений направлено на формирование у обучающихся навыков

самостоятельного практического применения современных методов и средств проектирования программного обеспечения информационных систем, основанных на использовании визуального моделирования.

В результате изучения материалов учебника студент должен освоить:

трудовые действия

- методологией *Rational United Process*;
- навыками добавления атрибутов, операций и связей в диаграммах класса *Add New Order*;
- навыками генерации кода C++ в *IBM Rational Rose*;

необходимые умения

- использовать модели UML для логического анализа и для проектирования, обеспечивающего реализацию системы;
- строить модели в *IBM Rational Rose*;
- создавать диаграммы вариантов использования, взаимодействия, классов, состояний, компонентов системы и размещения в *IBM Rational Rose*;

необходимые знания

- сущности и методов программной инженерии;
- проблем разработки программного обеспечения;
- моделей процесса разработки программного обеспечения;
- целей, значения и принципов виртуального моделирования систем;
- характеристики и классификации CASE-средств;
- технологии и инструментальных средств *IBM Rational*.

Часть I
ВВЕДЕНИЕ В ПРОГРАММНУЮ ИНЖЕНЕРИЮ

Глава 1
СУЩНОСТЬ И МЕТОДЫ ПРОГРАММНОЙ ИНЖЕНЕРИИ

1.1. Сущность программной инженерии

Технология разработки программных продуктов — это одна из областей инженерной науки. Для получения качественных и эффективных программных систем необходимо контролировать процесс их разработки, сроки и результаты специальными средствами, методами и технологиями инженерии программного обеспечения (программной инженерии) [10].

Целью программной инженерии является эффективное создание программных систем. Задача инженерии программного обеспечения заключается в том, чтобы организация-разработчик проводила все действия по программированию согласованно и в результате могла успешно создавать высококачественные программные продукты. Программная инженерия не рассматривает технические аспекты создания программного обеспечения (ПО) — в ее ведении такие вопросы, как управление проектом создания ПО и разработка средств, методов и теорий, необходимых для создания программных систем.

1.1.1. Развитие программной инженерии

В конце 1960-х гг. в результате объективной необходимости перехода от кустарных способов к индустриальным способам создания программного обеспечения появилась новая научная дисциплина — программная инженерия (*software engineering*).

Программная инженерия определяется как совокупность инженерных методов и средств создания программного обеспечения. В то же время — это дисциплина, изучающая при-

менение строгого систематического инженерного подхода к разработке, эксплуатации и сопровождению программного обеспечения.

Термин *software engineering* был впервые предложен в 1968 г. на конференции, проводившейся под эгидой NATO и посвященной так называемому кризису программного обеспечения (*software crisis*). Этот кризис был вызван появлением мощной вычислительной техники третьего поколения, позволявшей воплотить в жизнь более сложные, не реализуемые ранее, программные приложения.

В качестве основных инструментов создания программных продуктов начали применять алгоритмические языки высокого уровня. Они расширили возможности программистов, но, в свою очередь, привели к увеличению размеров и росту сложности программных систем, намного превысив аналогичные показатели при реализации разработок на вычислительной технике предыдущих поколений.

Оказалось, что неформальный подход недостаточен для разработки сложных систем нового поколения. На реализацию сложных программных проектов иногда уходили многие годы. Стоимость таких проектов многократно возрастала по сравнению с первоначальными расчетами, сами программные системы получались ненадежными, сложными в эксплуатации и сопровождении. Разработка программного обеспечения оказалась в кризисе. Стоимость аппаратных средств постепенно снижалась, тогда как стоимость программного обеспечения стремительно возрастала (рис. 1.1).

Возникла необходимость в новых технологиях и методах управления сложными комплексными проектами разработки программных систем.

В 1975 г. в Вашингтоне была проведена первая международная конференция, посвященная программной инженерии. Тогда же появилось первое издание, посвященное программной инженерии, — *IEEE Transactions on Software Engineering*.

Становление и развитие программной инженерии характеризуется двумя этапами: 1970-е и 1980-е гг. — систематизация и стандартизация создания программного обеспечения (на основе структурного подхода) и с начала 1990-х гг. — переход к сборочному, индустриальному способу создания программного обеспечения (на основе объектно-ориентированного подхода).

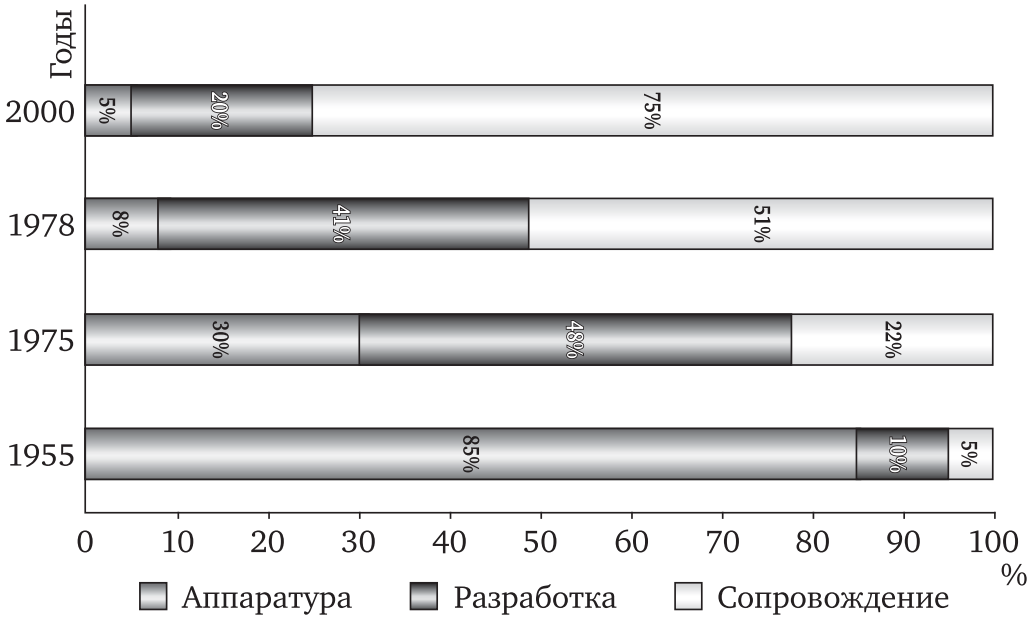

Рис. 1.1. **Тенденции изменения соотношения стоимости аппаратных средств и программного обеспечения**

1990-е гг. ознаменовали первую реальную попытку превратить разработку программного обеспечения в инженерную дисциплину с помощью концепций *CBSE* (*component-based software engineering* — компонентная разработка программного обеспечения) и *COTS* (*commercial off-the-shelf* — готовые коммерчески доступные компоненты).

Идея состоит в создании небольших, высококачественных модулей и последующего их объединения. Проблема, безусловно, заключается в том, что объединенные вместе высококачественные модули не обязательно превратятся в высококачественную систему. Комбинированная система может оказаться негодной из-за некорректного способа объединения, либо из-за ошибочных представлений о поведении компонентов или о среде, в которую они помещаются.

Специалисты отмечают, что развитие программной инженерии значительно повысило качество современного программного обеспечения. Созданы эффективные методы спецификации программного обеспечения, его разработки и внедрения. Новые средства и технологии позволяют существенно уменьшить усилия и затраты на создание программных систем.

Однако можно утверждать, что на сегодняшний день разработка программного обеспечения все еще является до известной степени ремеслом, а не инженерной дисциплиной. В инженер-

ных науках адекватность проекта его материальному воплощению проверяется на всех стадиях разработки с использованием системы основных принципов (часто прикладных законах физики), из которых можно вывести всю систему знаний.

В то же время разработка программного обеспечения базируется в основном на методе проб и ошибок. При этом механизм проверки программных систем — тестирование — выявляет дефекты проекта на поздних стадиях разработки, когда исправление ошибок становится дорогим и разрушительным для проекта.

Эксперт компании *IBM Rational* Кони Бюрер, анализируя причины укоренения идеологии проб и ошибок при разработке, приходит к следующим выводам. По своей сути разработка программного обеспечения является проектированием и не имеет признаков строительства или производства и соответственно конечный результат проектирования — код на языке высокого уровня — является чертежом программного обеспечения.

Если создание проекта (чертежа) требует значительных усилий, то дальнейшее построение на его основе программного продукта (двоичного исполняемого кода) осуществляется механически с помощью компилятора и компоновщика практически бесплатно. Поэтому идеология проб и ошибок глубоко укоренилась в процессе разработки программного обеспечения, а сообщество программистов не заинтересовано в исследованиях основных принципов разработки.

Чтобы стать наукой, разработка программного обеспечения должна подвергнуться сдвигу парадигмы от методов проб и ошибок к системе основных принципов, на основе которых можно было бы строить свои правила и методы.

Это не умаляет достоинства успешных методов и правил, часто называемых оптимальными методиками, которые создавались на протяжении последних двух десятилетий такими ведущими мастерами программной отрасли с мировыми именами, как Града Буч, Джеймс Рамбо, Айвар Якобсон и др.

Важный тезис программной инженерии: первый шаг на пути к созданию системы с заданными свойствами — это программная архитектура как структура, заключающая в себе программные элементы, их внешние связи и взаимосвязи.

Программная инженерия предполагает, что реализация проекта должна осуществляться на основе архитектуры программ-

ного обеспечения, и этот процесс предусматривает согласованность действий разработчиков со структурами и протоколами взаимодействия, заданными архитектурой [1].

Соответствие положениям архитектуры в первую очередь обеспечивается четкой документацией, а дополнительным преимуществом в контексте этой задачи будет среда или инфраструктура, активно содействующая (в отличие от простого кода) созданию и сопровождению архитектуры программного обеспечения.

Но в большинстве современных разработок хорошая программная архитектура — ключевой фактор успеха — создается путем проб и ошибок. Это объясняет, почему двумя наиболее явными проблемами неудачных программных проектов являются переделка программ и обнаружение негодности проекта на его поздних стадиях. Программист проектирует архитектуру на ранних стадиях разработки программного обеспечения, но не имеет возможности сразу же оценить ее качество, поскольку отсутствуют основные принципы для доказательства адекватности проекта.

Наряду с архитектурой программного обеспечения в соответствии с концепциями программной инженерии существенным фактором успешности является внедрение технологического процесса разработки. По данным SEI (*Software Engineering Institute* — Институт программной инженерии) в последние годы до 80 % всего эксплуатируемого программного обеспечения разрабатывалось без использования дисциплины проектирования, методом «*code and fix*» (кодирования и исправления ошибок). Одна из причин этого — стремление сэкономить время и средства на внедрение технологического процесса разработки программного обеспечения.

В рамках программной инженерии в 1990-х гг. было предложено еще одно решение проблемы качества программного обеспечения под названием «совершенствование процесса разработки программ». Был сформулирован основной принцип этой концепции: создание программного обеспечения — это задача управления, к которой можно применять соответствующие процедуры управления данными, процессами и практическими методами с целью создания оптимального решения.

Реализовать эту концепцию предлагалось в соответствии со ставшей теперь популярной и часто критикуемой моделью CMM (*Capability Maturity Model*), разработанной Институтом

программной инженерии (SEI) при Питтсбургском университете Карнеги-Меллона (*Carnegie Mellon University*) в США.

В настоящее время программная инженерия представляет собой обширную и хорошо разработанную область компьютерной науки и технологии, включающую в себя многообразные математические, инженерные, экономические и управленческие аспекты.

Согласно *SWEBOK* 2004 (*Guide to the Software Engineering Body of Knowledge* — Руководство к своду знаний по программной инженерии) программная инженерия включает в себя 10 основных и 7 дополнительных областей знаний, на которых базируются процессы разработки ПО. К основным областям знаний относятся следующие.

1. *Software requirements* — программные требования.
2. *Software design* — дизайн (архитектура).
3. *Software construction* — конструирование программного обеспечения.
4. *Software testing* — тестирование.
5. *Software maintenance* — эксплуатация (поддержка) программного обеспечения.
6. *Software configuration management* — конфигурационное управление.
7. *Software engineering management* — управление в программной инженерии.
8. *Software engineering process* — процессы программной инженерии.
9. *Software engineering tools and methods* — инструменты и методы.
10. *Software quality* — качество программного обеспечения.

Дополнительные области знаний включают в себя следующие.

1. *Computer engineering* — разработка компьютеров.
2. *Computer science* — информатика.
3. *Management* — общий менеджмент.
4. *Mathematics* — математика.
5. *Project management* — управление проектами.
6. *Quality management* — управление качеством.
7. *Systems engineering* — системное проектирование.

Известный аналитик Дэйв Парнас удачно сформулировал различие между программированием и программной инженерией. По его мнению, для продуктов, которые разрабаты-

вает один человек в единственной версии, программирования вполне достаточно. Однако если вы предполагаете, что с продуктом будут работать сторонние пользователи, или необходимо впоследствии дать ему самостоятельную оценку, без программной инженерии не обойтись.

Большинство экспертов считает, что залогом успешности будущих проектов является использование согласованных процессов программной инженерии. По мнению Иана Грэхема, всемирно известного специалиста в области объектных технологий, это будет отличительным признаком индустрии первого десятилетия XXI столетия, что позволит отметить этот период как «золотой век архитектуры и совершенствования процесса».

Необходимость перехода от традиционной техники разработки программного обеспечения к современным методам программной инженерии подтверждает базовый принцип индустриализации.

Его сущность заключается в том, что рост уровня производства предполагает повышение производительности труда членов команды разработчиков. Но эффективность традиционного кодирования, связанная с количеством строк программного кода, пропорциональна числу разработчиков в команде.

В то же время, как показал Филипп Канн — основатель корпорации *Borland International* в своем, так называемом законе Филиппа, сформулированном на конференции *COMDEX* (Лас-Вегас, 1992), производительность разработчиков обратно пропорциональна их числу в команде (рис. 1.2).

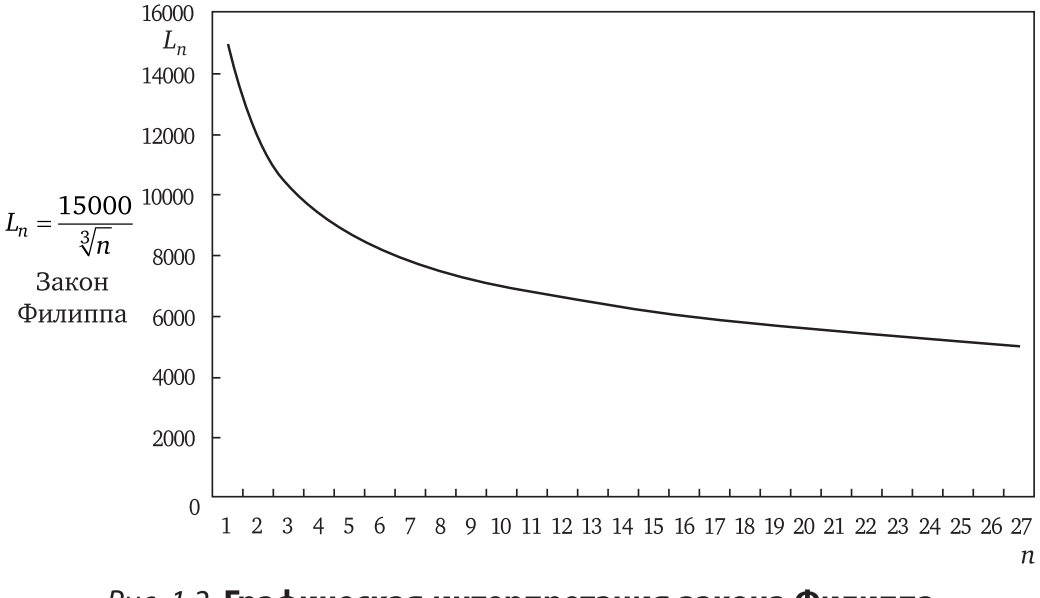

Рис. 1.2. **Графическая интерпретация закона Филиппа**

Отсюда следует, что индустриализация программной отрасли требует более эффективных методов разработки программного обеспечения, чем традиционное кодирование.

1.2. Методы программной инженерии

1.2.1. Компоненты методов

Методы программной инженерии — это структурные решения, предназначенные для разработки программного обеспечения и включающие системные модели, формализованные нотации и правила проектирования, а также способы управления процессом разработки.

В качестве основы для реализации методов программной инженерии используются средства автоматизированной разработки программного обеспечения — CASE-средства (*Computer Aided Software Engineering*).

Совокупность методов, применяемых в жизненном цикле разработки программного обеспечения и объединенных одним общим философским подходом, представляет собой методологию разработки.

Методы разработки чрезвычайно важны для преодоления сложности программирования системы, так как они упорядочивают процесс создания сложных систем, и как общие средства доступны для всей команды разработчиков. Кроме того, метод — это последовательный процесс создания моделей, который описывает определенным образом различные стороны разрабатываемой программной системы. Их применение позволяет менеджерам в процессе разработки оценить степень продвижения и риск.

Различают следующие методы программной инженерии:

- методы прототипирования, базирующиеся на различных формах прототипов;
- формальные методы, обоснованные математически;
- эвристические методы, касающиеся неформализованных подходов (структурные, ориентированные на данные, объектно-ориентированные).

Методы прототипирования используются для получения прототипов — промежуточных версий программной системы. Прототипы создаются в некоторых процессах разработки для демонстрации концепций, заложенных в системе, проверки

вариантов требований, а также выявления проблем, которые могут возникнуть как в ходе разработки, так и при эксплуатации системы.

Термин «*формальные методы*» подразумевает ряд операций, в состав которых входят создание формальной спецификации системы, анализ и доказательство спецификации, реализация системы на основе преобразования формальной спецификации в программы и верификация программ. В формальном языке системной спецификации заложены математические концепции. При этом используется область дискретной математики, основанной на алгебре, теории множеств и алгебре логики.

В 1980-х гг. многие исследователи считали, что формальные спецификации и формальные методы являются наиболее эффективным путем улучшения качества программного обеспечения. Они привели веские доводы в пользу того, что строгий и детальный анализ, который является неотъемлемой частью формальных методов, приведет к созданию программ с малым количеством ошибок. Эксперты предсказывали, что к XXI столетию большую часть программного обеспечения будут разрабатывать с использованием формальных методов.

Однако программная инженерия пошла другим путем. Прошло более 35 лет с начала исследований по использованию математических методов в процессе создания программного обеспечения. Так называемые формальные методы разработки программных систем не получили широкого признания, несмотря на то, что при их использовании может быть достигнуто повышение качества программ путем доказательства их правильности. Многие компании, разрабатывающие программное обеспечение, не считают экономически выгодным применение этих методов в процессе разработки.

Развитие структурных методов, методов управления конфигурацией и т. д. позволило повысить качество программ при более низких затратах по сравнению со стоимостью разработки формализованными методами. Кроме того, с формальной спецификацией плохо согласуются методы быстрой разработки программного обеспечения. Это вступает в противоречие с тем, что в настоящее время главным критерием программной индустрии для некоторых классов систем является не качество, а время поставки их на рынок.

Формальные математические спецификации не содержат деталей реализации системы, но должны представлять ее пол-

ную математическую модель. Существует два основных подхода к разработке формальной спецификации:

- алгебраический подход, при котором система описывается в терминах операций и их отношений;
- подход, ориентированный на моделирование, при котором модель системы строят с использованием математических конструкций, таких как множества и последовательности, а системные операции определяются тем, как они изменяют состояние системы.

Формальные методы оказались рентабельны в ограниченной области применения: это разработка критических систем, где важны такие свойства, как безопасность, безотказность и защищенность. Примерами критических систем, при разработке которых успешно применялись формальные методы, являются информационные системы управления воздушным транспортом, системы сигнализации на железной дороге, бортовые системы космических кораблей и медицинские системы управления.

В 1960—1970-е гг. было разработано много *эвристических методов*, помогающих справиться с растущей сложностью программ. Общей стратегией борьбы со сложностью программирования системы стала ее декомпозиция, т. е. разделение на мелкие составные части. Разработчики перестали ограничиваться единственным атрибутом качества программ — корректным функционированием, и перешли к структурированию.

На основе анализа потоков данных, диаграмм отношений «сущность-связь», информационной закрытости и ряда других принципов и методик сформировались новые проектные методологии. Наибольшее распространение получило структурное проектирование по методу «сверху вниз». Метод был непосредственно основан на топологии традиционных языков высокого уровня типа FORTRAN или COBOL.

В этих языках основной базовой единицей является подпрограмма, и программа в целом принимает форму дерева, в котором одни подпрограммы в процессе работы вызывают другие подпрограммы. Структурное проектирование использует именно такой подход: алгоритмическая декомпозиция применяется для разбиения большой задачи на более мелкие.

В большинстве появившихся впоследствии методов, предлагающих множество формализованных нотаций и нормативных руководств для проектирования программного обеспечения,

были устранены очевидные недостатки структурного проектирования.

Эвристические методы разработки программного обеспечения делятся на три основные группы [10]:

- метод структурного анализа и проектирования;
- метод потоков данных Джексона (JSD — *Jackson System Development*);
- объектно-ориентированные методы.

Структурный подход основан на алгоритмической декомпозиции. В методе потоков данных программная система рассматривается как преобразователь входных потоков в выходные. В основе объектно-ориентированного проектирования (ООП) лежит представление о том, что программную систему необходимо проектировать как совокупность взаимодействующих друг с другом объектов, рассматривая каждый объект как экземпляр определенного класса, причем классы образуют иерархию.

Все упомянутые методы основаны на идее создания моделей системы, которые можно представить графически, и на использовании этих моделей в качестве спецификации системы или ее структуры. Методы программной инженерии обычно включают в себя компоненты, совокупность которых обеспечивает взаимосвязанные процедуры моделирования создаваемых систем (табл. 1.1).

Таблица 1.1

Компоненты методов инженерии программного обеспечения

Компонент	Содержание	Пример
Описание модели системы	Описание моделей создаваемых систем и графическая нотация, используемая для разработки этих моделей	Модели объектов, модели статической структуры данных, модели потоков данных, модели изменения состояний системы и т. п.
Правила	Правила и ограничения, которые необходимо выполнять при разработке системы	Каждый элемент системы должен иметь уникальное имя
Рекомендации	Эвристические советы и рекомендации, отражающие практический опыт применения данного метода	Любой объект модели не должен иметь более семи подчиненных ему объектов

Окончание табл. 1.1

Компонент	Содержание	Пример
Руководство по применению метода	Описание работ, которые необходимо выполнить для построения модели системы, а также рекомендации по организации этих работ	Атрибуты любого объекта должны быть документированы, прежде чем будут определены операции, связанные этим объектом

1.2.2. Структурный подход

В *структурном подходе* (функционально-модульном) используется принцип функциональной декомпозиции, при которой структура системы описывается в терминах иерархии ее функций и передачи информации между отдельными функциональными элементами.

В период появления структурного подхода программные продукты отличались *процедурным* характером — запрограммированная процедура выполняла свою задачу последовательным и предсказуемым образом, после чего завершалась. *Структурный подход к разработке* успешно использовался для производства подобных систем.

Структурная методология предоставляла в распоряжение разработчиков строгие формализованные методы описания информационных систем и принимаемых технических решений. Она основывалась на наглядной графической технике: для описания проекта использовались различного рода диаграммы и схемы. Наглядность и строгость средств структурного анализа позволяла разработчикам и будущим пользователям системы с самого начала неформально участвовать в ее создании, обсуждать и закреплять понимание основных технических решений.

Однако широкое применение этой методологии и следование ее рекомендациям при разработке конкретных проектов встречалось достаточно редко, поскольку ее практически невозможно реализовать на должном уровне ручным неавтоматизированным способом. Также очень трудно разработать вручную и графически представить строгие формальные спецификации системы, проверить их полноту и непротиворечивость и, тем более, изменить. Если все же удается создать строгую систему проектных документов, то ее переработка при появлении серьезных изменений практически неосуществима.

Если участники проекта пытались прибегнуть к ручной разработке, то перед ними возникали следующие проблемы:

- неадекватная спецификация требований;
- неспособность обнаруживать ошибки в проектных решениях;
- низкое качество документации, снижающее эксплуатационные качества;
- затяжной цикл и неудовлетворительные результаты тестирования.

Отсутствие специализированных программно-технологических средств для разработки проектов, в частности, основанных на информатизации, затрудняло для проектировщиков информационных систем использование компьютерных технологий для повышения качества и производительности своей работы.

Структурный подход к анализу и проектированию отличается следующими особенностями, которые плохо увязываются с современными методами конструирования программного обеспечения.

- Этот подход скорее является последовательным и трансформационным, чем итеративным подходом с наращиванием возможностей, что осложняет реализацию непрерывного процесса разработки, осуществляемого посредством итеративной детализации и пошаговой поставки ПО с наращенными возможностями.
- Структурный подход предопределяет получение «негибких» решений, которые трудно масштабировать и расширять в дальнейшем.
- Подход предполагает разработку «с чистого листа» и не поддерживает повторное использование уже существующих компонент.

Трансформационный характер структурного подхода является источником повышенного риска неправильно истолковать исходные требования к системе в процессе разработки. Это связано с тем, что осуществляется постепенная замена декларативной семантики моделей анализа на процедурные решения для проектных моделей и программного кода.

Тем не менее, структурный подход по-прежнему сохраняет свою значимость и достаточно широко используется на практике, особенно в тех проектах сложной системы, в которых невозможно обойтись только одним способом декомпозиции.

1.2.3. Объектно-ориентированный подход

На основе структурного подхода сформировалась концепция объектно-ориентированного метода (программирования, анализа, проектирования, баз данных), — фактически целая философия разработки систем и представления знаний на базе мощного подхода [5]. Основное отличие объектно-ориентированного анализа и проектирования от структурного состоит в декомпозиции проблемы на понятия (объекты), а не на функции.

Исторически развитие этой области началось с объектно-ориентированного программирования. В 1980—1990-х гг. появились объектно-ориентированные методы разработки программного обеспечения, предложенные Гради Бучем (Booch), Джеймсом Рамбо (OMT — *Object Modeling Technique*) и Айваром Якобсоном (OOSE — *Object Oriented Software Engineering*).

Несмотря на явное преимущество объектно-ориентированных технологий разработки программного обеспечения перед структурными, их распространение было ограниченным, поскольку ни один из методов не давал единой и цельной объектной модели системы. Кроме того, широкому распространению объектно-ориентированных методов при разработке программного обеспечения мешало отсутствие единого стандарта языка объектно-ориентированного моделирования.

В течение 1994—1996 гг. упомянутые разработчики объединили свои усилия под эгидой *Rational Software Corporation* для создания единого языка моделирования, который воплотил бы все существенные и успешные разработки в данной области и стал бы стандартом языка объектно-ориентированного моделирования.

Грандиозный труд, в котором наряду с *Rational* участвовали представители таких крупных компаний, как *Microsoft, IBM, Hewlett-Packard, Oracle, Platinum Technology* и нескольких сотен других, завершился созданием в январе 1997 года версии 1.0 унифицированного языка моделирования UML (*Unified Modeling Language*) [9].

Объектно-ориентированный метод использует объектную декомпозицию. При этом структура системы описывается в терминах объектов и связей между ними, а поведение системы — в терминах обмена сообщениями между объектами.

Гради Буч сформулировал главное достоинство объектно-ориентированного подхода (ООП) следующим образом:

объектно-ориентированные системы более открыты и легче поддаются внесению изменений, поскольку их конструкция базируется на устойчивых формах. Это дает возможность системе развиваться постепенно и не приводит к полной ее переработке даже в случае существенных изменений исходных требований.

Буч отметил также ряд преимуществ ООП.

- Объектная декомпозиция дает возможность создавать программные системы меньшего размера путем использования общих механизмов, обеспечивающих необходимую экономию выразительных средств. Использование ООП существенно повышает уровень унификации разработки и пригодность для повторного использования не только программного обеспечения, но и проектов, что, в конце концов, ведет к сборочному созданию. Системы зачастую получаются более компактными, чем их не объектно-ориентированные эквиваленты, что означает не только уменьшение объема программного кода, но и удешевление проекта за счет использования предыдущих разработок.
- Объектная декомпозиция уменьшает риск создания сложных программных систем, так как она предполагает эволюционный путь развития системы на базе относительно небольших подсистем. Процесс интеграции системы растягивается на все время разработки, а не превращается в единовременное событие.
- Объектная модель вполне естественна, поскольку в первую очередь ориентирована на человеческое восприятие мира, а не на компьютерную реализацию.
- Объектная модель позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования.

По сути, все современные приложения являются объектно-ориентированными, а некоторые языки программирования (например, *Java*) предполагают использование объектно-ориентированных структур.

В соответствии с объектно-ориентированной парадигмой все приложения делятся на элементы кода (объекты), относительно независимые друг от друга. Готовое приложение можно затем создать, сложив эти объекты вместе. При этом программные объекты выполняются случайным, непредсказуемым образом, и программа не завершается до тех пор, пока пользователь не прекратит ее выполнение.

Одной из причин популярности объектного подхода является возможность обеспечения функциональных требований для *приложений с элементами мультимедиа*, к которым относятся и компьютерные обучающие системы, путем реализации концепции «помещения объектов в оболочку».

Объектный подход к разработке систем следует *итеративному* процессу *с наращиванием возможностей*. Единая модель (и один проектный документ) конкретизируется на этапах анализа, проектирования и реализации — в результате успешных итераций добавляются новые детали, при необходимости вводятся изменения и усовершенствования, а выпуски программных модулей с наращенными возможностями поддерживают эволюцию требований к системе и обеспечивают обратную связь, необходимую для продолжения разработки модулей.

Разработка с помощью последовательной детализации становится возможной благодаря тому, что все создаваемые в ходе разработки модели (анализа, проектирования и реализации) обладают семантическим богатством и базируются на одном и том же языке, так как базовый словарь этих моделей существенно не отличается (классы, атрибуты, методы, наследование, полиморфизм и т. д.).

Объектный подход устраняет большинство из наиболее значительных недостатков структурного подхода, однако служит источником некоторых новых нижеперечисленных проблем.

- Этап анализа проводится на еще более высоком уровне абстракции, и если серверная часть решения по реализации предполагает использование реляционной базы данных, то семантический разрыв между концепцией и ее реализацией может быть значительным.

Хотя анализ и проектирование могут проводиться итеративно с наращиванием возможностей, в итоге разработка достигает этапа реализации, которая требует трансформации решения применительно к реляционной базе данных. Если в качестве платформы реализации используется объектная или объектно-реляционная база данных, трансформация проекта проходит значительно легче.

- Управление проектом сложно осуществлять. Менеджеры измеряют степень продвижения разработки с помощью четко определенной декомпозиции работ, элементов комплекта поставки и ключевых этапов. При объектной разработке с помощью «детализации» не существует четких границ между этапами, а проектная документация непрерывно развивается.

Приемлемое решение в такой ситуации заключается в делении проекта на небольшие модули и управлении ходом разработки за счет частого выпуска выполняемых версий этих модулей (некоторые из этих выпусков могут быть для внутреннего применения, а другие — поставляться заказчику).

• Проблема использования объектного подхода связана с возрастающей сложностью решения, что, в свою очередь, сказывается на таких характеристиках программного обеспечения, как приспособленность к сопровождению и масштабируемость.

Как отмечает известный специалист в области программной инженерии А. Вендров, объектно-ориентированный подход не дает немедленной отдачи. Эффект от его применения начинает сказываться после разработки двух-трех проектов и накопления повторно используемых компонентов, отражающих типовые проектные решения в данной области. Переход организации на объектно-ориентированную технологию — это смена мировоззрения, а не просто изучение новых CASE-средств и языков программирования.

Однако, сложности, связанные с объектным подходом, преодолимы и не должны повлиять на прерогативу их применения для разработки программных систем по сравнению с процедурным стилем пакетных приложений на языках программирования типа COBOL.

На сегодняшний день объектно-ориентированный подход — единственный известный метод, позволяющий осуществить разработку подобных систем — нового управляемого событиями программного обеспечения, отличающегося высоким уровнем интерактивности.

Глава 2
ПРОЦЕСС РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Процесс разработки программного обеспечения — это совокупность взаимосвязанных процессов и результатов их выполнения, которые ведут к созданию программного продукта. Эти процессы основываются главным образом на технологиях инженерии программного обеспечения.

2.1. Проблемы разработки программного обеспечения

Американский ученый в области теории вычислительных систем Ф. Брукс в классической статье по инженерии программного обеспечения определил свойства, характерные для программной инженерии. Согласно Бруксу сущность программной инженерии определяется собственными свойствами программного обеспечения, которые вызывают трудности при его создании.

Эти трудности можно только осознать, но нельзя преодолеть за счет какого-либо технологического прорыва. Сущность программной инженерии проистекает из таких свойств ПО, *как сложность*, *податливость*, *изменчивость* и *неосязаемость*.

Эти четыре «существенные трудности» создания программного обеспечения определяют инвариант или неизменную составляющую процесса его разработки. Инвариант констатирует тот факт, что программное обеспечение является продуктом творческого акта разработки — ремесла или даже искусства.

Свойства программной инженерии, которые носят случайный характер, также порождают определенные трудности при создании программных систем. В свою очередь «случайные трудности» относятся к вариативным проблемам разработки и делятся на три категории:

— участники проекта;
— процесс;
— язык и средства моделирования.

Рассмотрим некоторые проблемы разработки программного обеспечения.

2.1.1. Инвариантные проблемы разработки ПО

По определению Ф. Брукса, сложность программного обеспечения — не случайное его свойство и вызывается четырьмя основными причинами:

- сложностью реальной предметной области, из которой исходит заказ на разработку;
- трудностью управления процессом разработки;
- необходимостью обеспечить достаточную гибкость программы;
- неудовлетворительными способами описания поведения больших дискретных систем.

Достижения программной инженерии привнесли в практику разработки большую определенность, однако (в отличие от традиционных инженерных дисциплин) успех программного проекта гарантировать нельзя.

Алгоритмы, библиотеки программ, повторно используемые классы, программные компоненты и т. д. представляют собой неполные, фрагментарные решения для того, что требуется описать при разработке программных систем. Проблема состоит в том, чтобы собрать воедино небольшие фрагменты в виде гармоничной *корпоративной системы*, которая удовлетворяет требования сложных бизнес-процессов.

Практика создания ПО способствует разработке систем из настраиваемых программных пакетов — решений на основе COTS-продуктов или ERP-систем (*Enterprise Resource Planning System* — система планирования корпоративных ресурсов). Программный пакет может обеспечить функции стандартного бухгалтерского учета, компьютерной обучающей системы или системы управления кадрами. Акцент смещается от разработки «с чистого листа» к «настройке» программного обеспечения, однако процесс производства все равно занимает значительное место.

Для всякой системы, разрабатываемой «с чистого листа», необходимо сначала создать *концептуальные конструкции* (модели) для конечного решения, которые бы удовлетворяли

специфические потребности организации. После их создания функциональные возможности программного пакета настраиваются в соответствии с концептуальными конструкциями.

Задачи программирования могут быть разными, но характер деятельности по анализу требований и системному проектированию, связанной с этими конструкциями, при разработке «с нуля» аналогичен. Таким образом, концептуальная конструкция (модель) сохраняется, несмотря на всевозможные изменения ее представления (реализации).

Что еще более важно — маловероятно, чтобы организация могла найти программный пакет для автоматизации ее ключевых бизнес-процессов. То, что заставляет компанию работать (и конкурировать), должно разрабатываться «с нуля» (или переделываться из *унаследованной системы*).

Конечно, в любом случае в процессе разработки должны использоваться преимущества *компонентной технологии*. Компонент — это исполняемый программный модуль, реализующий четко определенные функции (сервисы) и коммуникационные протоколы (интерфейсы) взаимодействия с другими компонентами. Компоненты можно сконфигурировать таким образом, чтобы удовлетворить требования к приложению. В настоящее время наибольшую популярность получили следующие компонентные технологии:

- CORBA (*Common Object Request Broker Architecture* — Общая архитектура брокера объектных запросов) от OMG (*Object Management Group*);
- DCOM (*Distributed Component Object Mode* — Распределенная модель компонентных объектов) от *Microsoft*;
- EJB (*Enterprise Java Beans* — Корпоративная технология для *Java*-приложений) от *Sun*.

Пакеты, компоненты и другие подобные методы не изменяют сущности создания программного обеспечения. В частности, принципы и задачи анализа требований и системного проектирования остаются неизменными. Конечный программный продукт может собираться из стандартных или разрабатываемых под заказ компонентов, но сам процесс «сборки» по-прежнему остается искусством.

2.1.2. Вариативные проблемы разработки ПО

Участники проекта по разработке программного обеспечения — это все персоналии, влияющие на разработку системы:

заказчики (пользователи и владельцы системы) и разработчики (менеджеры проекта, аналитики, проектировщики, программисты и т. д.).

Если рассматривать проблему неудачи проектов со стороны заказчиков, то эти причины заключаются в следующем:

- потребности заказчиков не поняты или не полностью зафиксированы;
- требования заказчиков изменяются слишком часто;
- заказчики не готовы выделить достаточно ресурсов под проект;
- заказчики не стремятся к сотрудничеству с разработчиками;
- ожидания заказчиков нереалистичны.

Заказчики, как правило, не понимают, как разрабатывается программное обеспечение. При этом лишь немногие разработчики могут рассказать им с начала и до конца, как будет разрабатываться заказанный продукт, демонстрируя хорошее понимание различных вопросов, которые могут возникнуть по ходу проекта, просто потому, что разработка программного обеспечения весьма сложна.

Кроме того, заказчики на самом деле обычно не заинтересованы принимать участие в сложном процессе, поскольку они плохо понимают происходящее и в такой ситуации предпочитают оставить подробности на усмотрение разработчиков. Они полагают, что достаточно ограничиться совещанием по определению требований в начале работы над проектом, утверждением промежуточных отчетов о состоянии проекта (хотя порой и не соответствующих действительности), тестированием приемки непосредственно перед поставкой продукта и окончательным утверждением системы.

Другими словами, современная практика неучастия заказчиков в работе над проектом часто приводит не только к срыву сроков, превышению бюджета, но и в конечном итоге система оказывается бесполезной для заказчиков. Эти проблемы являются веским аргументом необходимости использования процесса разработки программного обеспечения, который управляет действиями всех его участников — заказчиков, пользователей, разработчиков и руководства.

То, что современная ситуация в разработке программного обеспечения весьма далека от идеала, не в последнюю очередь объясняется и проблемами неудачи проектов по вине разработ-

чиков. В связи с увеличением сложности программных систем растет понимание того, что критическим фактором разработки становятся опыт и знания разработчиков.

Хорошие разработчики могут дать решение. Высококлассные разработчики могут дать значительно лучшее решение, намного быстрее и дешевле. На эту тему существует довольно известная шутка Ф. Брукса: «Великие проекты — удел великих разработчиков». Мастерство и ответственность разработчиков являются факторами, вклад которых в достижение качества и продуктивности программного обеспечения трудно переоценить.

По мнению известного эксперта в программной индустрии Скотта Амблера, корни проблемы неудачных проектов по вине разработчиков кроются в образовательных и социальных аспектах, которые характерны для всех стран.

Подготовка будущих разработчиков программного обеспечения в колледжах и университетах дает им базовые знания и навыки. Далее на профессиональном поприще молодые специалисты в основном изучают технологии, которые они считают наиболее важным аспектом своей деятельности, и работают с ними. Поскольку технологии постоянно изменяются, молодые разработчики имеют тенденцию применять изученную технологию в одном-двух проектах, после чего переходить к изучению новой технологии или последней версии той, с которой они работали раньше.

С. Амблер называет эту тенденцию «врожденным свойством разработчиков программного обеспечения». Проблема состоит в том, что они снова и снова продолжают изучать все те же оттенки того же нижнего уровня базовых знаний.

К счастью, базовые принципы даже после нескольких витков технологий практически не меняются. Это касается и программирования, например, кодирования управления транзакциями, и разработки пользовательских интерфейсов, и организации доступа к базам данных в различных оболочках, и во многих других областях. Понимание неизменности базовых принципов, которые были изучены еще в учебном заведении, приходит к разработчикам позже, когда они переходят в ранги лидеров проекта, руководителей проекта или специалистов по моделированию. На этих должностях от них уже не требуется постоянно и в большом объеме изучать новые технологии.

Таким образом, к тому времени, когда разработчики действительно начинают понимать свое дело, они понемногу перестают заниматься непосредственно разработкой. Но приходит новое поколение молодых специалистов и цикл повторяется. В результате большинство тех, кто активно разрабатывает программное обеспечение, умеют это делать не лучшим образом, а те, у кого это умение на высоте, не занимаются разработкой.

Чтобы нивелировать сложности процесса разработки проекта, связанные с человеческим фактором, целесообразно использовать процесс, позволяющий определять действия и организационные процедуры, направленные на усиление совместной работы в бригаде разработчиков с целью поставки заказчикам высококачественных программных продуктов. В этом случае на модель процесса возлагаются следующие функции:

- установление порядка выполнения действий;
- определение состава и времени поставки артефактов, создаваемых в процессе разработки;
- закрепление действий и артефактов за разработчиками;
- введение критериев отслеживания хода проекта, измерение результатов и планирование будущих проектов.

2.2. Модели процесса разработки программного обеспечения

2.2.1. Жизненный цикл программного обеспечения

Моделирование процесса разработки неразрывно связано с сущностью *жизненного цикла программного обеспечения*, являющегося одним из базовых понятий программной инженерии. Именно модель жизненного цикла любого конкретного программного обеспечения определяет характер процесса его создания.

Жизненный цикл (ЖЦ) программного обеспечения определяется как период времени, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации.

Основным нормативным документом, регламентирующим состав процессов ЖЦ ПО, является международный стандарт ISO/IEC 12207: 1995 «*Information Technology — Software Life Cycle Processes*». Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены

во время создания ПО (его российский аналог ГОСТ Р ИСО/МЭК 12207—99 введен в действие в июле 2000 г.).

В данном стандарте процесс определяется как совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. Каждый процесс разделен на набор действий, каждое действие — на набор задач. Каждый процесс, действие или задача инициируется и выполняется другим процессом по мере необходимости, причем не существует заранее определенных последовательностей выполнения.

В соответствии со стандартом ГОСТ Р ИСО/МЭК 12207—99 все процессы ЖЦ ПО разделены на три группы:

- основные процессы (приобретение, поставка, разработка, эксплуатация, сопровождение);
- вспомогательные процессы (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместная оценка, аудит, разрешение проблем);
- организационные процессы (управление, инфраструктура, усовершенствование, обучение).

Наиболее полный анализ стандарта жизненного цикла программного обеспечения и описание всех групп процессов ЖЦ ПО приведен в работе А. Вендрова [4].

В общем случае жизненный цикл определяется моделью и описывается в форме методологии (метода). Модель жизненного цикла определяет концептуальный взгляд на его организацию и нередко — основные фазы жизненного цикла и принципы переходов между ними.

Модель жизненного цикла ПО — это структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении ЖЦ. Модель ЖЦ ПО включает в себя стадии, результаты выполнения работ на каждой стадии, ключевые события — точки завершения работ и принятия решений.

Стандарт ГОСТ Р ИСО/МЭК 12207—99 не предлагает конкретную модель ЖЦ ПО. Его положения являются общими для любых моделей ЖЦ, методов и технологий создания ПО. Он описывает структуру процессов ЖЦ ПО, не конкретизируя в деталях, как реализовать или выполнить действия и задачи, включенные в эти процессы.

Понятие модели ЖЦ процесса разработки продукта более узкое по сравнению с понятием модели ЖЦ самого продукта, поскольку конкретный процесс разработки соответствует определенной части жизненного цикла продукта (рис. 2.1).

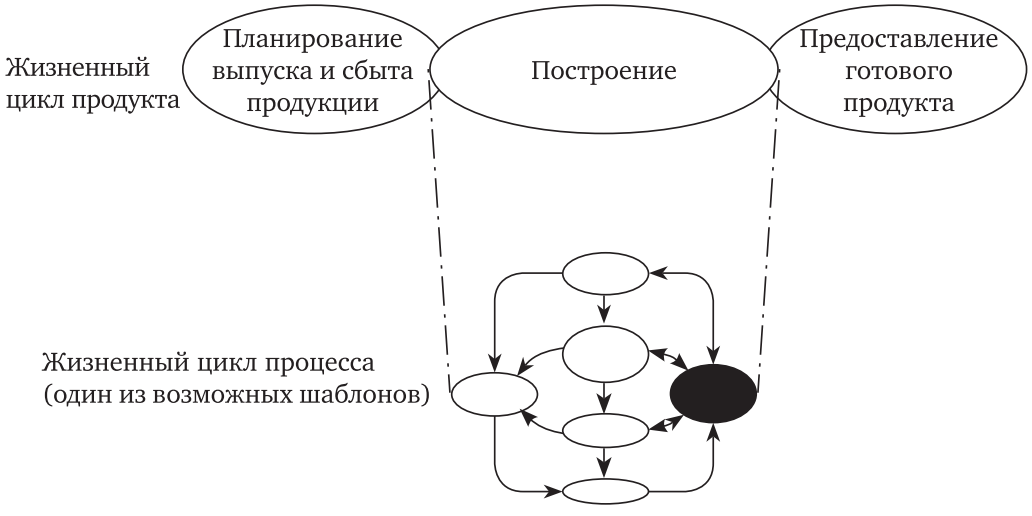

Рис. 2.1. **Соотношение жизненных циклов продукта и процесса разработки**

Модель процесса создания программного обеспечения представляет собой упрощенное описание процесса в виде последовательности практических этапов, необходимых для разработки создаваемого программного продукта.

Подобные модели, несмотря на их разнообразие, служат абстрактным представлением реального процесса создания программного обеспечения. Каждая такая модель представляет процесс создания в некотором ракурсе, используя только определенную часть всей информации о процессе. Модели могут отображать процессы, которые являются частью технологического процесса создания ПО, компоненты программных продуктов и действия людей, участвующих в создании ПО.

Приведем пример упрощенной модели процесса разработки программного обеспечения.

По определению американского аналитика в области программной инженерии Р. Ходгсона (R. Hodgson), процесс разработки системы — это процесс осмысления изобретения и реализации, в соответствии с которым предметная область осмысливается и постигается через феномен, понятия, сущности, действия, роли и утверждения. Это осмысление полностью соответствует анализу.

Однако понимание предметной области также влечет за собой одновременно осмысление структуры, компонентов, вычислительных моделей и других интеллектуальных построений, которые учитываются в области допустимых решений.

Эта изобретательская деятельность полностью соответствует процессу проектирования.

Таким образом, понимание ответов предшествует до некоторой степени пониманию проблемы, что противоречит традиционной философии проектирования. Но такой процесс познания не является противоречивым для когнитивных процессов, к которым относится и процесс проектирования программного обеспечения, и процесс реализации. Эти же соображения относятся к процессу реализации, где эти структуры и архитектурные компоненты проецируются на компиляторы и аппаратные средства.

Согласно упрощенной модели разработка начинается с установления требований предметной области и сбора информации о ней, а заканчивается тестированием и последующим сопровождением. Между ними осуществляются три главных этапа (рис. 2.2):

- определение технических требований и логическое моделирование (анализ);
- архитектурное моделирование (проектирование);
- реализация (кодирование и тестирование).

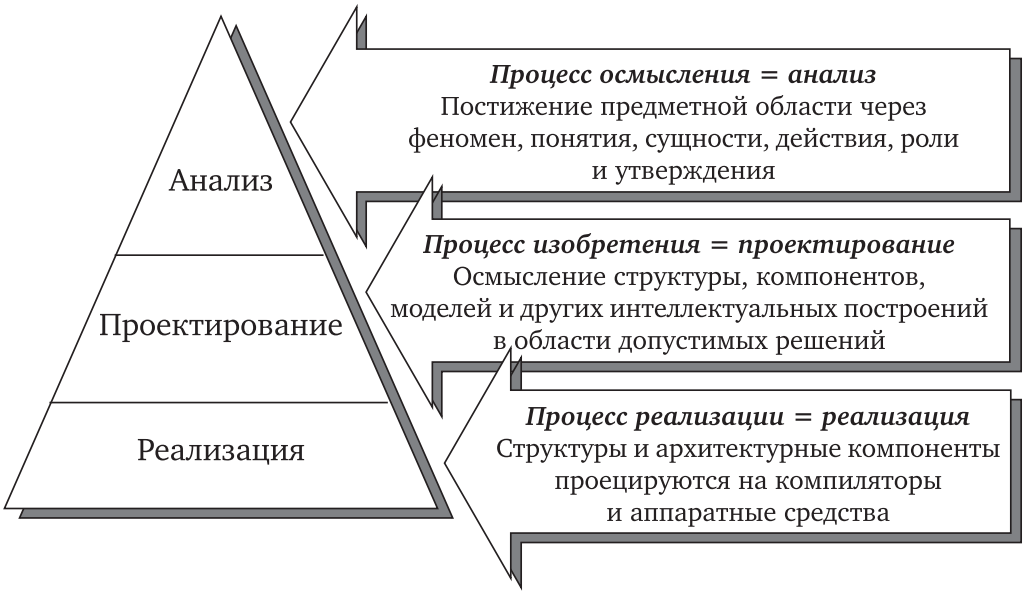

Рис. 2.2. **Основные этапы упрощенного процесса разработки программного обеспечения**

Эта упрощенная модель разработки программного обеспечения допускает итерации и создание прототипов. В реальной жизни процессы определения технических требований и про-

ектирования в значительной степени перекрываются. Это особенно верно для объектно-ориентированного проектирования и анализа, поскольку абстракции для них моделируются скорее на основе абстракций приложения, а не процессоров и дисков.

Как известно, проектирование может быть разделено на логическое и физическое. При объектно-ориентированном проектировании логическая стадия часто неотличима от некоторых частей объектно-ориентированного анализа.

Одной из главных проблем, с которыми сталкиваются при структурном анализе и методах проектирования, является отсутствие пересечения или плавного перехода между двумя стадиями. Это часто приводит к трудностям в отслеживании продуктов проектирования в обратном направлении к первоначальным требованиям пользователя или продуктам анализа.

Для объектно-ориентированного анализа и объектно-ориентированных методов проектирования общими являются следующие основные этапы, хотя их порядок и детали реализации в значительной степени варьируются.

- Поиск способов, с помощью которых система взаимодействует со своим окружением (прецеденты).
- Идентификация объектов и имен их атрибутов и методов.
- Определение связей между объектами.
- Определение интерфейсов каждого объекта, а также обработки исключительных ситуаций.
- Реализация и тестирование объектов.
- Компоновка и тестирование системы.

Анализ — это разбиение задачи на составляющие. В обработке данных это понимается как процесс спецификации структуры и функций системы, независимо от средств реализации или физического разбиения на модули или компоненты. Анализ традиционно выполнялся сверху вниз с использованием структурного подхода или эквивалентного метода, базирующегося на функциональном разбиении и анализе данных.

При этом стратегический анализ высокого уровня, управляемый производственными целями, изолируется от системного. При объектном подходе осуществляется анализ обоих типов (функциональный и системный). Это возможно, поскольку объектно-ориентированный анализ позволяет описывать систему в характеристиках реального мира, и системные абстракции более или менее точно соответствуют сущностям бизнес-процесса.

Объектно-ориентированный анализ — это анализ, который содержит и элемент синтеза. Формулировка требований пользователя и идентификация ключевых объектов предметной области сопровождаются компоновкой этих объектов в структуры, которые на более поздней стадии составят основу физического проектного решения.

Аспект синтеза присутствует обязательно, поскольку анализируется система; другими словами, на предметную область накладывается определенная структура. Это не значит, что проектное решение не будет уточняться. Хорошо детализированный проект может значительно отличаться от лаконичной модели спецификации.

«Идеального» процесса создания программного обеспечения не существует. Следует отметить, что определенные процессы более подходят для создания программных продуктов одного типа и менее — для другого типа программных приложений. Если использовать неподходящий процесс, это может привести к снижению качества и функциональности разрабатываемого программного продукта.

Важно подчеркнуть, что возможно усовершенствование выбранного в качестве основы процесса прежде всего за счет использования современных технологий и включения лучших методов современной инженерии программного обеспечения.

2.2.2. Классические модели процессов создания ПО

Опишем кратко обобщенные модели технологического процесса создания программного обеспечения, основанные на архитектурном подходе. В этом случае можно увидеть всю структуру процесса создания программного обеспечения, абстрагируясь от частных деталей отдельных его этапов.

Эти обобщенные модели не содержат точного описания всех стадий процесса создания систем. Напротив, они являются полезными абстракциями, помогающими применить различные подходы и технологии к процессу разработки.

Каскадная модель

Первая разработанная модель процесса создания программного обеспечения была аналогом моделей других инженерных процессов. Это наиболее известная модель жизненного цикла программного обеспечения — каскадная, или модель «водопада» (*waterfall model*), которую часто называют эквивалентом модели процесса в строительной промышленности (рис. 2.3).

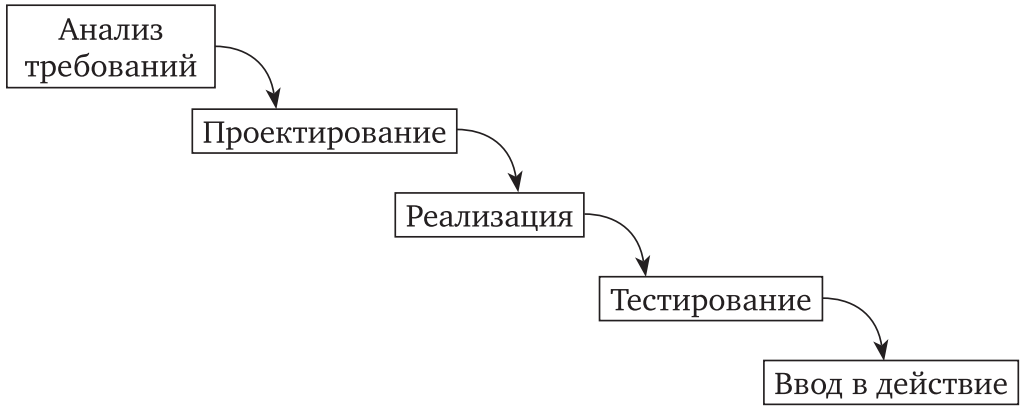

Рис. 2.3. **Каскадная модель разработки программного обеспечения**

Суть каскадной модели в том, что разработка программного обеспечения движется линейно через стадии формирования требований, проектирования, кодирования и тестирования отдельных модулей (компонентов), тестирования сборок и интегрированного тестирования всего конечного продукта.

Каждая стадия заканчивается получением некоторых результатов, которые утверждаются документально и служат в качестве исходных данных для следующей стадии. Таким образом, очередная стадия проекта может начаться только после завершения работ на предыдущей стадии без возврата на пройденные. При этом получение законченного набора документации на каждой стадии и возможность планирования сроков и затрат являются несомненными достоинствами последовательного процесса, каким является каскадный процесс.

Каскадный процесс — разумный подход к работе, но, к сожалению, он применим без исключения только тогда, когда область проблемы хорошо изучена, в то время как область программирования является сравнительно молодой, и технологии программного обеспечения находятся в стадии исследования.

Отсутствие фундаментальных законов программирования на фоне существующего темпа развития программного обеспечения делает эту область весьма рискованной. Кроме того, программно-инструментальные средства, методы и продукты достаточно быстро усовершенствуются и меняются. Поэтому при каждой попытке создания системы, которая сложнее предыдущей, больше или перспективнее, возрастает риск.

Основная проблема каскадного процесса заключается в том, что происходит нарастание риска из-за накопления различных

ошибок, допущенных на ранних стадиях проекта. Если только к концу проекта выявляются такие ошибки, то возврат к предыдущим стадиям с целью исправления ошибок становится крайне дорогостоящим.

В то же время каскадный процесс дает хорошие результаты для небольших проектов, когда удается достаточно точно и полно сформулировать все требования и можно предвидеть, что должно получиться в итоге разработки с учетом всех трудных моментов проекта. Применение каскадного процесса целесообразно также в том случае, если текущий проект в чем-то подобен проекту, ранее завершенному, и в нем используется та же команда разработчиков и та же программно-инструментальная платформа.

Но как только появляется проект, имеющий значительное количество новшеств и системную сложность (по многим параметрам), приводящую к бесчисленному количеству рисков и слабой взаимосвязи, эффективно использовать упрощенный подход последовательных преобразований не представляется возможным. Метод «водопада» с последовательным процессом не позволяет эффективно выявлять и нивелировать последствия подобных рисков.

Итерационная модель

В настоящее время применение каскадной модели процесса разработки и традиционный подход к управлению созданием программного обеспечения сокращается. Современные подходы требуют быстрого создания первоначальной версии системы на ранних этапах процесса разработки, в которой бы уделялось особое внимание высоким рискам, стабилизации базовой архитектуры и уточнению основных требований.

Решение таких задач осуществляется при *итеративной* разработке, которая зарекомендовала себя наилучшим образом при создании множества систем. Разработка выполняется в виде нескольких краткосрочных мини-проектов фиксированной длительности, называемых итерациями.

Каждая итерация включает в себя собственные этапы формирования требований, проектирования, реализации (кодирования) и завершается тестированием, интеграцией и созданием работающей версии некоторой части всей системы.

Разработка протекает как последовательность итераций, надстраивающих архитектурное ядро до тех пор, пока не будут

достигнуты желаемые уровни функциональности и производительности создаваемой системы, которая расширяется и дополняется шаг за шагом. Результатом итерации является *инкремент*, который представляет собой версию системы с дополнительными или усовершенствованными функциональными возможностями по сравнению с предыдущей версией.

Поэтому такой подход иногда называют итеративной и инкрементной разработкой (*iterative and incremental development*) (рис. 2.4).

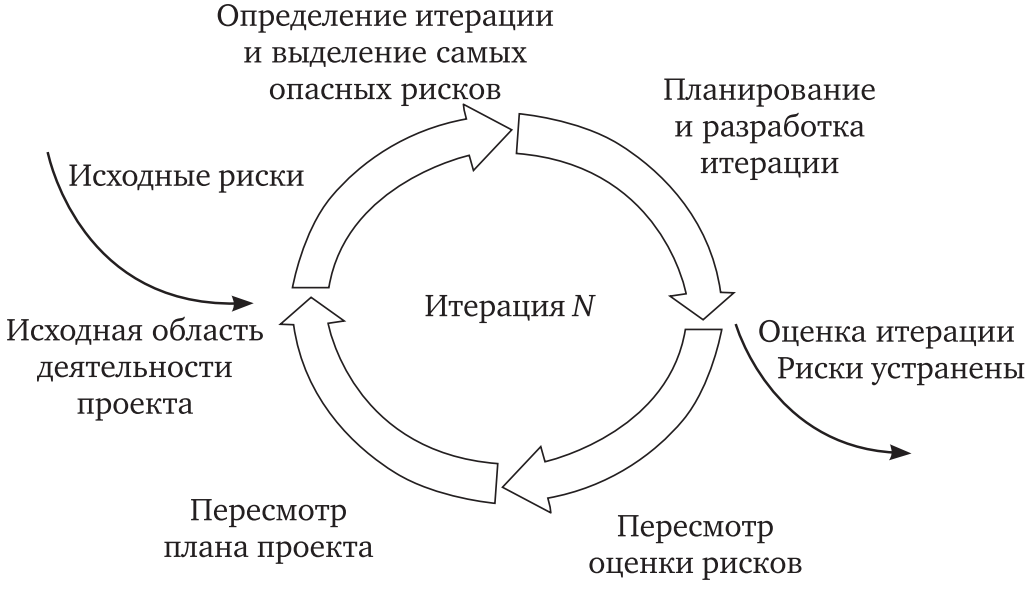

Рис. 2.4. **Итеративная и инкрементная модель процесса разработки**

Хотя, как правило, на каждой итерации определяются новые требования, и система постепенно расширяется, некоторые итерации могут быть полностью посвящены редактированию существующей программы и ее усовершенствованию. Например, одна итерация может потребоваться для повышения производительности системы, а не добавления новой функции.

Итеративный подход акцентирует работу команды в более предсказуемом и повторяемом направлении. Основные преимущества итеративного подхода:

- нивелирование воздействия серьезных рисков на ранних стадиях проекта, пока это еще можно сделать с минимальными затратами;
- возможность организовать обратную связь с будущими конечными пользователями с целью создания системы, реально отвечающей их потребностям;

- акцент усилий на наиболее важные и критичные направления проекта;
- непрерывное итеративное тестирование конечного продукта, позволяющее оценить успешность всего проекта в целом;
- раннее обнаружение несоответствий между требованиями, моделями и программным кодом;
- более равномерная загрузка участников проекта;
- эффективное использование опыта, полученного при реализации каждой итерации, для улучшения самого процесса разработки;
- накопление опыта;
- реальная оценка текущего состояния проекта и, как следствие, большая уверенность заказчиков и непосредственных участников в его успешном завершении.

Экономические преимущества, сопутствующие переходу от каскадной модели к итерационному процессу разработки, значительны, но трудно поддаются количественному определению.

Спиральная модель

Спиральная модель процесса создания программного обеспечения была предложена в середине 1980-х гг. известным исследователем программного обеспечения и технологий Барри Боэмом. В настоящее время модель получила широкую известность. В отличие от рассмотренных моделей, в которых процесс создания программного обеспечения выражается в виде последовательности отдельных процессов с возможной обратной связью между ними, здесь процесс разработки показан в виде спирали (рис. 2.5).

По сути, спиральная модель является вариантом итерационной модели. Каждый виток спирали соответствует одной стадии (итерации) процесса создания программного обеспечения и разбит на четыре следующих сектора.

1. *Определение целей*. Определяются цели каждой итерации проекта. Кроме того, устанавливаются ограничения на процесс создания ПО и на сам программный продукт, уточняются планы производства компонентов. Определяются проектные риски (например, риск превышения сроков или риск превышения стоимости проекта). В зависимости от выявленных рисков могут планироваться альтернативные стратегии разработки программного обеспечения.

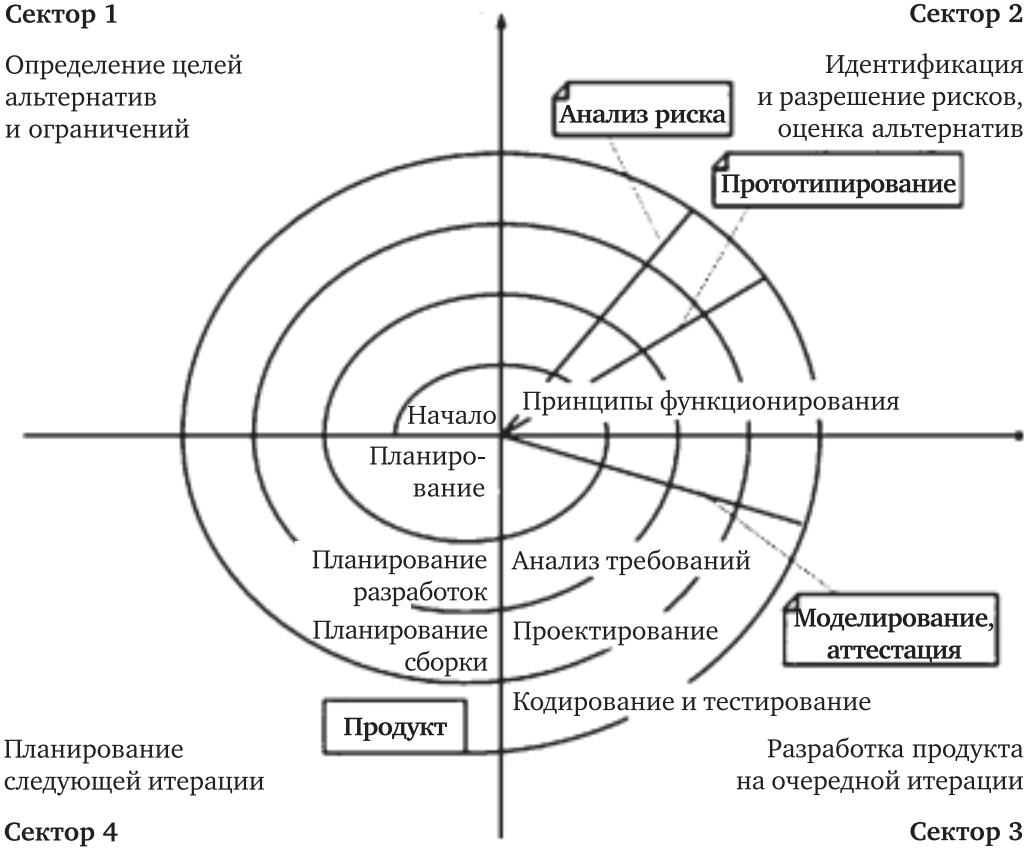

Рис. 2.5. **Спиральная модель процесса разработки программного обеспечения**

2. *Оценка и разрешение рисков*. Для каждого определенного проектного риска проводится его детальный анализ. Планируются мероприятия для уменьшения (разрешения) рисков. Например, если существует риск неверного определения системных требований, планируется разработка прототипа системы.

3. *Разработка и тестирование*. После оценки рисков выбирается модель процесса создания системы. Например, если доминируют риски, связанные с разработкой интерфейсов, наиболее подходящей будет эволюционная модель разработки программного обеспечения с прототипированием. Если основные риски связаны с соответствием системы и спецификации, скорее всего, следует применить модель формальных преобразований. Каскадную модель используют в том случае, если основные риски определены как ошибки, которые могут проявиться на этапе сборки системы.

4. *Планирование*. Здесь пересматривается проект и принимается решение о том, начинать ли следующий виток спирали.

Если принимается решение о продолжении работы, разрабатывается план на следующую стадию проекта.

В зависимости от заданных целей и масштабов проекта результаты каждой итерации могут быть различными. Например, для небольших проектов в конце первого витка спирали может быть создан прототип, фрагмент или версия программного обеспечения.

Для сложных проектов на первой итерации может осуществляться обзор и исследование программных аналогов в конкретной отрасли, а также анализ рисков при внесении изменений в выбранный аналог. Если позитивный риск значительно превышает негативный (т. е. значительно преобладают обстоятельства, свидетельствующие в пользу продолжения работы), то на следующей итерации будут определяться системные требования.

Существенное отличие спиральной модели от других моделей процесса создания программного обеспечения заключается в точном определении и оценке рисков. Например, если при написании программного кода используется новый язык программирования, то риск может заключаться в том, что компилятор этого языка окажется ненадежным или что результирующий код может быть недостаточно эффективным.

Риски могут также заключаться в превышении сроков или стоимости проекта. Таким образом, уменьшение (разрешение) рисков — важный элемент управления системным проектом. Акцент на анализе рисков — главное достоинство спиральной модели.

Первая итерация создания программного обеспечения по спиральной модели, как правило, начинается с тщательной проработки системных показателей (целей системы), таких как эксплуатационные показатели и функциональные возможности системы. Затем формируются и анализируются альтернативные пути достижения этих показателей или целей и стоимость альтернатив.

На второй итерации спиральной модели результаты анализа возможных альтернатив служат источником оценки проектного риска. Причем для оценки рисков используются более детальный анализ альтернатив, прототипирование, имитационное моделирование и т. п. С учетом полученных оценок риска выбирается тот или иной подход к разработке системных компонентов, далее он реализуется, затем осуществляется планирование следующего этапа (итерации) процесса создания программного обеспечения.

В спиральной модели нет фиксированных этапов, таких как разработка спецификации или проектирование. Эта модель может включать в себя любые другие модели разработки систем. Например, на одном витке спирали может использоваться прототипирование для более четкого определения требований (и, следовательно, для уменьшения соответствующих рисков). Но на следующем витке может применяться каскадная модель.

Отметим, что основная проблема спирального цикла — это определение момента перехода на следующую стадию. Для ее решения необходимо вводить временны́е ограничения на каждую итерацию.

Модель быстрой разработки приложений

Еще одним примером реализации итерационной модели разработки программного обеспечения является способ так называемой быстрой разработки приложений — RAD (*Rapid Application Development*), который появился в конце 1980-х гг. Первым формальным названием этой модели было *RIPP* (*Rapid Iterative Production Prototyping*), означающее быстрое итеративное создание прототипов. Как следует из названия метода RAD, он предполагает быструю поставку системных решений.

Технология RAD основана на следующих принципах:

- использование эволюционного прототипирования. Эволюционный прототип системы сохраняется после окончательного выявления требований и используется для создания конечного программного продукта;
- применение CASE-средств с возможностями генерации программ и циклической разработкой с переходом от проектных моделей к программе и обратно;
- ведение разработки немногочисленной высокопрофессиональной командой, члены которой имеют опыт в проектировании, программировании, тестировании ПО и владеющие развитыми инструментальными средствами;
- применение «временны́х блоков» для строгого ограничения итераций, чтобы обеспечить контроль над созданием прототипа. Этот подход препятствует «расползанию рамок проекта»; если проект затягивается, то рамки решения сужаются, чтобы завершить проект своевременно;
- интеграция с современным методом выявления требований JAD (*Joint Application Development* — совместная разработка приложений), состоящего в регулярном проведении семинаров

с привлечением всех участников проекта (заказчиков и разработчиков).

Жизненный цикл программного обеспечения в соответствии с подходом RAD состоит из четырех этапов:

- анализ и планирование требований;
- проектирование;
- реализация;
- внедрение.

Процесс быстрой разработки осуществляется итеративно, каждая стадия (итерация) включает в себя все этапы. Ключевым аспектом является непрерывное установление качества и тестирование разрабатываемых программ в течение всего жизненного цикла процесса. Тестирование может начаться сразу после создания первого прототипа, а затем процедуры тестирования будут развиваться на каждой итерации разработок.

На этапе анализа и планирования требований для проведения первой итерации разработки определяются только наиболее важные характеристики системы. Конечные пользователи и другие лица, формирующие требования, участвуют на каждой стадии проекта и оценивают новые версии продукта. Они могут предлагать изменения и новые требования, которые будут реализованы в следующей версии системы.

Различные модификации метода RAD рассчитаны главным образом на обеспечение быстрой разработки прототипов, а не на такие их системные характеристики, как производительность, удобство эксплуатации или безотказность. Они отличаются по степени вовлеченности пользователей в процесс создания прототипов и по используемым инструментальным средствам. Но на практике эти методы часто применяются совместно для разработки прототипов систем.

При использовании RAD-методов пользовательский интерфейс системы обычно создается интерактивными средствами. Вместо последовательного написания кода разработчик прототипа предпочитает работать с графическими пиктограммами, представляющими функции, данные или компоненты интерфейса пользователя, и соответствующими сценариями управления этими пиктограммами. Программа, готовая к исполнению, генерируется автоматически из визуального представления системы.

Это упрощает разработку интерфейса и уменьшает затраты на прототипирование системы.

Среди разработчиков нет единого мнения о том, для каких систем целесообразно применять технологию RAD. Большинство экспертов считают, что методы быстрой разработки наиболее хорошо подходят для относительно небольших проектов, которые не затрагивают сферу ключевых бизнес-процессов организации.

Следует отметить, что методы быстрой разработки, такие как RAD, особенно эффективны при высоких проектных рисках, например неясных целях проекта, недокументированных процедурах, нестабильных требованиях. Несмотря на то, что эффективность таких разработок достигается за счет повышения трудоемкости и материальных затрат, долговременные вложения в эти методы могут окупиться с лихвой.

Модель динамической разработки

Модель динамической разработки систем DSDM (*Dynamic System Development Method*) отражает процесс, который представляют как «контур элементов управления для быстрой разработки приложений», не предписывающий использование конкретных методов. Метод DSDM был разработан в 1994 г. консорциумом британских пользовательских организаций.

DSDM представляет собой элементарную, но высокоуровневую модель процесса, которую можно изменить индивидуально для любой организации в соответствии с ее требованиями. Этот метод основан на следующих базовых принципах:

- активное вовлечение пользователя обязательно;
- команды имеют право принимать решения;
- важно чаще выпускать новые версии продукта;
- итеративное развитие и инкрементная разработка компонентов необходимы, чтобы прийти к точному коммерческому решению;
- все изменения в процессе разработки обратимы;
- требования определяются на высоком уровне;
- на протяжении всего жизненного цикла постоянно проводится тестирование;
- важны сотрудничество и содействие всех заинтересованных лиц.

На рис. 2.6 изображена модель процесса разработки DSDM. Процесс начинается с оценки осуществимости проекта и изучения предметной области, затем следуют три итеративные и перекрывающиеся фазы. В основе этого процесса лежит преимущественно спиральная модель.

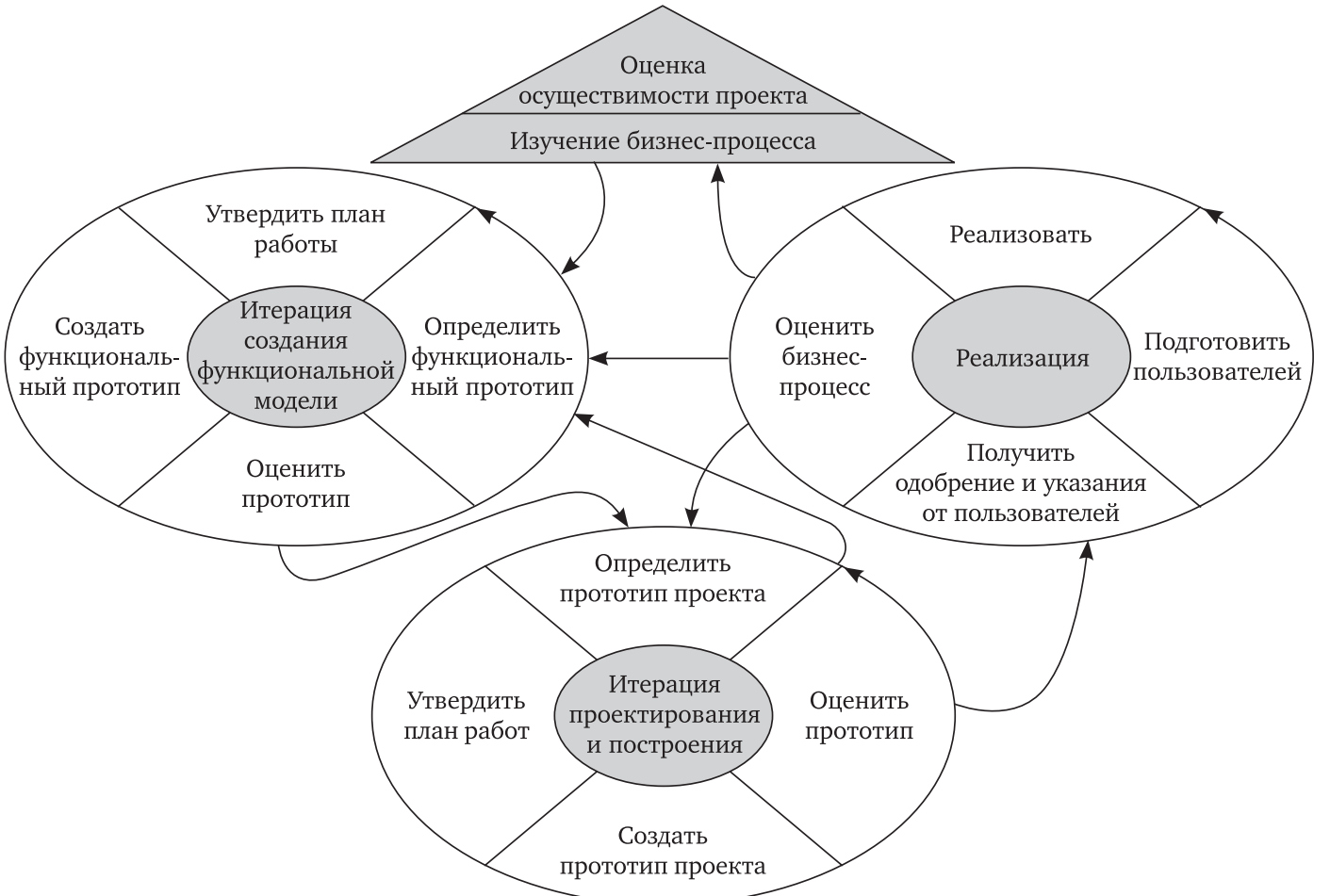

Рис. 2.6. **Модель процесса динамической разработки систем DSDM**

Метод DSDM не вполне подходит для объектно-ориентированной разработки, потому что ему недостает гибкости объектно-ориентированного подхода, но потенциально его фазы можно привести в соответствие с объектно-ориентированной моделью жизненного цикла. Данный метод не предполагает также формального деления программы на более мелкие проекты, которое используется в других методах. Кроме того, в DSDM не различаются жизненные циклы процесса и продукта, хотя он ориентирован на продукт.

Основное достоинство метода DSDM — это наличие структуры, которую можно наполнить конкретными характеристиками проекта. Применение временны́х рамок означает, что этапам соответствуют разработанные компоненты, что четко ориентирует этот подход на продукт. Тестирование проводится на протяжении всего жизненного цикла, а не на заключительном этапе, что ведет к значительному повышению качества продукта и сокращению количества сюрпризов при реализации.

Очень важно, что предоставляемые продукты и документация в DSDM имеют минимальные объемы и в то же время обеспечивают достойное качество и возможность дальнейшей разработки и сопровождения. Но, как показывает практика, без объектно-ориентированного подхода нельзя гарантировать простоту сопровождения системы в случае дальнейшего развития (изменений и дополнений) требований.

Как уже отмечалось, метод DSDM ориентирован на продукт, но в процессе нет акцента на производство действующих компонентов. В результате по методу DSDM конечным продуктом временно́го блока может стать модель данных, в то время как методом RAD произведенные объектные модели доводятся до уровня выполняемых модулей (хотя бы потенциально).

В DSDM основное внимание уделяется интерфейсу системы. Это идеально подходит для простых информационно-управляющих систем или для спецификации такого машинного оборудования, как торговые автоматы или коммутаторы дистанционной связи, но это далеко не то, что нужно для более сложных систем. Возможно, из-за этого ограничения метод DSDM не признали подходящим для приложений с какими-либо вычислительными сложностями.

Метод DSDM предлагает следующую классификацию моделей.

- Коммерческие прототипы.
- Прототипы возможностей/прототипы проекта.

- Прототипы производительности.
- Прототипы применимости.

Здесь отражено разделение итераций проектирования и реализации функциональных требований в жизненном цикле метода DSDM и его несовместимость с более целостными объектно-ориентированными подходами.

В заключение обзора моделей процесса обобщим значение их эволюции для программной инженерии.

В 1950-х гг., на первом этапе развития вычислительной техники программы разрабатывались методом *ad hoc* (специально для данного случая), и каждая программная система была уникальным, индивидуально созданным интеллектуальным продуктом.

Не существовало понятий повторного использования и взаимозаменяемости отдельных частей или формализованного проектирования. Процесс управления такими системами и их расширения представлял определенные трудности. При каждом изменении получалась новая система, еще более сложная, чем исходная.

В 1960-х гг. усилия по созданию более простого в управлении программного обеспечения привели к первому существенному пересмотру философии его развития. Метод *ad hoc* уступил место каскадному подходу, в соответствии с которым процесс создания программного обеспечения состоял из некоторого числа последовательных формально законченных фаз.

Последующие этапы совершенствования процесса разработки программного обеспечения характеризуются разработками технологий и методов, которые являются результатом рационального сочетания усовершенствованных моделей каскадной и итерационной разработки. Самыми известными вариантами таких технологий являются *RUP* (*Rational Unified Process*), *MSF* (*Microsoft Solution Framework*) и *XP* (*Extreme Programming*).

Глава 3
ВИЗУАЛЬНОЕ МОДЕЛИРОВАНИЕ СИСТЕМ

В настоящее время моделирование программного обеспечения, особенно объектно-ориентированных разработок, является важной частью программной инженерии. Моделирование широко распространено во всех инженерных дисциплинах, в значительной степени из-за того, что оно реализует принципы декомпозиции, абстракции и иерархии. Общим свойством всех моделей является их подобие оригинальной системе или системе-оригиналу. При этом процесс построения и последующего применения моделей для получения информации о свойствах или поведении системы-оригинала получил название «моделирование».

3.1. Цели и значение моделирования

Визуальным моделированием (*visual modelling*) называется процесс графического представления модели с помощью некоторого стандартного набора графических элементов [6]. Общение между пользователями, разработчиками, аналитиками, тестировщиками, менеджерами и всеми остальными участниками проекта является основной целью визуального моделирования. Общение можно обеспечить и с помощью невизуальной (текстовой) информации, но сложная информация понимается легче, если она представлена визуально, а не описана в тексте.

Любое программное обеспечение является моделью некоторых явлений или процессов реального мира. При разработке программного обеспечения существует несколько методов моделирования. Важнейшие из них — алгоритмический и объектно-ориентированный.

Алгоритмический метод представляет собой традиционный подход к созданию программного обеспечения. Основным строительным блоком является процедура или функция, а вни-

мание уделяется, прежде всего, вопросам передачи управления и декомпозиции больших алгоритмов на меньшие. При изменении требований или увеличении размера сопровождать их становится сложнее.

Наиболее современным подходом к моделированию программного обеспечения является *объектно-ориентированный*, при котором основным строительным блоком является объект или класс. В самом общем смысле объект — это сущность, извлекаемая из словаря предметной области или решения, а класс является описанием множества однотипных объектов. Каждый объект обладает идентичностью, состоянием и поведением.

Объектно-ориентированная технология моделирования программных систем, отражающая иерархию классов и объектов, является в настоящее время преобладающей для построения систем в различных областях, любой степени сложности [3].

Модель — это абстракция, описывающая моделируемую систему с определенной точки зрения и на определенном уровне абстрагирования. Под точкой зрения понимается, например, в данном контексте представление требований к системе или представление проектирования, а уровень абстракции в модели отражает степень существенности влияния на результат включенных элементов.

Система может быть описана с разных точек зрения, для чего используются различные модели, каждая из которых, следовательно, является семантически замкнутой абстракцией системы. Модель может быть структурной, подчеркивающей организацию системы, или поведенческой, отражающей ее динамику. При этом модели помогают добиться лучшего понимания создаваемой системы, что зачастую приводит к ее упрощению и возможности повторного использования ее компонентов.

Моделирование позволяет проследить путь от запросов пользователей к требованиям, модели и затем к коду и обратно, не теряя при этом наработок. При моделировании решают следующие задачи:

- определение структуры или поведения системы;
- получение шаблона, позволяющего сконструировать систему;
- документирование принимаемых решений на основе полученных моделей.

Особое значение имеет моделирование сложных систем, поскольку восприятие сложной системы как единого целого может исказить ее отдельные аспекты.

Объектно-ориентированные модели раскрывают весь спектр важнейших конструкторских решений, которые необходимо рассматривать при разработке сложной системы, и таким образом ведут к созданию проектов, обладающих пятью атрибутами хорошо организованных сложных систем:

- соответствие заданным (возможно, неформальным) функциональным спецификациям;
- согласованность с ограничениями, накладываемыми оборудованием;
- соответствие явным и неявным требованиям по эксплуатационным качествам и ресурсопотреблению;
- соответствие явным и неявным критериям дизайна продукта;
- соответствие требованиям к самому процессу разработки, как, например, продолжительность и стоимость, а также привлечение дополнительных инструментальных средств.

3.2. Принципы моделирования

Длительный опыт использования моделирования во всех инженерных дисциплинах позволил сформулировать основные принципы создания моделей программных систем.

Первый принцип: *выбор модели оказывает определяющее влияние на подход к решению проблемы и на то, как будет выглядеть это решение*.

Если программная система должна работать с большими базами данных или производить сложные математические расчеты, то основное внимание будет уделяться моделям «сущность-связь», где поведение инкапсулировано в триггерах и хранимых процедурах.

Структурный аналитик, как правило, создает модель, в центре которой находятся алгоритмы и передача данных от одного процесса к другому. Результатом труда разработчика, использующего объектно-ориентированную технологию, будет модель системы, архитектура которой основана на множестве классов и образцах взаимодействия, определяющих, как эти классы действуют совместно.

Второй принцип: *каждая модель может быть воплощена с различной степенью абстракции*.

Например, при моделировании компьютерных обучающих систем на этапе специфицирования требований целесообразно представить простую, быстро созданную и легко модифицируемую модель пользовательского интерфейса.

На этапе специфицирования межсистемных интерфейсов необходимо создавать их модели с высоким уровнем детализации. Для аналитика или конечного пользователя наибольший интерес представляют модели, отвечающие на вопрос «что», а для разработчика — на вопрос «как».

В общем случае лучшей моделью будет та, которая представляет уровень детализации, соответствующий категории исследователя (или пользователя) и цели исследования (или использования) модели.

Третий принцип: *лучшие модели — те, что ближе к реальному отражению системы*.

Поскольку модель всегда упрощает реальность, то задача состоит в том, чтобы это упрощение не повлекло за собой существенные потери при реализации системы. Различие модели и реальной системы должно быть проанализировано и учтено в процессе разработки.

Рассматриваемый принцип легче реализуется при объектно-ориентированном подходе, чем при структурном. Это следует из того, что «ахиллесовой пятой» структурного анализа является несоответствие модели, принятой в нем, и модели проекта.

Если этот разрыв не будет устранен, то поведение созданной системы с течением времени начнет все больше отличаться от задуманного. При объектно-ориентированном подходе можно объединить все почти независимые представления системы в единое семантическое целое.

3.2.1. Принцип многомодельности

Важным принципом моделирования сложных программных систем является принцип многомодельности. Он формулируется следующим образом.

Нельзя ограничиваться созданием только одной модели. Наилучший подход при разработке любой нетривиальной системы — использовать совокупность нескольких моделей, почти независимых друг от друга.

Определение «почти независимые» означает, что модели могут создаваться и изучаться по отдельности, но вместе с тем остаются взаимосвязанными.

Принцип многомодельности применительно к методологии объектно-ориентированного анализа и проектирования может быть реализован моделями, которые показаны на рис. 3.1.

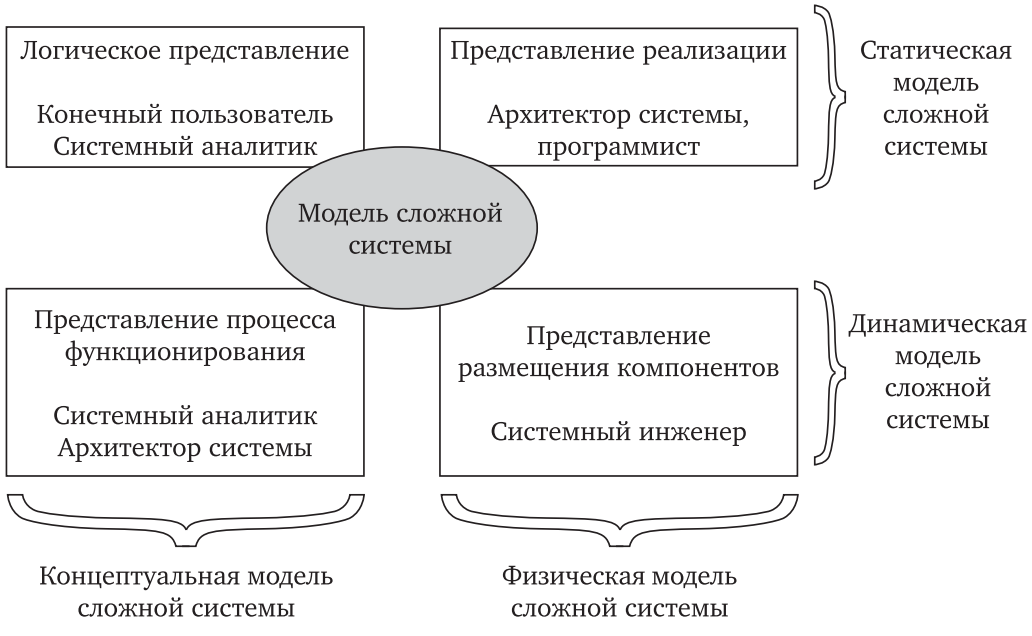

Рис. 3.1. **Представление сложной системы в виде моделей**

Эти модели считаются главными в объектно-ориентированном подходе. В совокупности они семантически достаточно информативны и универсальны, чтобы разработчик мог реализовать все стратегические и тактические решения, которые он должен принять при анализе системы и формировании ее архитектуры. Кроме того, эти модели достаточно полны, чтобы служить техническим проектом реализации практически на любом объектно-ориентированном языке.

Создавая визуальную модель системы, можно показать ее работу на различных уровнях: моделировать взаимодействие между пользователями и системой, между объектами внутри системы и даже между различными системами.

Управление моделями облегчается средствами визуального моделирования, так как возможно предъявление модели с различной степенью полноты. Визуальное моделирование способствует поддержанию непротиворечивости артефактов системы: требований, проектов и реализаций, что является одной из сложнейших задач для команды разработчиков.

Модели используются практически каждым членом группы разработчиков, начиная с руководителя (координатора) проекта, работа которого связана с системой в целом, и заканчивая программистом, отвечающим за программную реализацию компонентов системы (рис. 3.2).

Рис. 3.2. **Модель как главный артефакт процесса разработки**

Созданные визуальные модели представляют всем заинтересованным сторонам, которые могут извлечь из них ценную информацию. Например, глядя на модель, пользователи визуализируют свое взаимодействие с системой. Аналитики увидят взаимодействие между объектами модели. Разработчики поймут, какие объекты нужно создать и что эти объекты должны делать. Тестировщики визуализируют взаимодействие между объектами, что позволит им построить тесты. Менеджеры увидят как всю систему в целом, так и взаимодействие ее частей. Руководители информационной службы, глядя на высокоуровневые модели, поймут, как взаимодействуют друг с другом системы в их организации.

3.3. Графические нотации моделирования

Система обозначений, или нотация (*notation*), представляющая собой совокупность графических объектов, используемых в моделях, является синтаксисом языка моделирования.

Об обозначениях в разработке программного обеспечения имеется достаточно много литературы. И. Грэхем дал обзор ряда нотаций, специфичных для объектно-ориентированных методов [5].

По определению Гради Буча, нотация выполняет следующие функции [3]:

— играет роль языка, используемого для описания неочевидных выводов, которые не проистекают непосредственно из кода как такового;

— сообщает семантику всех стратегических и тактических решений;

— обеспечивает форму представления, достаточно конкретную для восприятия человеком, и возможности манипуляции ею с помощью инструментальных средств.

Предоставляя более формализованные нотации для отображения и визуализации абстракций программного обеспечения, объектно-ориентированная технология оказывает основное влияние на экономические показатели технологии разработки, которое заключается в уменьшении общего размера продукта.

Язык должен обладать мощной *визуальной* составляющей для построения моделей на различных уровнях абстракции и детализации и *декларативной семантикой* для фиксации «процедурного» значения в форме «декларативного» предложения. Иначе говоря, мы должны иметь возможность сказать, «что» именно необходимо сделать, без описания механизма реализации.

Нотация должна быть понятна всем заинтересованным сторонам, иначе визуальная модель будет бесполезна. Множество разработчиков предлагали свои варианты решения этого вопроса. Из них наибольшую поддержку получили метод обозначений Буча (Booch), обозначения по методу объектного моделирования OMT (*Object Modeling Technology*) и унифицированный язык моделирования UML (*Unified Modeling Language*).

Большинство производителей поддержало язык UML. В результате такие комитеты по стандартам, как ANSI и Object Management Group (OMG), в 1995 г. приняли спецификацию UML 1.1 в качестве стандарта языка моделирования. Подробнее о языке UML сказано в гл. 1 второй части книги.

Глава 4
ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И СРЕДСТВА АВТОМАТИЗАЦИИ

4.1. Характеристика и классификация CASE-средств

Революционным шагом в индустрии разработки сложных программных систем стало появление специализированных программно-технологических средств для разработки проектов CASE-средств.

Термин CASE (*Computer-Aided Software Engineering* — автоматизированная разработка программного обеспечения) сегодня понимается достаточно широко. Первоначальное значение термина, ограниченное вопросами автоматизации разработки программного обеспечения, в настоящее время приобрело новый смысл, и теперь это понятие охватывает процесс разработки сложных программных систем в целом.

В соответствии с международным стандартом ISO/IEC 14102:1995 (E) под CASE-*средством* понимается программное средство, поддерживающее процессы жизненного цикла программного обеспечения (определенные в стандарте ISO/IEC 12207:1995).

В рамках программной инженерии CASE-технология представляет собой методологию проектирования программных систем, а также набор инструментальных средств (CASE-средств), позволяющих в наглядной форме моделировать предметную область, анализировать эту модель на всех этапах разработки и сопровождения систем программного обеспечения и разрабатывать приложения в соответствии с потребностями пользователей.

Согласно западным исследованиям CASE-технология попала в разряд наиболее стабильных информационных технологий. Ведущие российские эксперты в области объектно-ориентированных технологий подтверждают значимость CASE-средств для моделирования сложных программных систем, отмечая следующие причины использования графических языков моделирования.

- *Изучение методов проектирования*. Процесс освоения объектно-ориентированных методов в профессиональной среде и сфере образования осуществляется значительно эффективнее с использованием графических средств, предоставляемых CASE-технологиями.
- *Общение с экспертами организации*. Графические модели позволяют составить внешнее представление о системе и объясняют, что эта система будет делать.
- *Получение общего представления о системе*. Графические модели помогают быстро получить общее представление о системе, выявить типы абстракций внутри системы, определить необходимые уточнения в модели.

CASE-средства позволяют получить описание работы создаваемой системы раньше, чем ее построили. С их помощью можно оптимизировать подготавливаемые решения, включая анализ требований к системе, проектирование прикладного программного обеспечения и баз данных, генерацию кода, тестирование, обеспечение качества, управление проектом, а также другие процессы.

CASE-средства, предназначенные для анализа спецификаций и проектирования ПО, иногда называют CASE-средствами верхнего уровня, поскольку их применяют на начальной стадии разработки программных систем. В то же время CASE-средства, используемые для разработки и тестирования программного обеспечения (отладчики, системы анализа программ, генераторы тестов и редакторы программ), называют CASE-средствами нижнего уровня.

Вместе с системным программным обеспечением и техническими средствами CASE-средства образуют среду разработки программных систем (*Software Engineering Environment*).

Современным CASE-средствам присущи следующие основные особенности:

- наличие мощных графических средств для описания и документирования системы, обеспечивающих удобный

интерфейс с разработчиком и развивающих его творческие возможности;

- интеграция отдельных компонентов CASE-средств, обеспечивающая управляемость процесса разработки программного обеспечения;
- использование специальным образом организованного хранилища проектных метаданных (репозитория).

Интегрированное CASE-средство (комплекс средств, поддерживающих полный жизненный цикл программного обеспечения) содержит центральный репозиторий, который обеспечивает хранение, доступ, обновление, анализ и визуализацию всей информации по проекту. Репозиторий является базой для стандартизации документации по проекту и контроля проектных спецификаций. Все отчеты строятся автоматически по содержимому репозитория.

Наиболее трудоемкими стадиями разработки программного обеспечения являются стадии формирования требований и проектирования, в процессе которых CASE-средства обеспечивают качество принимаемых технических решений и подготовку проектной документации. При этом большую роль играют автоматизированные методы визуального представления информации.

По замечанию Лешека Мацяшека, видного ученого и признанного профессионала в области программной инженерии, «моделировать программные артефакты с помощью карандаша и бумаги уместно только в аудитории, но никак ни при работе над реальным проектом» [8].

Современный рынок программных средств насчитывает около 300 различных CASE-средств, наиболее мощные из которых используются практически всеми ведущими западными фирмами. На российском рынке CASE-средства появились в 1992 г. Первыми такими средствами были *ProKit Workbench* фирмы *McDonnell Douglas Information Systems*, *Design/IDEF* фирмы *Metasoftware* и отечественная разработка фирмы «Эйтикс» под названием CASE Аналитик.

Классификация CASE-средств помогает понять их основные типы и роль, которую они играют в поддержке процессов создания программного обеспечения. Существует несколько различных классификаций CASE-средств, которые отражают различные аспекты их применения.

Распространенными вариантами классификаций CASE-средств являются:

- по выполняемым функциям;
- по типам процессов разработки, которые они поддерживают;
- по категориям, где CASE-средства классифицируются по степени интеграции программных модулей, поддерживающих различные процессы разработки.

Классификация CASE-средств по категориям определяет степень интеграции по выполняемым функциям и основана на поддержке большинства процессов разработки программного обеспечения с помощью данного средства. Классификация CASE-средств может содержать следующие категории.

1. *Вспомогательные программы*, поддерживающие отдельные процессы разработки ПО: проверка непротиворечивости архитектуры системы, компиляция программ, сравнение результатов тестов и т. п. Вспомогательные программы могут быть универсальными функционально-законченными средствами или входить в состав инструментальных средств.

2. *Инструментальные средства* поддерживают определенные процессы разработки ПО, например специфицирование требований, проектирование и т. д. Обычно инструментальные средства являются набором вспомогательных программ, которые в большей или меньшей степени интегрированы.

3. *Рабочие среды разработчика* поддерживают все или большинство процессов разработки ПО. Рабочие среды обычно включают в себя несколько различных интегрированных инструментальных средств.

На практике границы между CASE-средствами разных категорий размыты. Вспомогательную программу можно приобрести как отдельный продукт, но ее можно использовать для поддержки различных процессов разработки.

4.2. Технологии и инструментальные средства *IBM Rational*

На сегодняшний день все ведущие компании — лидеры в разработке технологий и программных продуктов (*IBM, Microsoft, Oracle, Borland, Computer Associates* и др.) располагают развитыми технологиями и средствами, предназначенными для автоматизации процессов разработки программного обеспечения.

Представим краткий обзор технологий создания программного обеспечения и CASE-средств компании *IBM Rational*.

До 2003 г. компания *Rational Software* представляла собственные разработки технологий создания программного обеспечения, которые получили признание в мировой программной индустрии. В 2003 г. корпорация *Rational Software* вошла в состав компании *IBM*, и технологии *Rational* стали неотъемлемой частью портфеля программного обеспечения *IBM*. С этих пор продукты корпорации распространяются под торговой маркой *IBM Rational*, которая выпускает CASE-средства, системы автоматизированного проектирования программного обеспечения, а также средства управления проектами.

Рассмотрим некоторые инструментальные средства *IBM Rational* для решения основных задач разработки.

Средства по управлению проектами и портфелями

Управление проектом по разработке программного обеспечения — это своего рода искусство балансирования между конкурирующими целями, рисками, различными ограничениями и обстоятельствами. Основной задачей данного процесса является обеспечение успешной поставки продукта, удовлетворяющего потребностям заказчиков и конечных пользователей.

Основные цели управления проектами:

- организация процесса управления проектом, планирование проекта на протяжении всего жизненного цикла и отдельной итерации;
- соблюдение основных принципов планирования, управления персоналом, выполнения работ и мониторинга проекта с помощью соответствующих метрик;
- эффективное управление рисками.

Управление портфелем проектов — это подход, который позволяет держать под контролем широкий диапазон проектов и ресурсов, обеспечивая необходимый уровень их управляемости. Помимо управления единым финансовым портфелем, управление проектами включает в себя множество различных процессов — управление ресурсами, затратами, рисками, качеством, а также другие связанные с этим процессы, и все они должны выполняться совместно.

В составе новой линейки средств *IBM Rational* предлагается инструмент для управления проектами и портфелями: *IBM Rational Portfolio Manager* (рис. 4.1).

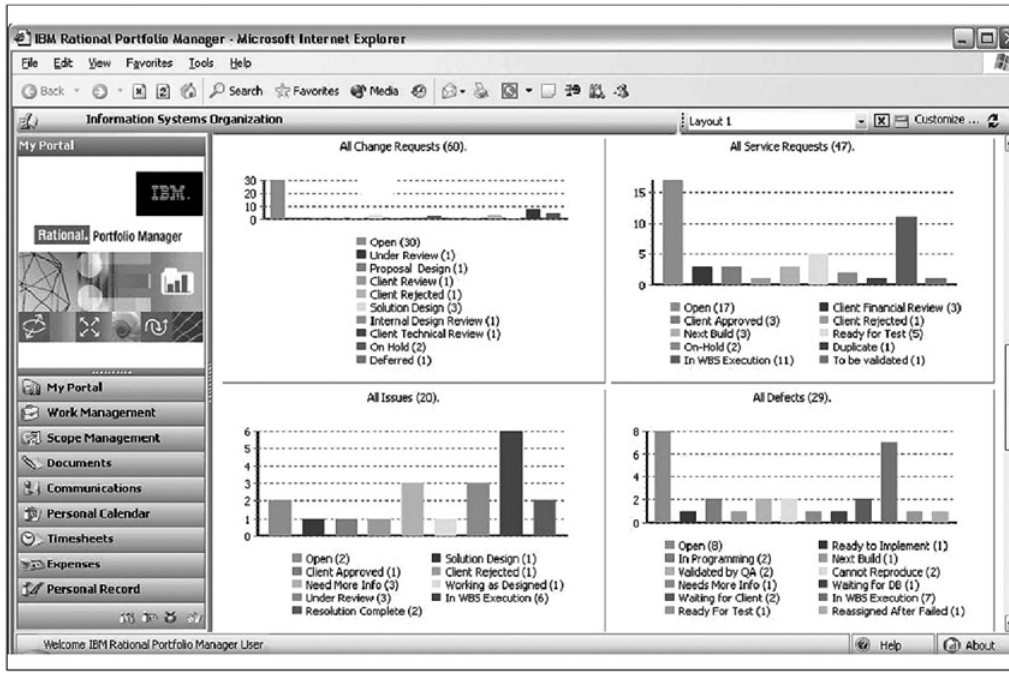

Рис. 4.1. **Рабочее окно *IBM Rational Portfolio Manager***

Этот инструмент играет ключевую роль в обеспечении процесса разработки программного обеспечения, управляемого бизнес-целями компании. Он предоставляет командам разработчиков возможности по управлению проектами, значительно превышающие возможности аналогов, например программы *MS Project*, но в то же время позволяет интегрироваться с ней.

Функциональные возможности *IBM Rational Portfolio Manager* обеспечивают высокий уровень прозрачности и предсказуемости выполнения проектов, что крайне важно для стратегических решений, принимаемых менеджерами проектов и руководством.

Средства поддержки основных процессов создания ПО

Одной из ключевых особенностей новой линейки продуктов *IBM Rational* является более тесная интеграция предлагаемых средств поддержки жизненного цикла разработки.

Наиболее полно интеграция инструментальных средств основных процессов создания приложений на платформе *J2EE* — моделирования, разработки и тестирования — реализована в пакете *IBM Rational Professional Bundle*. Наличие единой интегрированной среды разработки на основе *Eclipse* позволяет легко переходить от моделирования высокоуровне-

вых представлений бизнес-процессов к проектированию модулей разрабатываемой программной системы с последующей их реализацией и тестированием.

Моделирование и проектирование. Интегрированное решение по проектированию и разработке приложений — *IBM Rational Software Architect* (RSA), которое входит в *IBM Rational Professional Bundle*, содержит средства моделирования *IBM Rational Software Modeler* (RSM) и быстрой разработки приложений *IBM Rational Application Developer* (RAD).

Одним из преимуществ новых средств моделирования является возможность автоматизированного преобразования моделей, позволяющая быстро переходить от высокоуровневого моделирования к разработке и тестированию приложений. Это позволяет более эффективно использовать шаблоны проектирования (паттерны) и лучшие проектные решения для создания высококачественного кода и повышения общей эффективности проектов разработки программных систем.

Специализированные средства моделирования позволяют автоматизировать повторяющиеся действия, повышая не только продуктивность, но и уровень зрелости процесса разработки программного обеспечения в целом. Во многом этому способствует использование стандартизованного языка моделирования *Unified Modeling Language*.

Для поддержки инструментальных средств и платформ, не вошедших в пакет *IBM Rational Software Architect*, используются средства моделирования из пакета *IBM Rational Suite*. В состав *IBM Rational Suite* входит пакет *IBM Rational Rose* — популярное средство визуального моделирования, которое считается стандартом де-факто среди средств визуального проектирования приложений (рис. 4.2).

Инструментальное средство *IBM Rational Rose* расширяет возможности моделирования программных систем, выходящих за рамки платформы *J2EE* и инструментальных средств моделирования в составе *IBM Rational Professional Bundle*.

С помощью *Rational Rose* можно визуализировать, анализировать и уточнять требования к создаваемому продукту. Возможность описывать графический интерфейс отдельно от бизнес-логики приводит в конечном счете к лучшим результатам на уровне всего проекта. Использование единого инструмента моделирования на протяжении всего жизненного цикла разработки помогает создавать «идеальную» систему.

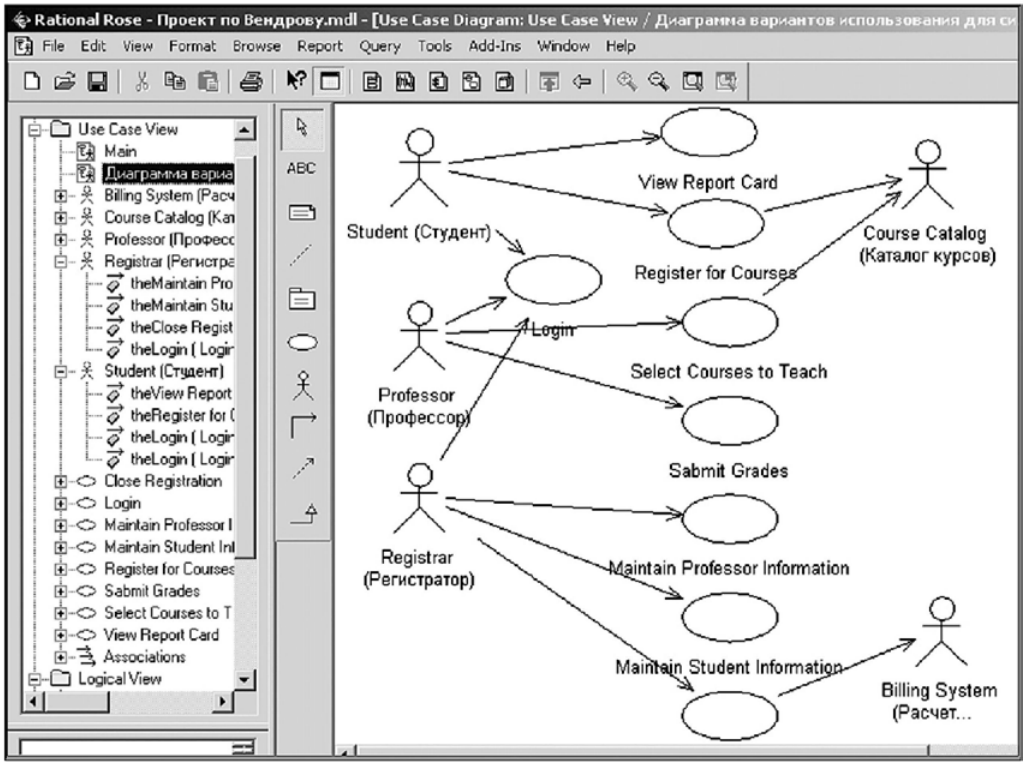

Рис. 4.2. **Рабочее окно программы *Rational Rose***

Rational Rose предлагает плавный процесс разработки информационных систем. Любые модели, создаваемые с помощью данного средства, являются взаимосвязанными: бизнес-модель, функциональная, анализа, проектирования, базы данных, компонентов и физического развертывания системы.

Возможности по созданию и использованию шаблонов архитектурных решений позволяют эффективно использовать опыт, накопленный в предыдущих проектах.

Rational Rose является ведущим инструментом визуального моделирования в программной индустрии, благодаря полноценной поддержке языка моделирования *UML* и многоязыковой командной разработке. Инструмент полностью поддерживает компонентно-ориентированный процесс создания информационных систем.

4.3. Унифицированный процесс разработки

4.3.1. Развитие процесса

Одной из наиболее совершенных технологий, претендующих на роль мирового корпоративного стандарта, является

разработка корпорации *IBM Rational — Rational Unified Process* (RUP) [7]. Унифицированный процесс является результатом трех десятилетий разработки и практического использования.

Развитие *Rational Unified Process* (рис. 4.3) отражает коллективный опыт специалистов и различных компаний.

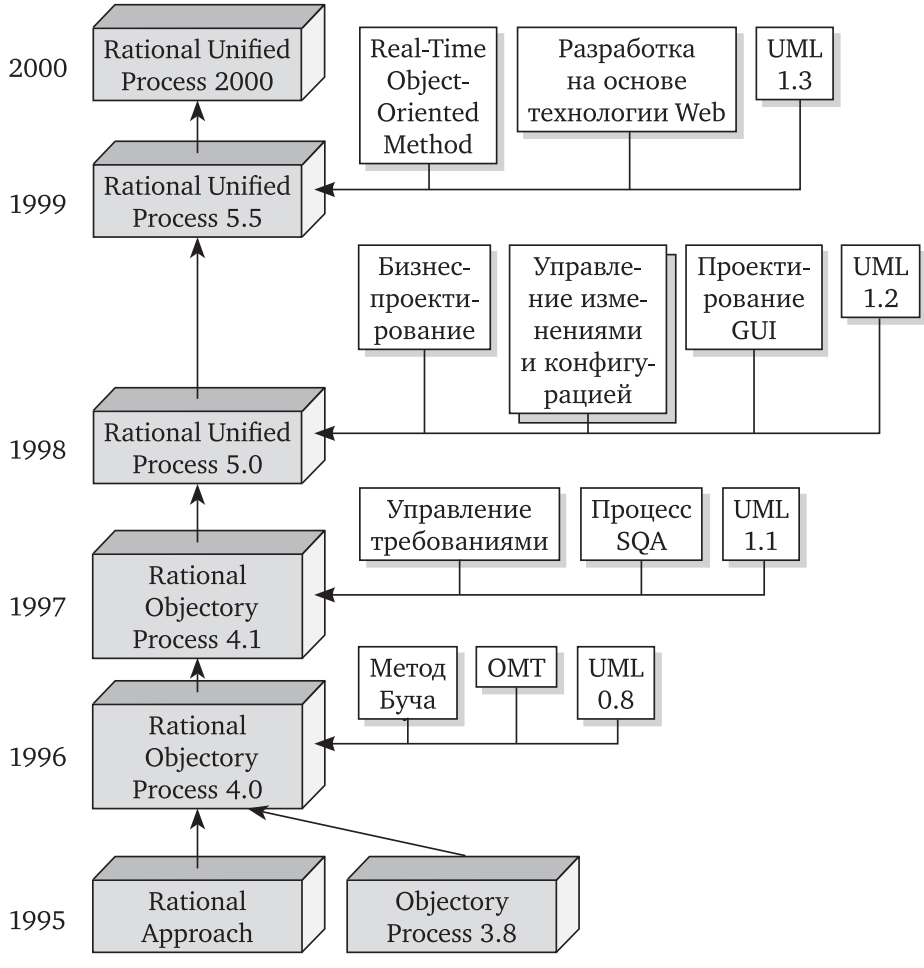

Рис. 4.3. **Разработка *Rational Unified Process***

Rational Unified Process как продукт был разработан в 1998 г. на основе *Objectory Process*, впервые выпущенного в 1987 г. компанией *Ericsson*.

Основанный на концепции использования прецедентов (вариантов использования) и объектно-ориентированном проектировании, этот процесс получил признание в индустрии программных средств и был принят (полностью или в качестве составной части) различными компаниями по всему миру.

В основе многих программных методологий, объединяющих инженерные методы создания программного обеспечения,

лежит «пошаговый подход». Он определяет этапы жизненного цикла, контрольные точки, правила работ для каждого этапа и тем самым упорядочивает проектирование и разработку программных систем.

Для каждого этапа жизненного цикла методология задает:

- состав и последовательность работ, а также правила их выполнения;
- распределение полномочий среди участников проекта (роли);
- состав и шаблоны формируемых промежуточных и итоговых документов;
- порядок контроля и проверки качества.

На рис. 4.4 показано рабочее окно *Rational Unified Process*, в котором отражены состав работ и создаваемые артефакты при разработке архитектуры программной системы.

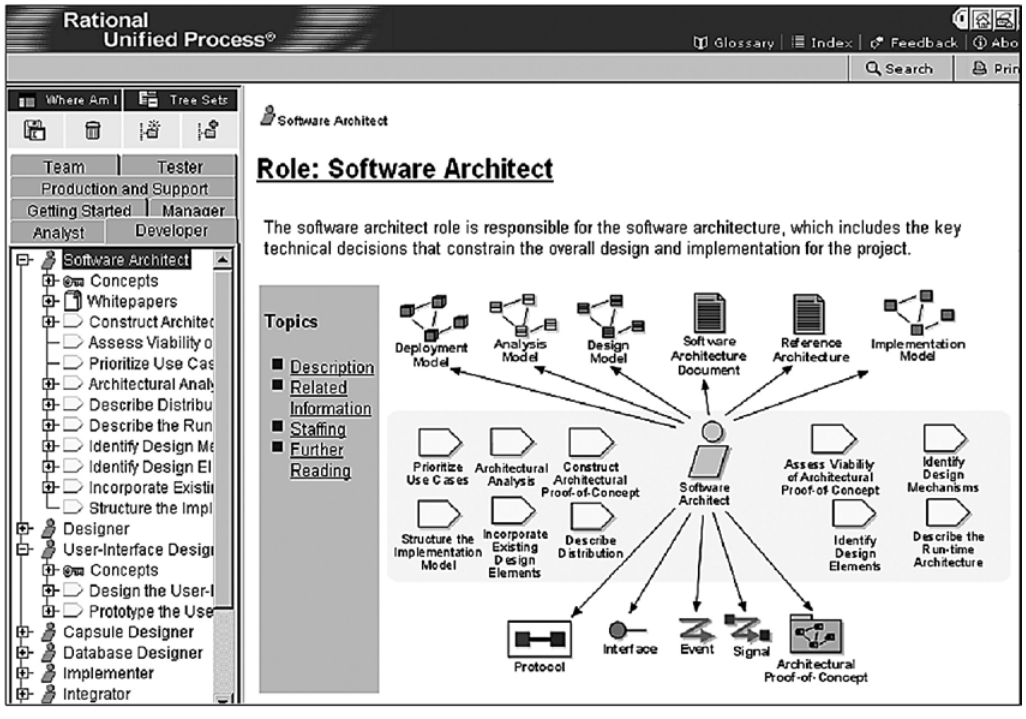

Рис. 4.4. **Рабочее окно *Rational Unified Process***

В основе *Rational Unified Process* лежат следующие ключевые принципы:

- процесс управляется прецедентами;
- процесс ориентирован на архитектуру;
- процесс является итеративным и инкрементным.

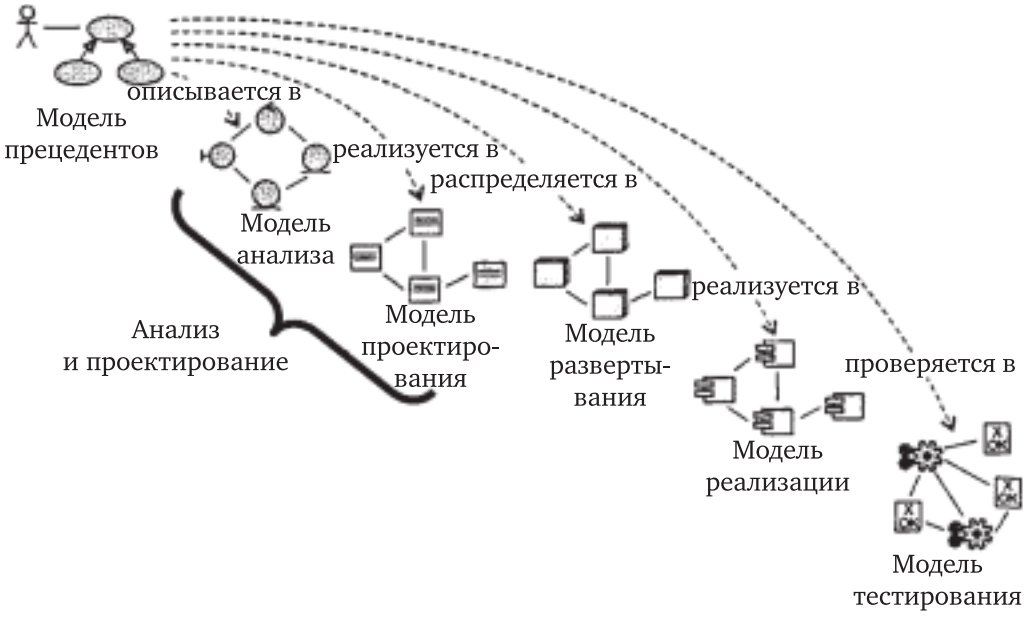

Рис. 4.5. **Схема связи между моделью прецедентов и другими моделями унифицированного процесса**

Важной особенностью *Rational Unified Process* является возможность отслеживания взаимодействия и связей элементов моделей. Например, прецеденты модели связываются с реализациями прецедентов и диаграммами последовательностей, которые, в свою очередь, ассоциируются с классами и их взаимосвязями, реализующими сценарии. Возможность отслеживания позволяет менеджерам проекта в процессе разработки перемещаться по моделям и находить требуемые ответы и решения.

На рис. 4.5 показана схема связи между моделью прецедентов и другими моделями унифицированного процесса.

4.3.2. Методология *Rational Unified Process*

Методология *Rational Unified Process* определяет инструментальную поддержку всех этапов жизненного цикла разработки ПО. Она опирается на проверенные практикой методы анализа, проектирования и разработки ПО, методы управления проектами, обеспечивает прозрачность и управляемость процесса.

Сущность *Rational Unified Process* можно интерпретировать тремя определениями:

- RUP — это четко определенная технология программной инженерии, обеспечивающая определенную структуру жизненного цикла проекта.

• RUP — это набор правил, охватывающих технологические и организационные аспекты процесса разработки ПО и регламентирующих приемы анализа и проектирования. Основной упор в *RUP* делается на моделирование разрабатываемой системы.

• RUP — это технологический программный продукт, содержащий практические методы с инструментом конфигурирования для выбора из них наиболее подходящих.

На рис. 4.6 изображена схема технологических этапов разработки программного обеспечения в *Rational Unified Process*.

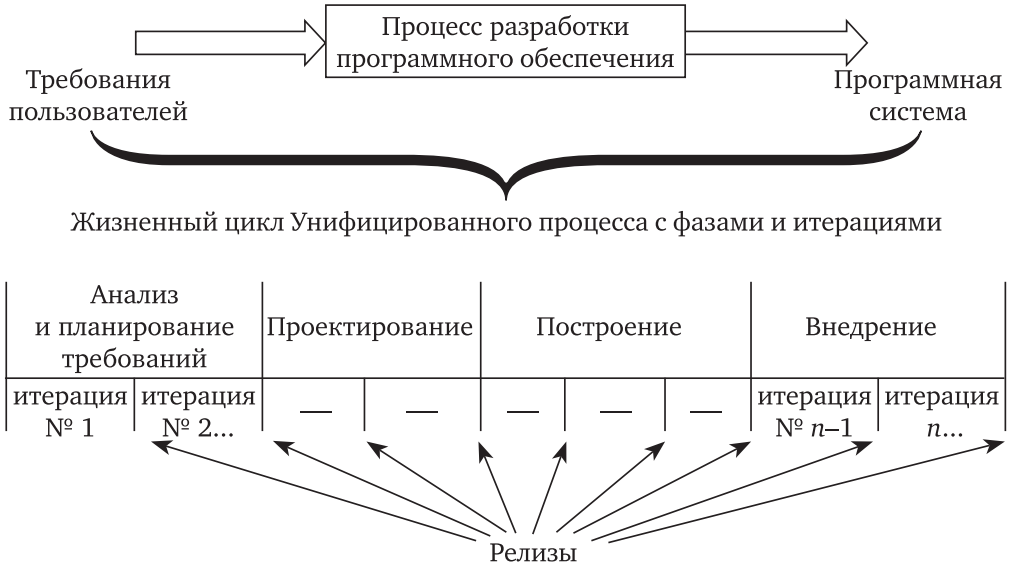

Рис. 4.6. **Унифицированный процесс разработки программного обеспечения (*Rational Unified Process*)**

Rational Unified Process компонентно-ориентирован, т. е. создаваемая программная система строится на основе программных компонентов, связанных интерфейсами как наборами операций, используемых для описания сервиса компонента. При этом неотъемлемой частью RUP является *Unified Modeling Language* — они и разрабатывались совместно. Основой создания RUP стали подходы по объектно-ориентированной разработке.

При внедрении RUP в организации, перед тем, как можно будет приступить к проекту по разработке программных систем, необходима определенная подготовка. Целесообразно определить последовательность задач по внедрению для того,

чтобы RUP стал процессом разработки программного обеспечения.

На рис. 4.7 показана схема возможной последовательности действий при внедрении RUP в масштабах всей организации.

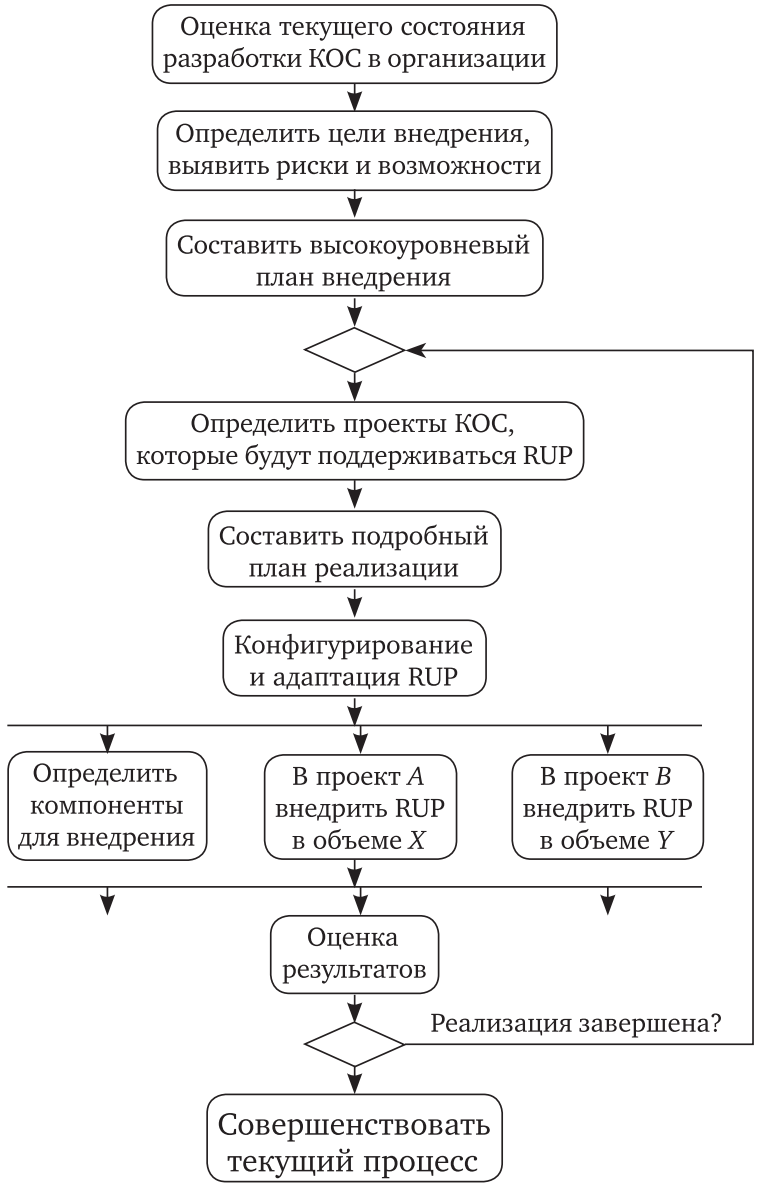

Рис. 4.7. **Дорожная карта реализации *Rational Unified Process***

При использовании *Rational Unified Process* разработка программных продуктов должна осуществляться через моделирование требований, адекватно которым будет сгенерирован код, — при этом проект получит программное воплощение, что и является конечной целью разработки.

Библиографический список

1. *Басс, Л.* Архитектура программного обеспечения на практике / Л. Басс, П. Клементс, Р. Кацман ; пер. с англ. — СПб. : Питер, 2006. — 575 с.
2. *Боггс, У.* UML и Rational Rose 2002 / У. Боггс, М. Боггс ; пер. с англ. — М. : Изд. «Лори», 2004. — 510 с.
3. *Буч, Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / Г. Буч ; пер. с англ. — М. : Бином ; СПб. : Невский диалект, 1999. — 320 с.
4. *Вендров, А. М.* Проектирование программного обеспечения экономических информационных систем / А. М. Вендров. — М. : Финансы и статистика, 2005. — 544 с.
5. *Грэхем, И.* Объектно-ориентированные методы. Принципы и практика / И. Грэхем ; пер. с англ. — М. : Издательский дом «Вильямс», 2004.— 880 с.
6. *Кватрани, Т.* Визуальное моделирование с помощью Rational Rose 2002 и UML / Т. Кватрани ; пер. с англ. — М. : Издательский дом «Вильямс», 2003. — 192 с.
7. *Кролл, П.* Rational Unified Process — это легко. Руководство по RUP / П. Кролл, Ф. Крачтен ; пер. с англ. — М. : КУДИЦ-ОБРАЗ, 2004. — 432 с.
8. *Мацяшек, Л.* Анализ требований и проектирование систем. Разработка информационных систем с использованием UML / Л. Мацяшек ; пер. с англ. — М. : Издательский дом «Вильямс», 2002. — 432 с.
9. *Рамбо, Дж.* UML: специальный справочник / Дж. Рамбо ; пер. с англ. — СПб. : Питер, 2002. — 656 с.
10. *Соммервилл, И.* Инженерия программного обеспечения / И. Соммервилл ; пер. с англ. — М. : Изд. дом «Вильямс», 2002. — 624 с.

Часть II
ПРАКТИКУМ ПО ВИЗУАЛЬНОМУ МОДЕЛИРОВАНИЮ ПРОГРАММНЫХ СИСТЕМ

Глава 1
ВИЗУАЛЬНОЕ МОДЕЛИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1. Унифицированный язык моделирования UML (*Unified Modeling Language*)

Для анализа и проектирования программных систем используют язык моделирования, включающий процедуры для решения вопросов моделирования предметной области и требований.

Таким языком является UML (*Unified Modeling Language* — унифицированный язык моделирования), разработанный компанией *Rational Software Corporation* для унификации лучших свойств, которыми обладали более ранние методы нотации.

В 1997 г. рабочей группой по развитию стандартов объектного программирования (*Object Management Group* — OMG) UML был принят в качестве стандартного языка моделирования. Он получил дальнейшее развитие и широкое признание в отрасли информационных технологий, в том числе и в качестве основной платформы для ускоренной разработки программных приложений.

Состояние индустрии объектно-ориентированного проектирования в первой половине 1990-х гг. многие аналитики характеризуют как «войну методов» или «войну методологий».

Даже на пороге слияния наиболее мощных и перспективных методов — OMT (*Object Modeling Technique*), метода Буча и OOSE (*Object-Oriented Software Engineering*) — не были решены проблемы, порождаемые уникальными системами обозначения в каждом методе. Нотации обеспечивали «видение» системы

в процессе ее создания, что крайне важно для разработчиков, но объединение методов требовало введения единого языка моделирования.

Конец «войне методов» положило принятие стандарта *Unified Modeling Language* — унифицированного языка моделирования UML, применяемого для определения, визуализации и документирования сущностей разрабатываемой объектно-ориентированной системы [4].

Составляющие UML

Язык моделирования UML унифицирует системы обозначений, принятые в методе Буча, OMT и OOSE, сочетает в себе все лучшее из результатов, полученных авторами других методологий, и дополняет их новыми возможностями.

При разработке UML были поставлены и реализованы следующие цели:

- предоставить пользователям готовый выразительный язык визуального моделирования, позволяющий разрабатывать осмысленные модели и обмениваться ими;
- предусмотреть механизмы и специализации для расширения базовых концепций;
- обеспечить независимость от конкретных языков программирования и процессов разработки;
- обеспечить формальную основу для понимания этого языка моделирования (язык должен быть одновременно точным и доступным для понимания, без лишнего формализма);
- стимулировать рост рынка объектно-ориентированных инструментальных средств;
- интегрировать лучший практический опыт.

UML представляет собой успешную попытку стандартизации составляющих процесса анализа и проектирования — семантических моделей, синтаксической нотации и диаграмм.

Первая версия стандарта (0.8) появилась в 1995 г., а в ноябре 1997 г. организация OMG одобрила и утвердила спецификацию UML версии 1.1 в качестве стандарта языка моделирования. С тех пор UML получил дальнейшее развитие и широкое признание в отрасли информационных технологий. В версию 1.4 было введено множество детализированных понятий о компонентах и профилях, а в версию 1.5 добавили семантику действия. В 2005 г. была стандартизована версия UML 2.0.

Язык UML не зависит от применяемого процесса разработки программного обеспечения, хотя позже компания *Rational* предложила процесс, соответствующий этому языку, под названием *Rational Unified Process* (Унифицированный процесс *Rational*) [5]. Заметим, что название методологии можно перевести как «рациональный унифицированный процесс».

Элементы нотации *Unified Modeling Language*

На рис. 1.1—1.6 показаны нотации для основных диаграмм UML, а именно: прецедентов, типового решения «модель — представление — управление», последовательности, кооперации, деятельности, классов.

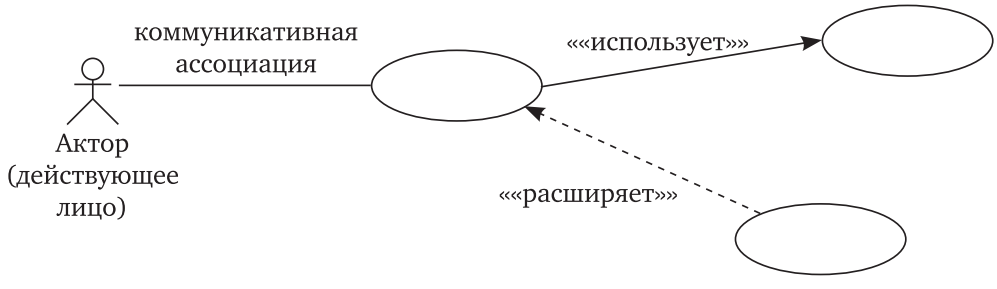

Рис. 1.1. **Нотация диаграммы прецедентов (вариантов использования) системы**

Рис. 1.2. **Нотация стереотипов типового решения «модель — представление — управление» диаграммы прецедентов**

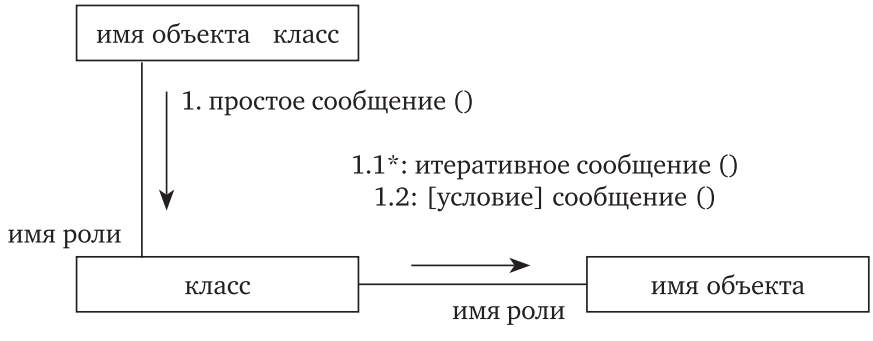

Рис. 1.3. **Нотация диаграммы кооперации**

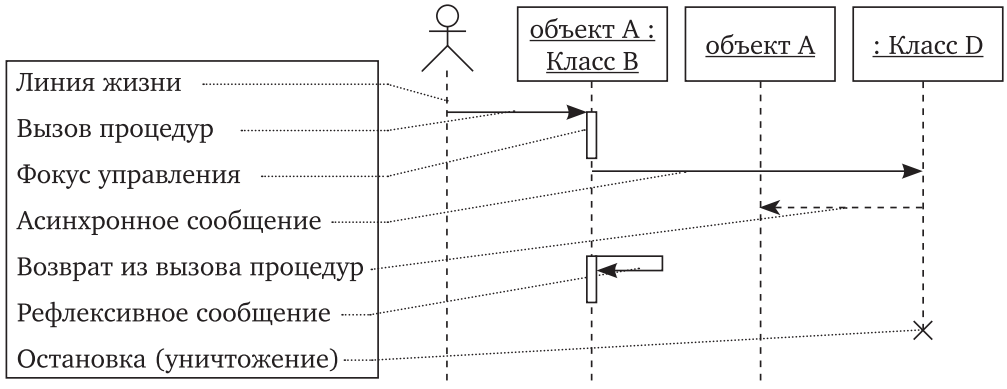

Рис. 1.4. **Нотация диаграммы последовательности**

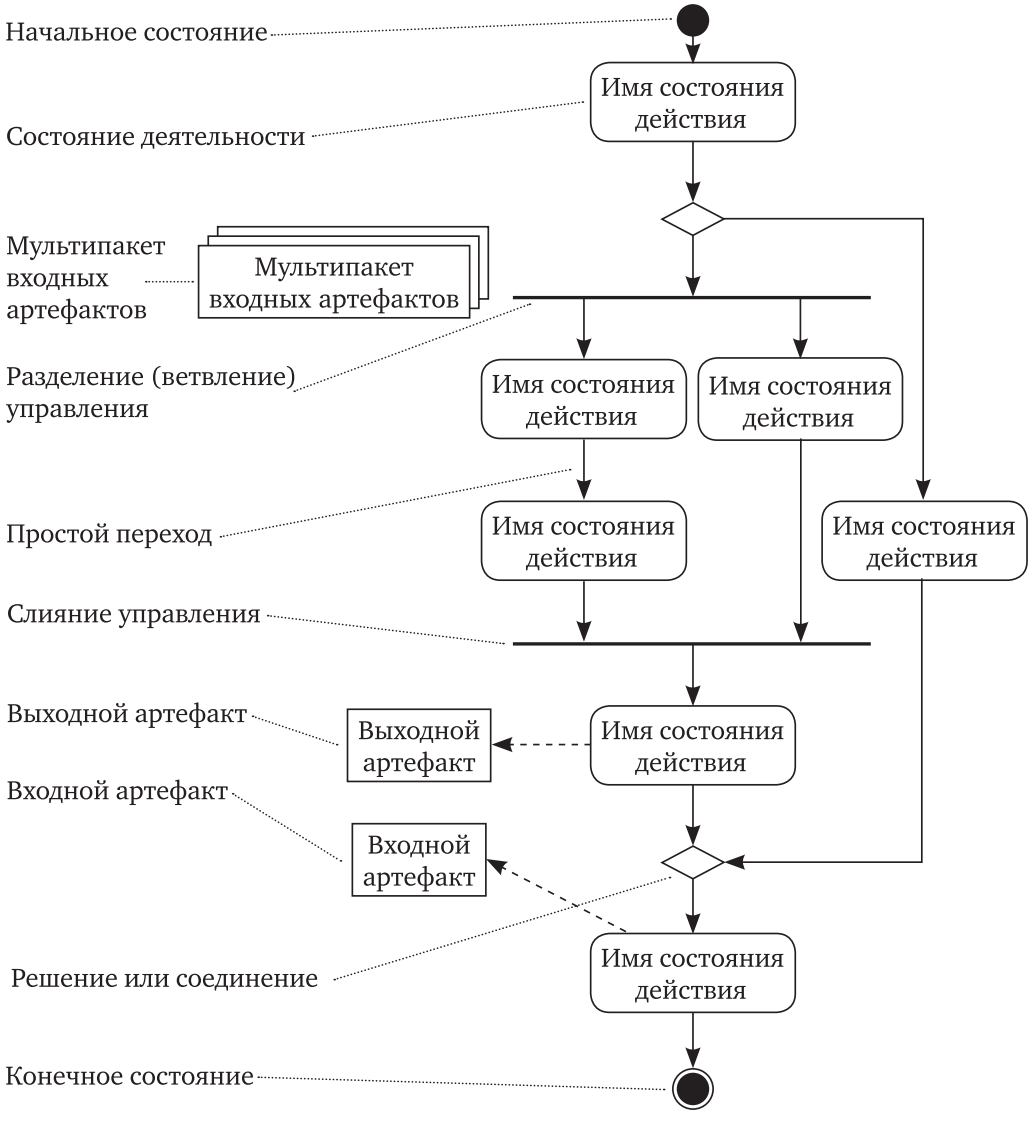

Рис. 1.5. **Нотация диаграммы деятельности**

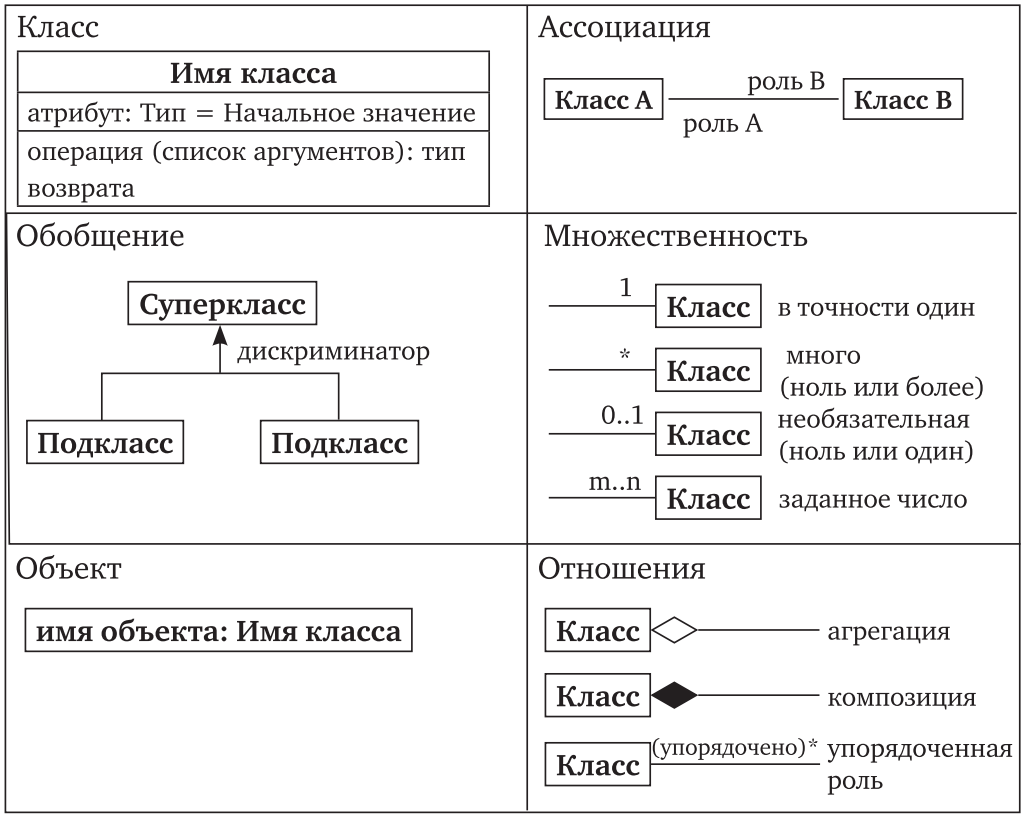

Рис. 1.6. **Нотация диаграммы классов**

Направления унификации UML

Понятие «унифицированный» в применении к UML в трактовке его создателей означает следующее.

- *Унификация старых методов и нотаций*. В языке UML собраны и четко определены общепризнанные концепции из многих объектно-ориентированных методов, поэтому он может представлять большинство существующих моделей на том же уровне или даже лучше, чем это делалось в исходных методах.
- *Унификация по этапам разработки*. UML может успешно использоваться на всех этапах разработки программного обеспечения — от определения требований до развертывания программного обеспечения. Один и тот же набор концепций и нотаций можно использовать на разных стадиях разработки и даже смешивать в одной модели. Эта особенность языка моделирования важна для итеративной, инкрементной разработки.
- *Унификация по предметным областям*. UML предназначен для моделирования в любых предметных областях, включая

моделирование больших и сложных систем, систем реального времени, распределенных систем и систем с интенсивными вычислениями или обработкой данных.

- *Унификация по платформам и языкам реализации*. UML может использоваться для систем, реализованных на разных языках и платформах, включая языки программирования, базы данных, языки четвертого поколения (4GL), организационные документы, встроенное программное обеспечение и т. д. При этом внешние интерфейсы будут во всех случаях идентичными или сходными, а работа внутренней, серверной части для различных сред будет несколько различаться.
- *Унификация по процессам разработки*. UML — это язык моделирования, а не подробное описание процесса разработки. Он построен так, чтобы быть пригодным для использования в большинстве существующих и новых процессов разработки — подобно тому, как язык программирования общего назначения пригоден для программирования в различных стилях. Главным образом UML ориентирован на итеративный, инкрементный стиль разработки, который является приоритетным в объектно-ориентированном подходе.

Символы UML напоминают нотации Буча и OMT, но содержат также элементы из других нотаций.

Область применения UML

Очевидно, что процесс, в котором в качестве базового языка принят UML, должен поддерживать *объектно-ориентированный подх*од к созданию программного обеспечения. Язык UML не подходит для несовременных структурных подходов, результатом которых являются системы, реализованные с помощью процедурных языков программирования, наподобие языка COBOL.

Язык UML не зависит от технологий реализации (поскольку они являются объектно-ориентированными). Некоторые эксперты считают, что это делает UML ограниченным при поддержке этапа детализированного проектирования жизненного цикла программного обеспечения. В то же время UML становится более устойчивым к частой смене платформ реализации. Разработчики языка отмечают следующие ключевые особенности процессов, эффективно работающих в комбинации с UML:

- управляемые прецедентами;
- ориентированные на архитектуру;
- итеративные и инкрементные.

Унифицированный язык моделирования UML позволяет создать своеобразный чертеж, подробно описывающий архитектуру системы, поскольку за каждым графическим символом стоит хорошо определенная семантика. С помощью такого описания (или модели) упрощается разработка и обновление программной системы и повышается адекватность реализации всех технических требований к приложениям.

В настоящее время UML используется практически всеми крупнейшими компаниями — производителями программного обеспечения (*Microsoft*, *IBM*, *Hewlett-Packard*, *Oracle*, *Sybase* и др.). Кроме того, все значительные мировые производители CASE-средств, помимо *Rational Software*, используют UML в своих разработках:

— *Paradigm Plus* 3.6;
— *System Architec*;
— *Microsoft Visual Modeller for Visual Basic*;
— *Delphi*;
— *PowerBuilder* и др.

UML не является языком программирования. С его помощью можно писать программы, но в нем отсутствуют свойственные большинству языков программирования синтаксические и семантические соглашения, облегчающие работу программиста.

С точки зрения большинства программистов, размышления по поводу реализации проекта почти эквивалентны написанию для него кода. Действительно, некоторые вещи лучше всего выражаются непосредственно в коде на каком-либо языке программирования, поскольку текст программы — это самый простой и короткий путь для записи алгоритмов и выражений. Но даже в таких случаях программист занимается моделированием, хотя и неформально.

С помощью инструментальных средств, поддерживающих UML и содержащих генераторы кода, на основе созданной модели можно получить программный код на различных языках, и, наоборот, по исходному коду уже существующих программ можно восстановить их UML-модели.

UML — язык дискретного моделирования. Он не предназначен для разработки непрерывных систем, встречающихся в физике и механике. UML создавался как язык моделирования общего назначения для применения в таких дискретных систе-

мах, как программное обеспечение, аппаратные средства или цифровая логика.

В более специфических случаях, например для проектирования схем со сверхвысоким уровнем интеграции или создания систем, обладающих искусственным интеллектом, целесообразно использовать специализированные инструменты и языки моделирования.

Визуализация, специфицирование, конструирование и документирование артефактов программных систем — это и есть назначение языка UML.

Концептуальные области UML

Все концепции и модели языка UML можно отнести к четырем концептуальным областям.

1. Статическая структура.

В первую очередь, любая точная модель должна определять полное множество объектов, т. е. ключевые концепции приложения, их внутренние свойства и отношения между собой. Эта структура и есть *статическое представление системы*.

Концепции приложения моделируются как классы, каждый из которых описывает тип дискретных объектов, содержащих определенную информацию и взаимодействующих между собой для реализации некоторого поведения.

Информация, которую содержат объекты, моделируется как атрибуты, поведение — как операции. Используя механизм обобщения, несколько классов могут иметь одну общую структуру. Класс-потомок содержит в себе структуру и поведение, унаследованные от общего класса-предка, а также собственную, уникальную для него структуру и поведение.

В процессе работы одни объекты имеют связи с другими. Такие отношения между объектами моделируются как ассоциации между соответствующими классами. Другие отношения между элементами, включая отношения между уровнями абстракции, связывание фактических значений с формальными параметрами шаблона, наделение правами и использование одного элемента другим, группируются как отношения зависимости.

Классы могут иметь интерфейсы, которые описывают поведение класса, видимое извне. Статическое представление системы изображается с помощью диаграмм классов или их вариаций.

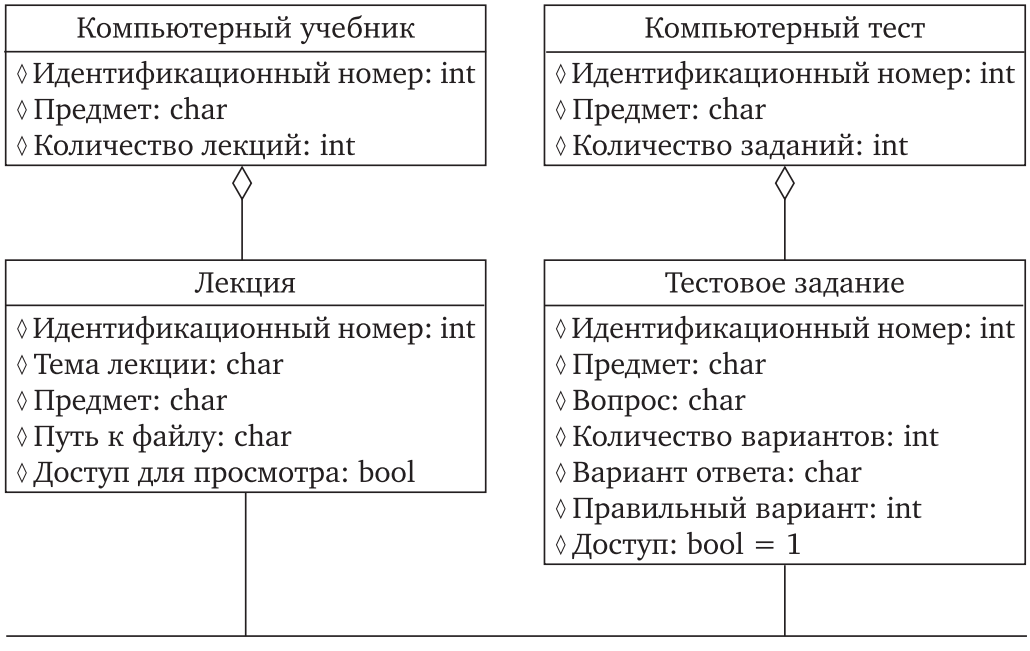

Рис. 1.7. **Фрагмент диаграммы классов сущностей с атрибутами для компьютерной обучающей системы**

Модели UML создаются как для логического анализа, так и для проектирования, обеспечивающего реализацию системы. Некоторые конструкции в модели представляют собой проектные элементы.

Структурированный классификатор раскрывает реализацию класса в виде набора частей, скрепленных между собой соединителями. Класс может инкапсулировать свою внутреннюю структуру за внешне видимыми портами. Кооперация моделирует набор объектов, играющих определенные роли в меняющемся контексте. Компонент — это замещаемая часть системы, которая соответствует некоторому набору интерфейсов и обеспечивает их реализацию. Компонент должен легко замещаться другими компонентами с тем же набором интерфейсов.

Элементами развертывания являются:

- узел — это вычислительный ресурс периода прогона, который определяет местонахождение исполняемых компонентов и объектов;
- артефакт — это физическая единица информации или описания поведения вычислительной системы. Артефакты развертываются в узлах и могут быть реализацией компонента;

• представление развертывания системы — описывает конфигурацию узлов рабочей системы и расположение артефактов в этих узлах.

2. *Динамическая структура.*

Известны три способа для моделирования поведения. Первый представляет собой историю жизни объекта и его взаимодействия с остальным миром. Второй описывает паттерны (образцы) взаимодействия для набора соединенных объектов при реализации определенного поведения. Третий способ — это описание эволюции процесса исполнения программы в ходе осуществления ею разнообразной деятельности.

Взаимодействие отражает структурированный классификатор или кооперацию с потоком сообщений между частями. Взаимодействия показывают на диаграммах последовательности и диаграммах коммуникации. Диаграммы последовательности выделяют упорядочение сообщений по времени, а диаграммы кооперации — связи объектов.

На рис. 1.8 показан пример диаграммы последовательности для прецедента «Генерация вопросов для самоконтроля» модуля генерации учебно-тренировочных заданий, а на рис. 1.9 — диаграмма кооперации для этого прецедента.

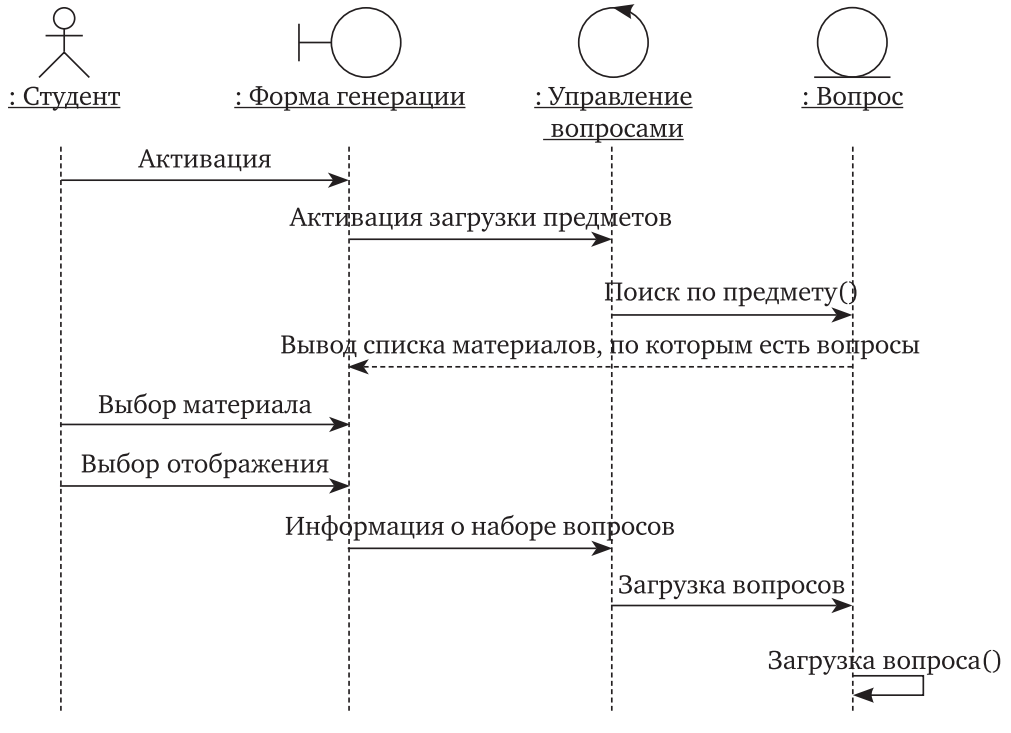

Рис. 1.8. **Диаграмма последовательности для прецедента «Генерация вопросов для самоконтроля»**

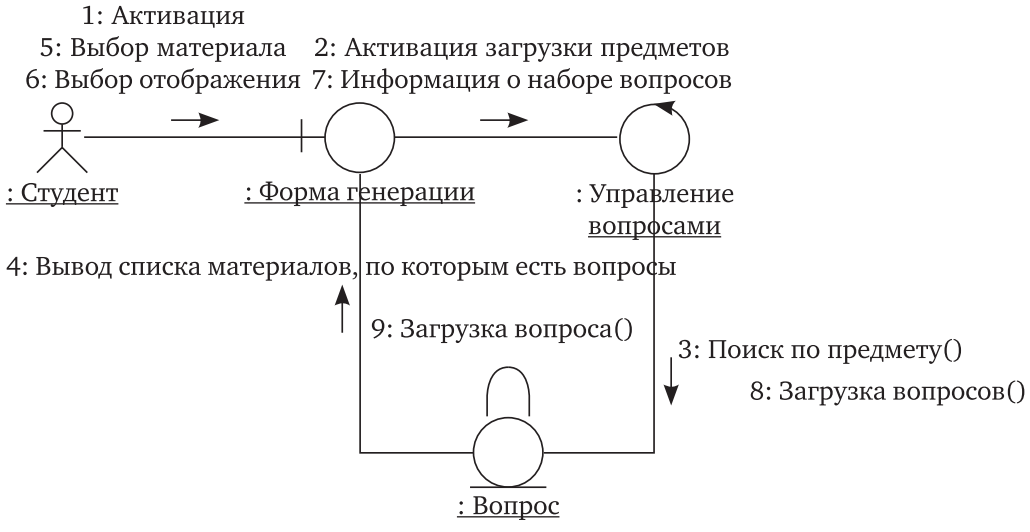

Рис. 1.9. **Диаграмма кооперации для прецедента «Генерация вопросов для самоконтроля»**

Деятельность описывает процесс выполнения вычислений. Она моделируется как набор узлов деятельности, связанных потоками управления и потоками данных.

Деятельность может моделировать как последовательное, так и параллельное поведение. Она включает в себя традиционные конструкции потока управления — разветвления и циклы. На диаграммах деятельности можно изображать ход вычислений, а также рабочие потоки в организациях.

3. *Организация модели.*

При разработке больших систем вся информация о модели должна быть поделена на связные понятные части, чтобы команды разработчиков могли параллельно работать над различными разделами системы. Даже при разработке небольших систем лучше разделить содержимое модели на несколько пакетов приемлемого размера.

Пакетами в языке UML называются иерархически организованные блоки моделей. Их можно использовать для хранения, контроля доступа, управления конфигурацией и конструирования библиотек, содержащих фрагменты моделей многократного пользования. Зависимости между пакетами определяются совокупностью зависимостей между их содержимым и могут также задаваться системной архитектурой. Таким образом, содержимое пакетов должно соответствовать зависимостям между пакетами и заданной архитектуре системы.

4. *Профили.*

Как бы велики ни были возможности языка, появится необходимость их расширить. UML обладает способностью к расширению без изменений основных положений языка.

Как и к расширению возможностей любого другого языка, к расширению UML нужно подходить очень осторожно, так как можно легко создать новый диалект языка, непонятный для остальных.

Концептуальная модель UML

Концептуальная модель UML включает в себя три составные части:

- основные строительные блоки языка;
- правила сочетания;
- общие для всего языка механизмы.

Основными объектно-ориентированными блоками UML являются сущности, с помощью которых можно создавать корректные модели.

Структурные сущности — это имена существительные в моделях на языке UML. Они представляют собой статические части модели, соответствующие концептуальным или физическим элементам системы.

Существует семь базовых разновидностей структурных сущностей: классы, интерфейсы, кооперации, прецеденты, активные классы, компоненты и узлы. Связующими строительными блоками в UML являются отношения, которые применяются для создания корректных моделей. UML характеризуется набором правил, определяющих, как должна выглядеть хорошо оформленная модель, т. е. семантически согласованная и находящаяся в гармонии со всеми моделями, которые с нею связаны.

Конструкции языка UML позволяют моделировать статику (структуру) и динамику (поведение) системы. Систему представляют в виде взаимодействующих *объектов* (программных модулей), которые реагируют на внешние события. Отдельные модели отображают определенные стороны системы и пренебрегают другими сторонами, которые охватывают другие модели. Взятые в комплексе модели обеспечивают полное описание системы.

Как уже отмечалось, наибольшего и разностороннего понимания системы можно достичь при моделировании с различных, но взаимосвязанных точек зрения.

1.2. Визуальные модели и диаграммы программных систем

Моделирование используют при изучении всех инженерных дисциплин, в значительной степени из-за того, что оно реализует принципы декомпозиции, абстракции и иерархии. Общим свойством всех моделей является их подобие оригинальной системе или системе-оригиналу. Процесс построения и последующего применения моделей для получения информации о свойствах или поведении системы-оригинала называется моделированием.

Визуальным моделированием (*visual modeling*) называется процесс графического представления модели с помощью некоторого стандартного набора графических элементов. Общение между пользователями, разработчиками, аналитиками, тестировщиками, менеджерами и всеми остальными участниками проекта является основной целью визуального моделирования. Общение можно обеспечить и с помощью невизуальной (текстовой) информации, но сложная информация понимается легче, если она представлена визуально, а не описана в тексте.

В рамках UML-модели все представления о системе фиксируются в виде графических конструкций — диаграмм. С помощью диаграмм можно визуализировать систему с различных точек зрения. Например, можно описывать взаимодействие пользователя с системой, изменение состояний системы в процессе ее работы и т. д. Сложную систему можно представить в виде набора небольших моделей-диаграмм, причем ни одна из них не является достаточной для получения полного представления о системе. Это следует из того, что каждая из диаграмм отражает определенный аспект функционирования системы на разных уровнях абстракции.

Таким образом, следует подчеркнуть, что ни одна отдельная диаграмма не является моделью. Диаграммы — средство визуализации модели, и эти два понятия следует различать. Лишь набор диаграмм составляет модель системы и наиболее полно ее описывает.

В унифицированном языке моделирования UML определены три группы диаграмм:

- представляющие статическую структуру приложения;
- представляющие поведенческие аспекты системы;
- представляющие физические аспекты функционирования системы (диаграммы реализации).

Важно понимать, что количество диаграмм для конкретного приложения не является строго фиксированным. Для простых приложений нет необходимости строить все без исключения диаграммы. Перечень диаграмм зависит от специфики разрабатываемого проекта и определяется самим разработчиком.

1.3. Виды диаграмм

Рассмотрим некоторые типы диаграмм для программной системы банкомата (*Automated Teller Machine* — ATM):

- вариантов использования;
- последовательности;
- кооперации;
- классов;
- состояний;
- размещения.

Представим эти диаграммы для программной системы банкомата, чтобы описать систему с различных точек зрения.

Диаграммы вариантов использования[1]

Диаграммы вариантов использования отображают общие особенности функционирования моделируемой системы без рассмотрения ее внутренней структуры. Диаграммы вариантов использования представляют также действующих лиц (actor), инициирующих варианты использования системы.

На рис. 1.10 изображена диаграмма вариантов использования для банкомата.

На диаграмме показано взаимодействие между вариантами использования и действующими лицами для ATM. Она отражает требования к системе с точки зрения пользователя. В данном примере клиент банка инициирует различные варианты использования банкомата:

- Снять деньги со счета.
- Перевести деньги.
- Положить деньги на счет.
- Показать баланс.
- Изменить идентификационный номер.
- Произвести оплату.

---

[1] Поскольку русскоязычная терминология языка UML не унифицирована, в литературе можно встретить различные переводы названия диаграммы use case: диаграмма *вариантов использования*, диаграмма *прецедентов*, а также просто транслитерация *юзкейс*.

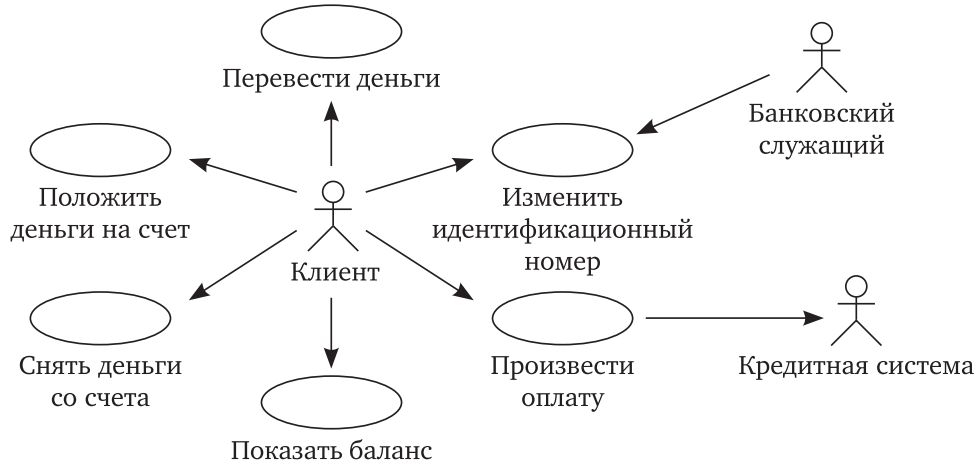

Рис. 1.10. **Диаграмма вариантов использования для банкомата**

От варианта использования «Произвести оплату» идет стрелка к Кредитной системе. Действующими лицами могут быть и внешние системы, в данном случае Кредитная система показана именно как действующее лицо — она является внешней для системы банкомата. Стрелка, направленная от прецедента к действующему лицу, показывает, что прецедент предоставляет некоторую информацию действующему лицу. В данном случае прецедент «Произвести оплату» предоставляет Кредитной системе информацию об оплате по кредитной карточке.

Из диаграмм вариантов использования можно получить довольно много информации о системе, поскольку в них описывается общая функциональность системы. Пользователи, менеджеры проектов, аналитики, разработчики, специалисты по контролю качества могут, изучая диаграммы вариантов использования, понять, что система должна делать.

Диаграммы последовательности

Диаграммы последовательности относятся к диаграммам взаимодействия UML, описывающим поведенческие аспекты системы и отражающим поток событий, происходящих в рамках конкретного варианта использования.

На рис. 1.11 показана диаграмма последовательности для варианта использования «Снять деньги», который предусматривает несколько возможных последовательностей: снятие денег; попытка снять деньги при отсутствии их достаточного количества на счету; попытка снять деньги по неправильному идентификационному номеру и некоторые другие.

Нормальный сценарий снятия некоторой суммы со счета (при отсутствии таких проблем, как неправильный идентификационный номер или недостаток денег на счете) показан на рис 1.11.

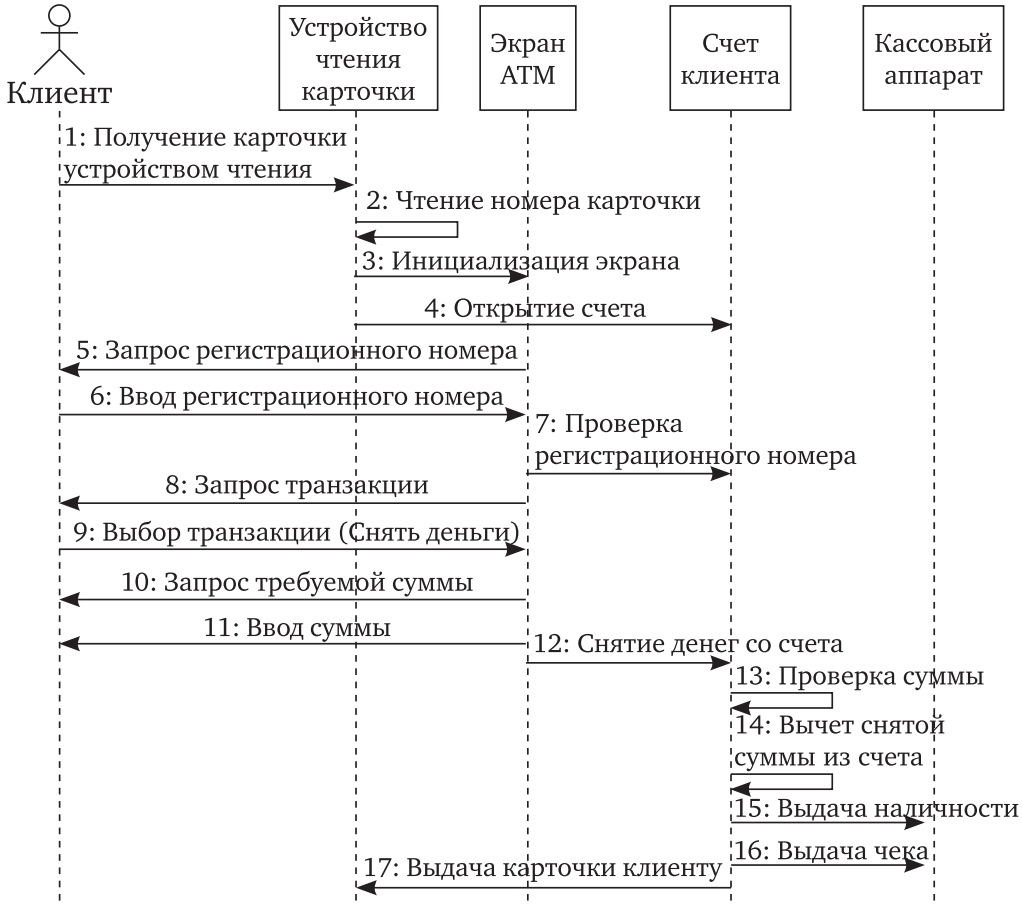

Рис. 1.11. **Диаграмма последовательности для прецедента «Снять деньги»**

Диаграмма последовательности отображает поток событий варианта использования «Снять деньги». В верхней части диаграммы показаны все действующие лица и объекты, необходимые системе для выполнения этого варианта использования. Стрелки соответствуют сообщениям, которыми обмениваются действующее лицо или объекты между собой для выполнения требуемых функций.

По данной диаграмме аналитики видят последовательность (поток) действий, разработчики — объекты, которые надо создать, и их операции. Специалисты по контролю качества поймут детали процесса и смогут разработать тесты для

их проверки. Таким образом, диаграммы последовательности полезны всем участникам проекта.

Совокупность диаграмм последовательности для всех вариантов использования системы является ядром динамической модели, в котором подробно определяется поведение системы во время выполнения действий и то, как реализуется это поведение.

Диаграммы кооперации

Диаграммы кооперации получаются в *Rational Rose* автоматическим преобразованием из диаграмм последовательности, позволяя подробно проанализировать взаимодействие объектов системы при описании каждого прецедента.

На рис. 1.12 показана диаграмма кооперации для прецедента «Снять деньги». Если диаграмма последовательности показывает взаимодействие между действующими лицами и объектами во времени, то на диаграмме кооперации связь со временем отсутствует.

Рис. 1.12. **Диаграмма кооперации для прецедента «Снять деньги»**

Анализ диаграмм кооперации помогает выявить «узкие» места (избыточную загруженность) динамической модели проектируемой системы. Например, если диаграмма кооперации построена в виде звезды, где несколько объектов связаны с одним центральным, то архитектор системы может сделать вывод, что система находится в слишком тесной зависимости от центрального объекта. Из этого следует, что целесообразно перепроектировать систему так, чтобы процессы были распределены более равномерно. На диаграмме последовательности такой тип взаимодействия было бы трудно увидеть.

Проверка следования определенным правилам взаимодействия объектов системы друг с другом дает возможность также проверить наличие ошибок в диаграммах последовательности.

Диаграммы классов

Диаграммы классов отражают взаимодействие классов системы. Классы системы можно рассматривать как типы объектов. Например, счет клиента — это объект. Типом такого объекта можно считать счет вообще, т. е. «Счет» (*Account*) — это класс. Классы содержат данные и поведение (действия), влияющие на эти данные. Класс *Account* содержит идентификационный номер клиента и проверяющие его действия. На диаграмме класс создается для каждого типа объектов из диаграммы последовательности или диаграммы кооперации.

Диаграмма классов для варианта использования «Снять деньги» показана на рис. 1.13.

На диаграмме показаны связи между классами, реализующими вариант использования «Снять деньги». В этом процессе задействованы четыре класса: *Card Reader* (устройство для чтения карточек), *Account* (счет), ATM *Screen* (экран банкомата) и *Cash Dispenser* (кассовый аппарат).

Каждый класс на диаграмме классов изображается в виде прямоугольника, разделенного на три части. В первой части указывается имя класса, во второй — его *атрибуты*. Атрибут — это некоторая информация, характеризующая класс. Например, класс *Account* (счет) имеет три атрибута: *Account Number* (номер счета), *PIN* (идентификационный номер) и *Balance* (баланс). В последней части содержатся *операции* класса, отражающие его поведение (действия, выполняемые классом). Для класса *Account* определены четыре операции: *Open* (открыть), *Withdraw Funds* (снять деньги), *Deduct Funds* (вычесть сумму денег из счета) и *Verify Funds* (проверить наличие денег).

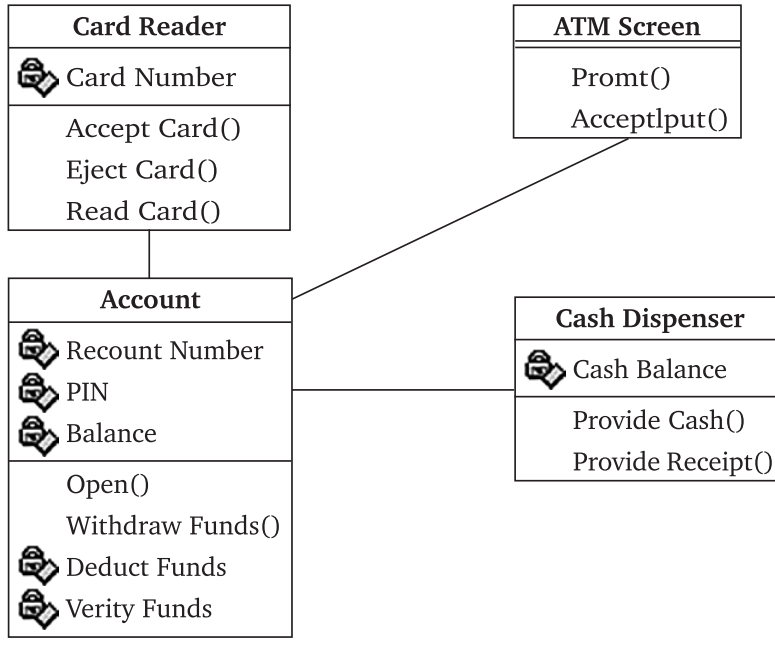

Рис. 1.13. **Диаграмма классов для варианта использования «Снять деньги»**

Линии, связывающие классы, показывают взаимодействие между классами. Класс *Account* связан с классом ATM *Screen*, потому что они непосредственно взаимодействуют друг с другом. Класс *Card Reader* не связан с классом *Cash Dispenser*, поскольку они не общаются друг с другом непосредственно.

Обратите внимание, что слева от некоторых атрибутов и операций помещены маленькие значки в виде висячего замка . Они указывают на то, что атрибут или операция класса закрыты (*private*). Доступ к закрытым атрибутам или операциям возможен только из содержащего их класса. *Account Number, PIN* и *Balance* — закрытые атрибуты класса *Account*. Кроме того, закрытыми являются операции *Deduct Funds* и *Verify Funds*.

Разработчики используют диаграммы классов для реального создания классов. Такие инструменты, как *Rational Rose*, генерируют основу кода классов, которую программисты заполняют деталями на выбранном ими языке. С помощью этих диаграмм аналитики могут показать детали системы, а архитекторы — понять ее дизайн. Если, например, какой-либо класс несет слишком большую функциональную нагрузку, это будет видно на диаграмме классов, и архитектор сможет перераспределить ее между другими классами.

С помощью диаграммы можно также выявить случаи, когда между сообщающимися классами не определено никаких связей. Диаграммы классов следует создавать, чтобы показать взаимодействующие классы в каждом варианте использования.

Диаграммы состояний

Диаграммы состояний предназначены для моделирования различных состояний, в которых может находиться объект. В то время как диаграмма классов показывает статическую картину классов и их связей, диаграммы состояний применяют при описании динамики поведении системы. Такие диаграммы широко используются в проектировании систем реального времени. Пакет *Rational Rose* способен по диаграмме состояний генерировать полный код для системы реального времени.

Диаграммы состояний отображают поведение объекта. Так, банковский счет может иметь несколько различных состояний. Он может быть открыт, закрыт или превышен кредит по нему. Поведение счета меняется в зависимости от состояния, в котором он находится. На диаграмме состояний показывают именно эту информацию. На рис. 1.14 приведен пример диаграммы состояний для банковского счета.

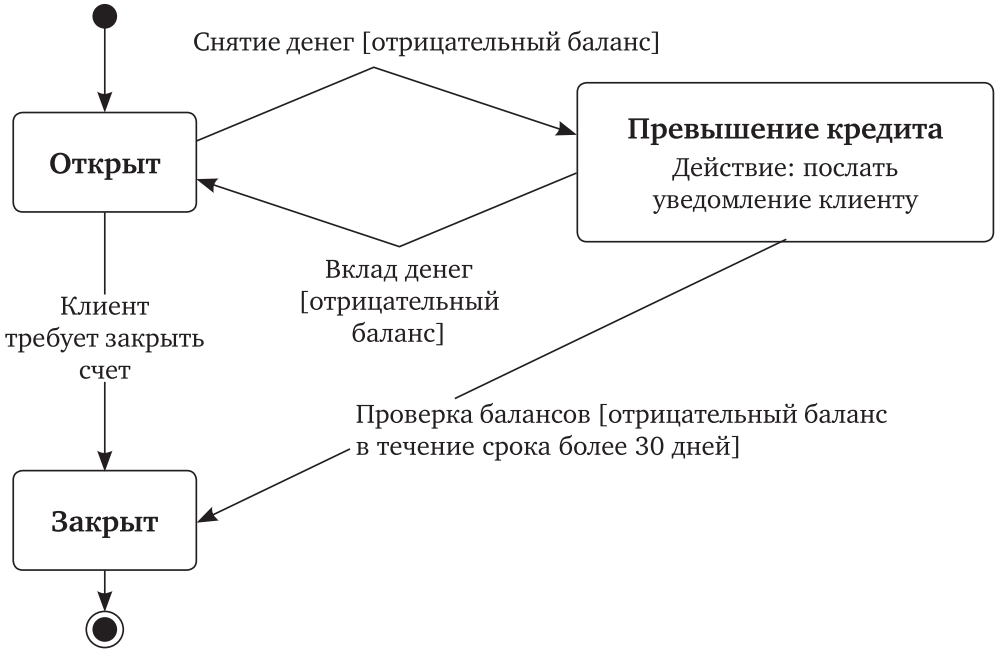

Рис. 1.14. **Диаграмма состояний для класса *Account***

На данной диаграмме показаны возможные состояния счета, а также процесс перехода счета из одного состояния в другое. Например, если клиент требует закрыть открытый счет, последний переходит в состояние «Закрыт». Требование клиента называется *событием* (*event*), и именно события вызывают переход из одного состояния в другое.

Когда клиент снимает деньги с открытого счета, счет может перейти в состояние «Превышение кредита». Это происходит в случае, если баланс по счету меньше нуля, что отражено условием [отрицательный баланс] на диаграмме. Заключенное в квадратные скобки *ограничивающее условие* (*guard condition*) определяет, когда может или не может произойти переход из одного состояния в другое.

На диаграмме имеются два специальных состояния — *начальное* (*start*) и *конечное* (*stop*). Начальное состояние выделяется черной точкой; оно соответствует состоянию объекта в момент его создания. Конечное состояние обозначается черной точкой в белом кружке; оно соответствует состоянию объекта непосредственно перед его уничтожением. На диаграмме состояний может быть одно и только одно начальное состояние. В то же время может быть столько конечных состояний, сколько вам нужно, или их может не быть вообще.

Когда объект находится в каком-то конкретном состоянии, могут выполняться те или иные процессы. В данном примере при превышении кредита клиенту посылается соответствующее сообщение. Процессы, происходящие при нахождении объекта в определенном состоянии, называются *действиями* (*action*).

Диаграммы состояний не нужно создавать для каждого класса, их применяют только в сложных случаях. Если объект класса может существовать в нескольких состояниях и в каждом из них ведет себя по-разному, то для него, вероятно, понадобится такая диаграмма. В основном они необходимы для документирования.

При генерации в среде *Rational Rose* на основе диаграммы состояния содержащаяся в ней информация не преобразуется в код. Тем не менее, для работающих в режиме реального времени систем доступны различные расширения *Rational Rose*, которые позволяют генерировать исполняемый код, основываясь на диаграммах состояния.

Диаграммы размещения

Диаграммы размещения показывают физическое расположение различных компонентов системы в сети. Для рассматриваемого примера система банкомата ATM состоит из большого количества подсистем, выполняемых на отдельных физических устройствах или узлах (*node*).

Диаграмма размещения для системы банкомата ATM показана на рис. 1.15.

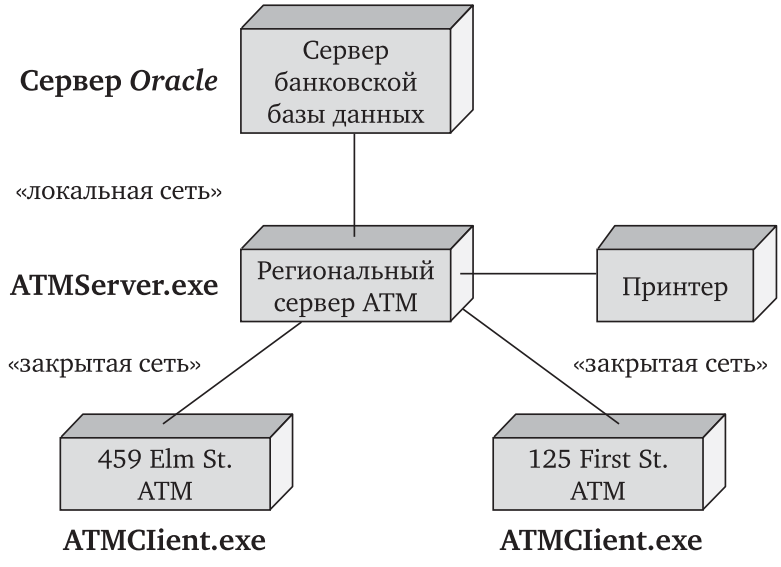

Рис. 1.15. **Диаграмма размещения для системы банкомата ATM**

Из диаграммы следует, что клиентские программы ATM будут работать в нескольких местах на различных сайтах. Через закрытые сети осуществляется сообщение клиентов с региональным сервером ATM, на котором будет работать программное обеспечение сервера. Региональный сервер взаимодействует с сервером банковской базы данных, работающим под управлением *Oracle*. С региональным сервером ATM соединен принтер. Таким образом, реализована трехуровневая архитектура.

Диаграмма размещения используется менеджером проекта, пользователями, архитектором системы и эксплуатационным персоналом для определения физического размещения системы и расположения ее отдельных подсистем.

Глава 2
ИНСТРУМЕНТАЛЬНОЕ СРЕДСТВО *IBM RATIONAL ROSE*

2.1. Элементы интерфейса *IBM Rational Rose*

Интерфейс *Rose* содержит пять основных элементов — браузер, окно документации, панели инструментов, окно диаграммы и журнал (*log*). Их назначение заключается в следующем.

- Браузер (*browser*) — используется для быстрой навигации по модели.
- Окно документации (*documentation window*) — применяется для работы с текстовым описанием элементов модели.
- Панели инструментов (*toolbars*) — применяются для быстрого доступа к наиболее распространенным командам.
- Окно диаграммы (*diagram window*) — используется для просмотра и редактирования одной или нескольких диаграмм UML.
- Журнал (*log*) — применяется для просмотра ошибок и отчетов о результатах выполнения различных команд.

На рис. 2.1 показаны основные элементы интерфейса *Rose*.

Браузер

Браузер — это иерархическая структура, позволяющая осуществлять навигацию по модели. Все, что добавляется к ней — действующие лица, варианты использования, классы, компоненты, — будет показано в окне браузера.

С помощью браузера можно:

— добавлять к модели элементы (действующие лица, варианты использования, классы, компоненты, диаграммы и т. д.);

— просматривать существующие элементы модели;

— просматривать существующие связи между элементами модели;

— перемещать элементы модели;
— переименовывать эти элементы;
— добавлять элементы модели к диаграмме;
— группировать элементы в пакеты;
— открывать диаграмму.

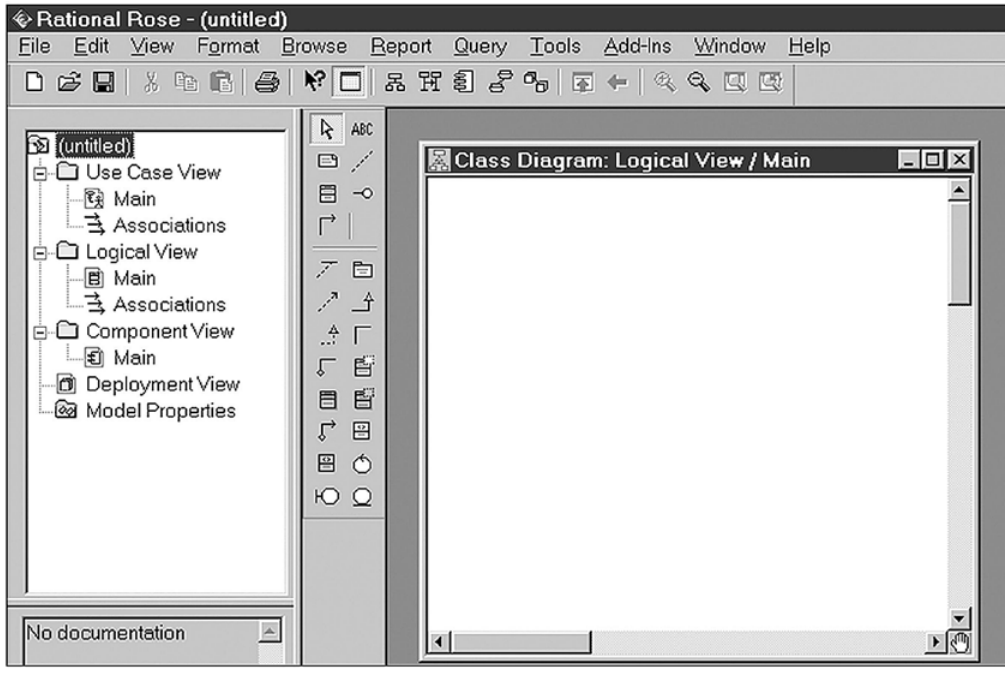

Рис. 2.1. **Интерфейс *Rational Rose***

Браузер поддерживает четыре представления (*view*): вариантов использования, компонентов, размещения и логическое представление. Браузер представлен в древовидном стиле. Каждый элемент модели может содержать другие элементы, находящиеся ниже его в иерархии. Знак «−» около элемента означает, что его ветвь полностью раскрыта. Знак «+»— что его ветвь свернута.

Окно документации

В этом окне можно документировать элементы модели *Rose*. Например, можно сделать короткое описание каждого действующего лица. При документировании класса все, что будет написано в окне документации, появится затем как комментарий в сгенерированном коде, что избавляет от необходимости впоследствии вносить эти комментарии вручную. Документация будет выводиться также в отчетах, создаваемых в среде *Rose*.

Панели инструментов

Панели инструментов *Rational Rose* обеспечивают быстрый доступ к наиболее распространенным командам. В этой среде существует два типа панелей инструментов: стандартная и панель диаграммы. Стандартная панель видна всегда, ее кнопки соответствуют командам, которые могут использоваться для работы с любой диаграммой. Панель диаграммы своя для каждого типа диаграмм UML.

Все панели инструментов могут быть изменены и настроены пользователем. Для этого выберите пункт меню *Tools* > *Options*, затем выберите вкладку *Toolbars*.

Чтобы показать или скрыть стандартную панель инструментов (или панель инструментов диаграммы):

1. Выберите пункт *Tools* > *Options*.
2. Выберите вкладку *Toolbars*.
3. Чтобы сделать видимой/невидимой стандартную панель инструментов, пометьте/снимите контрольный переключатель *Show Standard ToolBar* (или *Show Diagram ToolBar*).

Чтобы увеличить размер кнопок на панели инструментов:

1. Щелкните правой кнопкой мыши на требуемой панели.
2. Выберите во всплывающем меню пункт *Use Large Buttons* (Использовать большие кнопки).

Чтобы настроить панель инструментов:

1. Щелкните правой кнопкой мыши на требуемой панели.
2. Выберите пункт *Customize* (настроить).
3. Чтобы добавить или удалить кнопки, выберите соответствующую кнопку и щелкните мышью на кнопке *Add* (добавить) или *Remove* (удалить) (рис. 2.2).

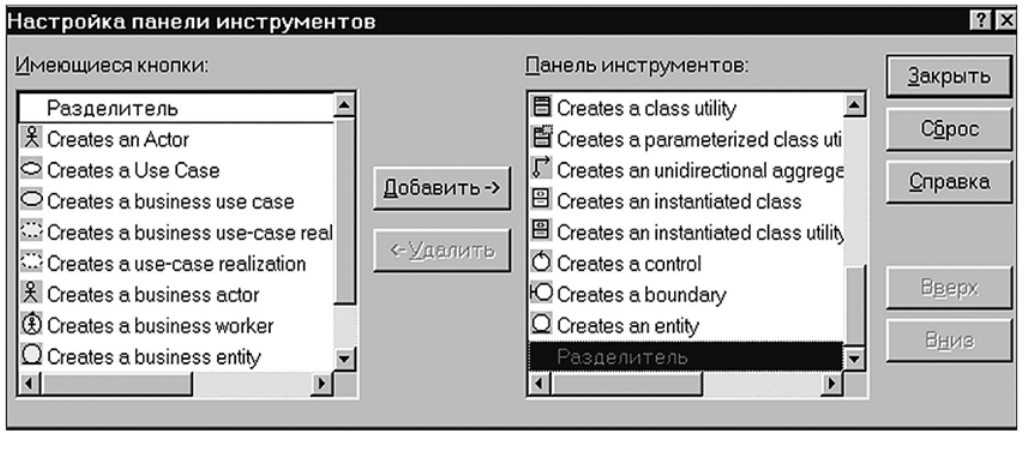

Рис. 2.2. **Настройка стандартной панели инструментов**

Окно диаграммы

В окне диаграммы видно, как выглядит одна или несколько диаграмм модели UML. При внесении в элементы диаграммы изменений *Rational Rose* автоматически обновит браузер. Аналогично, при внесении изменений в элемент с помощью браузера *Rational Rose* автоматически обновит соответствующие диаграммы. Это помогает поддерживать модель в непротиворечивом состоянии.

Журнал

По мере работы над вашей моделью определенная информация будет направляться в окно журнала. Например, туда помещаются сообщения об ошибках, возникающих при генерации кода. Не существует способа закрыть журнал совсем, но его окно может быть минимизировано.

2.2. Представление моделей в *IBM Rational Rose*

В модели *Rational Rose* поддерживается четыре представления (*views*) — вариантов использования, логическое, компонентов и размещения. Каждое из них предназначено для своих целей и определенной аудитории.

Представление вариантов использования

Представление содержит все действующие лица, все варианты использования и их диаграммы для конкретной системы. Оно может также содержать некоторые диаграммы последовательности и кооперативные диаграммы. На рис. 2.3 показано представление вариантов использования в браузере *Rational Rose*.

Представление вариантов использования содержит:

— действующие лица;

— варианты использования;

— документацию по вариантам использования, детализирующую происходящие в них процессы (потоки событий), включая обработку ошибок. Эта пиктограмма соответствует внешнему файлу, прикрепленному к модели *Rational Rose*. Вид пиктограммы зависит от приложения, используемого для документирования потока событий;

— диаграммы вариантов использования. Обычно у системы бывает несколько таких диаграмм, каждая из которых пока-

зывает подмножество действующих лиц и (или) вариантов использования;

— пакеты, являющиеся группами вариантов использования и (или) действующих лиц.

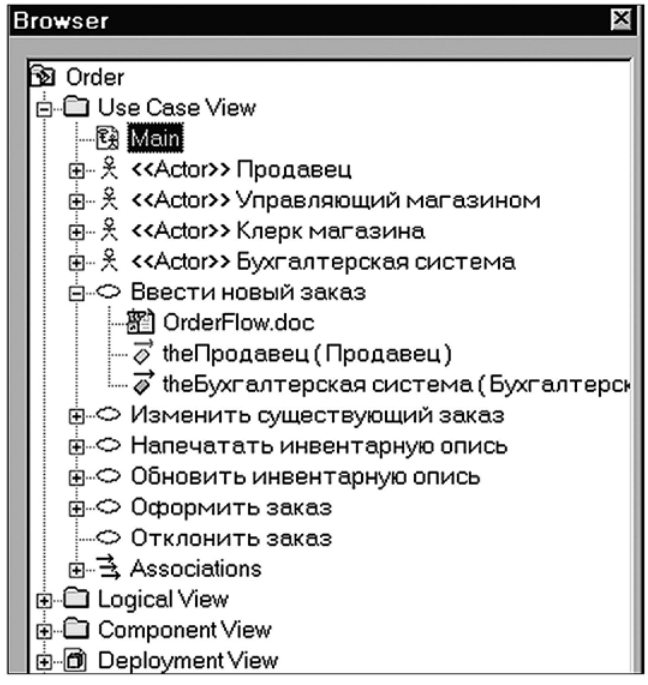

Рис. 2.3. **Представление вариантов использования**

Логическое представление

Логическое представление (рис. 2.4) концентрируется на том, как система будет реализовывать поведение, описанное в вариантах использования. Оно дает подробную картину составных частей системы и описывает их взаимодействие. Логическое представление включает в себя, помимо прочего, конкретные требуемые классы, диаграммы классов и диаграммы состояний. G их помощью конструируется детальный проект создаваемой системы.

Логическое представление содержит:

— классы;

— диаграммы классов. Как правило, для описания системы используется несколько диаграмм классов, каждая из которых отображает некоторое подмножество всех классов системы;

— диаграммы взаимодействия, применяемые для отображения объектов, участвующих в одном потоке событий варианта использования;

— диаграммы состояний;
— пакеты, являющиеся группами взаимосвязанных классов.

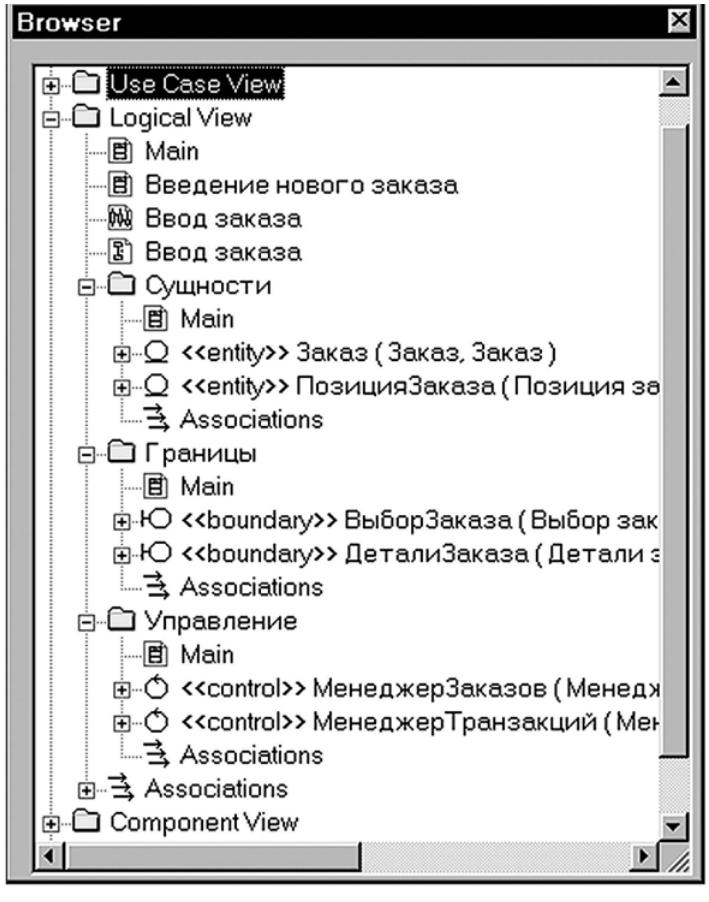

Рис. 2.4. **Логическое представление системы**

Представление компонентов

Представление компонентов содержит:
— компоненты, являющиеся физическими модулями кода;
— диаграммы компонентов;
— пакеты, являющиеся группами связанных компонентов.

Представление размещения

Последнее представление *Rational Rose* — это представление размещения. Оно соответствует физическому размещению системы, которое может отличаться от ее логической архитектуры.

В представление размещения входят:
— процессы, являющиеся потоками (*threads*), исполняемыми в отведенной для них области памяти;

— процессоры, включающие любые компьютеры, способные обрабатывать данные. Любой процесс выполняется на одном или нескольких процессорах;

— устройства, т. е. любая аппаратура, не способная обрабатывать данные. К числу таких устройств относятся, например, терминалы ввода-вывода и принтеры;

— диаграмма размещения.

2.3. Параметры настройки отображения

Изображение атрибутов и операций на диаграммах классов

В *Rational Rose* имеется возможность настроить диаграммы классов так, чтобы:

— показывать все атрибуты и операции;

— скрыть операции;

— скрыть атрибут;

— показывать только некоторые атрибуты или операции;

— показывать операции вместе с их полными сигнатурами или только их имена;

— показывать/не показывать видимость атрибутов и операций;

— показывать/не показывать стереотипы атрибутов и операций.

Значения каждого параметра по умолчанию можно задать с помощью окна, открываемого при выборе пункта меню *Tools* > *Options*.

У данного класса на диаграмме можно:

— показать все атрибуты;

— скрыть все атрибуты;

— показать только выбранные вами атрибуты;

— подавить вывод атрибутов.

Подавление вывода атрибутов приведет не только к исчезновению атрибутов с диаграммы, но и к удалению линии, показывающей место расположения атрибутов в классе.

Существует два способа изменения параметров представления атрибутов на диаграмме. Можно установить нужные значения у каждого класса индивидуально. Можно также изменить значения нужных параметров по умолчанию до начала создания диаграммы классов. Внесенные таким образом изменения повлияют только на вновь создаваемые диаграммы.

Чтобы показать все атрибуты класса:

1. Выделите на диаграмме нужный класс.
2. Щелкните на нем правой кнопкой мыши, чтобы открыть контекстно-зависимое меню.
3. В нем выберите *Options > Show All Attributes*.

Чтобы показать у класса только избранные атрибуты:

1. Выделите на диаграмме нужный вам класс.
2. Щелкните на нем правой кнопкой мыши, чтобы открыть контекстно-зависимое меню.
3. В нем выберите *Options > Select Compartment Items*.
4. Укажите нужные вам атрибуты в окне *Edit Compartment*.

Чтобы подавить вывод всех атрибутов класса диаграммы:

1. Выделите на диаграмме нужный вам класс.
2. Щелкните на нем правой кнопкой мыши, чтобы открыть контекстно-зависимое меню.
3. В нем выберите *Options > Suppress Attributes*.

Чтобы изменить принятый по умолчанию вид атрибута:

1. В меню модели выберите пункт *Tools > Options*.
2. Перейдите на вкладку *Diagram*.
3. Для установки значений параметров отображения атрибутов по умолчанию воспользуйтесь контрольными переключателями *Suppress Attributes* и *Show All Attributes*. Изменение этих значении по умолчанию повлияет только на новые диаграммы. Вид существующих диаграмм классов не изменится.

Как и в случае атрибутов, имеется несколько вариантов представления операций на диаграммах:

— показать все операции;
— показать только некоторые операции;
— скрыть все операции;
— подавить вывод операций.

Чтобы показать все операции класса:

1. Выделите на диаграмме нужный вам класс.
2. Щелкните на нем правой кнопкой мыши, чтобы открыть контекстно-зависимое меню.
3. В нем выберите *Options > Show All Operations*.

Чтобы показать только избранные операции класса:

1. Выделите на диаграмме нужный вам класс.
2. Щелкните на нем правой кнопкой мыши, чтобы открыть контекстно-зависимое меню.
3. В нем выберите *Options > Select Compartment Items*.
4. Укажите нужные вам операции в окне *Edit Compartment*.

Чтобы подавить вывод всех операций класса диаграммы:

1. Выделите на диаграмме нужный вам класс.
2. Щелкните на нем правой кнопкой мыши, чтобы открыть контекстно-зависимое меню.
3. В нем выберите *Options > Suppress Operations*.

Чтобы показать на диаграмме классов сигнатуру операции:

1. Выделите на диаграмме нужный вам класс.
2. Щелкните на нем правой кнопкой мыши, чтобы открыть контекстно-зависимое меню.
3. В нем выберите *Options > Show Operation Signature*.

Чтобы изменить принятый по умолчанию вид операции:

1. В меню модели выберите пункт *Tools > Options*.
2. Перейдите на вкладку *Diagram*.
3. Для установки значений параметров отображения операций по умолчанию воспользуйтесь контрольными переключателями *Suppress Operations, Show All Operations* и *Show Operation Signatures*.

Чтобы показать видимость атрибута или операции класса:

1. Выделите на диаграмме нужный вам класс.
2. Щелкните на нем правой кнопкой мыши, чтобы открыть контекстно-зависимое меню.
3. В нем выберите *Options > Show Visibility*.

Чтобы изменить принятое по умолчанию значение параметра показа видимости:

1. В меню модели выберите пункт *Tools > Options*.
2. Перейдите на вкладку *Diagram*.
3. Для установки параметров отображения видимости по умолчанию воспользуйтесь контрольным переключателем *Show Visibility*.

Для переключения между нотациями видимости *Rational Rose* и UML:

1. В меню модели выберите пункт *Tools > Options*.
2. Перейдите на вкладку *Notation*.
3. Для переключения между нотациями воспользуйтесь переключателем *Visibility as Icons*. Если этот переключатель помечен, будет использоваться нотация *Rational Rose*. Если нет, то нотация UML. Изменение этого параметра повлияет только на новые диаграммы. Существующие диаграммы классов останутся прежними.

Глава 3
ПОСТРОЕНИЕ МОДЕЛЕЙ В *IBM RATIONAL ROSE*

Упражнение 1. Создание диаграммы вариантов использования

Создайте диаграмму вариантов использования для системы обработки заказов. Требуемые для этого действия подробно перечислены далее (рис. 3.1).

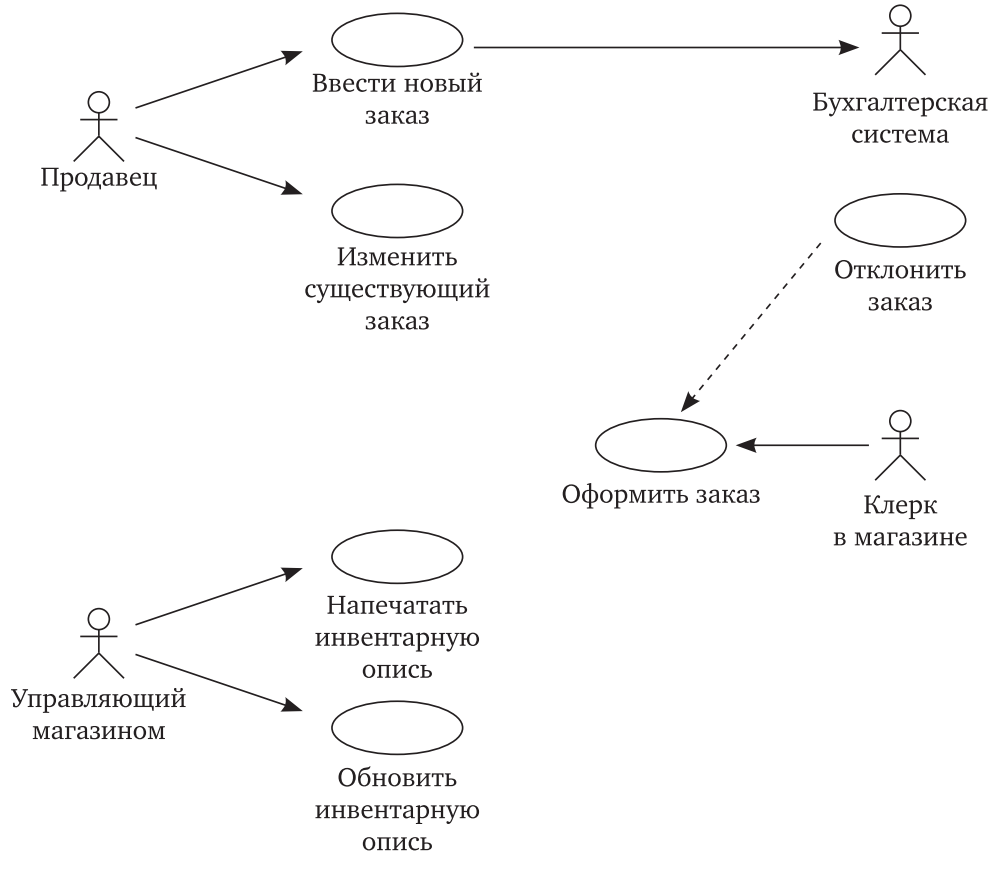

Рис. 3.1. **Диаграмма вариантов использования для системы обработки заказов**

Выполнение упражнения

Создать диаграмму вариантов использования:

1. Дважды щелкните на главной диаграмме вариантов использования (*Main*) в браузере, чтобы открыть ее.
2. С помощью кнопки *Use* Case панели инструментов поместите на диаграмму новый вариант использования.
3. Назовите этот вариант использования «Ввести новый заказ».
4. Повторите шаги 2 и 3, чтобы поместить на диаграмму остальные варианты использования: Изменить существующий заказ, Напечатать инвентарную опись, Обновить инвентарную опись, Оформить заказ, Отклонить заказ.
5. С помощью кнопки *Actor* панели инструментов поместите на диаграмму новое действующее лицо.
6. Назовите его «Продавец».
7. Повторите шаги 5 и 6, поместив на диаграмму остальных действующих лиц: Управляющий магазином, Клерк магазина, Бухгалтерская система.

Пометить абстрактные варианты использования:

1. Щелкните правой кнопкой мыши на варианте использования «Отклонить заказ» на диаграмме.
2. В открывшемся меню выберите пункт *Open Specification* (Открыть спецификацию).
3. Пометьте контрольный переключатель *Abstract* (Абстрактный), чтобы сделать вариант использования абстрактным.

Добавить ассоциации:

1. С помощью кнопки *Unidirectional Association* (Однонаправленная ассоциация) панели инструментов нарисуйте ассоциацию между действующим лицом Продавец и вариантом использования «Ввести новый заказ».
2. Повторите этот шаг, чтобы поместить на диаграмму остальные ассоциации.

Добавить связь расширения:

1. С помощью кнопки *Dependency* панели инструментов нарисуйте связь между вариантом использования «Отклонить заказ» и вариантом использования «Оформить заказ». Стрелка должна протянуться от первого варианта использования ко второму. Связь расширения означает, что вариант использования «Отклонить заказ» при необходимости дополняет функциональные возможности варианта использования «Оформить заказ».

2. Щелкните правой кнопкой мыши на новой связи между вариантами использования «Отклонить заказ» и «Оформить заказ».

3. В открывшемся меню выберите пункт *Open Specification* (Открыть спецификацию).

4. В раскрывающемся списке стереотипов введите слово *extend* (расширение), затем нажмите ОК.

5. Слово «*extend*» появится на линии данной связи.

Добавить описания к вариантам использования:

1. Выделите в браузере вариант использования «Ввести новый заказ».

2. В окне документации введите следующее описание к этому варианту использования: «Этот вариант использования дает клиенту возможность ввести новый заказ в систему».

3. С помощью окна документации введите описания ко всем остальным вариантам использования.

Добавить описания к действующему лицу:

1. Выделите в браузере действующее лицо Продавец.

2. В окне документации введите для этого действующего лица следующее описание: «Продавец — это служащий, доставляющий и старающийся продать продукцию».

3. С помощью окна документации введите описания к оставшимся действующим лицам.

Прикрепление файла к варианту использования:

1. Для описания главного потока событий варианта использования «Ввести новый заказ» создайте файл *OrderFlow.doc*, содержащий следующий текст:

1. Продавец выбирает пункт «Создать новый заказ» из имеющегося меню.

2. Система выводит форму «Подробности заказа».

3. Продавец вводит номер заказа, заказчика и то, что заказано.

4. Продавец сохраняет заказ.

5. Система создает новый заказ и сохраняет его в базе данных.

2. Щелкните правой кнопкой мыши на варианте использования «Ввести новый заказ».

3. В открывшемся меню выберите пункт *Open Specification* (Открыть спецификацию)

4. Перейдите на вкладку файлов.

5. Щелкните правой кнопкой мыши на белом поле и из открывшегося меню выберите пункт *Insert File* (Ввести файл).

6. Укажите файл *OpenFlow.doc* и нажмите на кнопку *Open* (Открыть), чтобы прикрепить файл к варианту использования.

Упражнение 2. Создание диаграмм взаимодействия

Создайте диаграмму последовательности и кооперативную диаграмму, отражающую ввод нового заказа в систему обработки заказов (рис. 3.2).

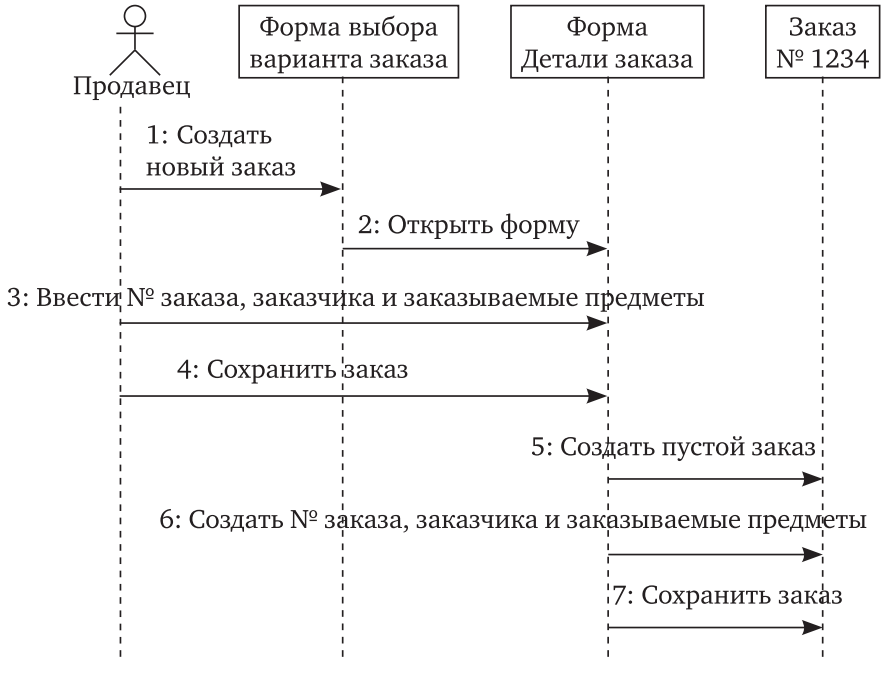

Рис. 3.2. **Диаграмма последовательности ввода нового заказа после завершения первого этапа работы**

Это только одна из диаграмм, необходимых для моделирования варианта использования «Ввести новый заказ». Она соответствует успешному варианту хода событий. Для описания того, что случится, если возникнет ошибка, или если пользователь выберет другие действия из предложенных, придется разработать другие диаграммы. Каждый альтернативный поток варианта использования может быть промоделирован с помощью своих собственных диаграмм взаимодействия.

Выполнение упражнения

Настройка:

1. В меню модели выберите пункт *Tools* > *Options*.

2. Перейдите на вкладку диаграмм.

3. Контрольные переключатели *Sequence Numbering, Collaboration Numbering* и *Focus of Control* должны быть помечены.

4. Нажмите ОК, чтобы выйти из окна параметров.

Создание диаграммы последовательности:

1. Щелкните правой кнопкой мыши на Логическом представлении браузера.

2. В открывшемся меню выберите пункт *New > Sequence Diagram*.

3. Назовите новую диаграмму «Ввод заказа».

4. Дважды щелкните на ней, чтобы открыть ее.

Добавление на диаграмму действующего лица и объектов:

1. Перетащите действующее лицо Продавец (*Salesperson*) с браузера на диаграмму.

2. На панели инструментов нажмите кнопку *Object* (Объект).

3. Щелкните мышью в верхней части диаграммы, чтобы поместить туда новый объект.

4. Назовите объект «*Order Options Form* — Выбор варианта заказа».

5. Повторите шаги 3 и 4, чтобы поместить на диаграмму все остальные объекты:

«*Order Detail Form*» — Форма Детали заказа.

«*Order* N1234» — Заказ № 1234.

Добавление сообщений на диаграмму:

1. На панели инструментов нажмите кнопку *Object Message* (Сообщение объекта).

2. Проведите мышью от линии жизни действующего лица Продавец к линии жизни объекта Выбор варианта заказа.

3. Выделив сообщение, введите его имя «*Create New Order*» — Создать новый заказ.

4. Повторите шаги 2 и 3, чтобы поместить на диаграмму дополнительные сообщения:

Open form — Открыть форму (между Выбором варианта заказа и Деталями заказа).

Enter order number, customer, order items — Ввести номер заказа, заказчика и число заказываемых предметов (между Продавцом и Деталями заказа).

Save the order — Сохранить заказ (между Продавцом и Деталями заказа).

Create new, blank order — Создать пустой заказ (между Деталями заказа и Заказом № 1234).

Set the order number, customer, order items — Ввести номер заказа, заказчика и число заказываемых предметов (между Деталями заказа и Заказом № 1234).

Save the order — Сохранить заказ (между Деталями заказа и Заказом № 1234).

Мы завершили первый этап работы. Готовая диаграмма последовательности показана на рис. 3.2. Теперь надо позаботиться об управляющих объектах и взаимодействии с базой данных. Как очевидно из диаграммы, объект Детали заказа имеет множество обязанностей, с которыми лучше всего мог бы справиться управляющий объект. Кроме того, новый заказ должен сохранять себя в базе данных сам. Эту обязанность лучше было бы переложить на другой объект.

Добавление на диаграмму дополнительных объектов:

1. На панели инструментов нажмите кнопку *Object*.
2. Щелкните мышью между объектами Детали заказа и Заказ № 1234, чтобы поместить туда новый объект.
3. Введите имя объекта — *Order Manager* (Управляющий заказами).
4. На панели инструментов нажмите кнопку *Object*.
5. Новый объект расположите справа от Заказа № 1234.
6. Введите его имя — *Transaction Manager* (Управляющий транзакциями).

Назначение обязанностей объектам:

1. Выделите сообщение 5 (Создать пустой заказ).
2. Нажмите комбинацию клавиш CTRL + Del, чтобы удалить это сообщение.
3. Повторите шаги 1 и 2, чтобы удалить два последних сообщения:

Вести номер заказа, заказчика и число заказываемых предметов.

Сохранить заказ.

4. На панели инструментов нажмите кнопку *Object Message*.
5. Поместите на диаграмму новое сообщение, расположив его под сообщением 4 между Деталями заказа и Управляющим заказами.
6. Назовите его *Save the order* (Сохранить заказ).
7. Повторите шаги 4—6, добавив сообщения с шестого по девятое и назвав их:

Create new, blank order (Создать новый заказ) — между Управляющим заказами и Заказом № 1234.

Set the order number, customer, order items (Вести номер заказа, заказчика и число заказываемых предметов) — между Управляющим заказами и Заказом № 1234.

Save the order (Сохранить заказ) — между Управляющим заказами и Управляющим транзакциями.

Collect order information (Информация о заказе) — между Управляющим транзакциями и Заказом № 1234.

8. На панели инструментов нажмите кнопку *Message to Self* (Сообщение себе).

9. Щелкните на линии жизни объекта Управляющий транзакциями ниже сообщения 9, добавив туда рефлексивное сообщение.

Соотнесение объектов с классами:

1. Щелкните правой кнопкой мыши на объекте Выбор варианта заказа.

2. В открывшемся меню выберите пункт *Open Specification* (Открыть спецификацию).

3. В раскрывающемся списке классов выберите пункт *<New>* (Создать). Появится окно спецификации классов.

10. Назовите его *Save the order information to the database* (Сохранить информацию о заказе в базе данных).

Диаграмма последовательности с новыми объектами показана на рис. 3.3.

4. В поле имени введите имя *OrderOptions* (Выбор заказа).

5. Щелкните на кнопке ОК. Вы вернетесь к окну спецификации объекта.

6. В списке классов выберите теперь класс *OrderOptions*.

7. Щелкните на кнопке ОК, чтобы вернуться к диаграмме. Теперь объект называется *Order Options Form* : *OrderOptions* (Выбор варианта заказа : *OrderOptions*).

8. Для соотнесения остальных объектов с классами повторите шаги с 1 по 7:

Класс *OrderDetail* соотнесите с объектом Детали заказа.

Класс *OrderMgr* — с объектом Управляющий заказами.

Класс *Order* — с объектом Заказ № 1234.

Класс *TransactionMgr* — с объектом Управляющий транзакциями.

После завершения этих действий диаграмма имеет вид, как показано на рис. 3.4.

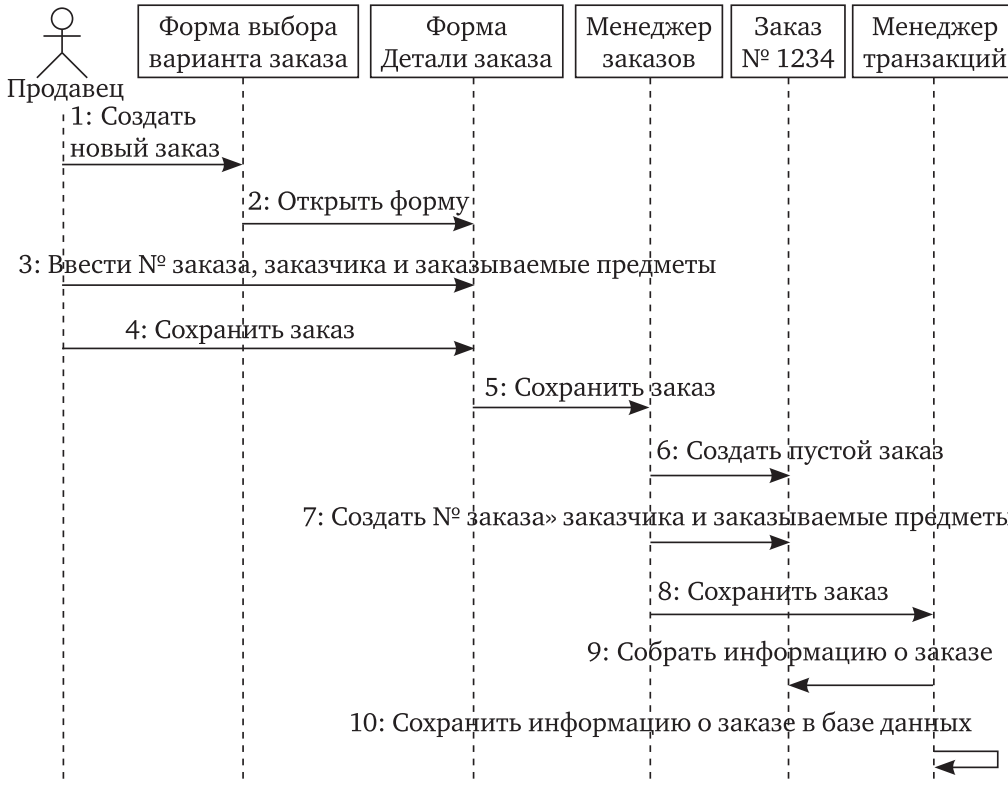

Рис. 3.3. **Диаграмма последовательности с новыми объектами**

Рис. 3.4. **Диаграмма последовательности с именами классов**

Соотнесение сообщений с операциями:

1. Щелкните правой кнопкой на сообщении 1, Создать новый заказ.

2. В открывшемся меню выберите пункт *<new operation>* (создать операцию). Появится окно спецификации операции.

3. В поле имени введите имя операции — *Create* (Создать).

4. Нажмите на кнопку ОК, чтобы закрыть окно спецификации операции и вернуться на диаграмму.

5. Еще раз щелкните правой кнопкой мыши на сообщении 1.

6. В открывшемся меню выберите новую операцию *Create*().

7. Повторите сообщения с 1 по 6, пока не соотнесете с операциями все остальные сообщения:

Сообщение 2: Открыть соотнесите с операцией *Open*().

Сообщение 3: Ввести номер заказа, заказчика и число заказываемых предметов — с операцией *SubmitInfo*().

Сообщение 4: Сохранить заказ — с операцией *Save*().

Сообщение 5: Сохранить заказ — с операцией *SaveOrder*().

Сообщение 6: Создать пустой заказ — с операцией *Create*().

Сообщение 7: Ввести номер заказа, заказчика и число заказываемых предметов — с операцией *SetInfo*().

Сообщение 8: Сохранить заказ — с операцией *SaveOrder*().

Сообщение 9: Информация о заказе — с операцией *Getlnfo*().

Сообщение 10: Сохранить информацию о заказе в базе данных — с операцией *Commit*.

Диаграмма имеет вид, как показано на рис. 3.5.

Создание кооперативной диаграммы:

Для создания кооперативной диаграммы достаточно просто нажать клавишу F5 или, если вы хоти сами проделать все требуемые операции, воспользуйтесь приводимым далее планом.

1. Щелкните правой кнопкой мыши на логическом представлении в браузере.

2. В открывшемся меню выберите пункт *New > Collaboration Diagram*.

3. Назовите эту диаграмму Ввод заказа.

4. Щелкните на ней дважды, чтобы открыть ее.

Добавление действующего лица и объектов на диаграмму:

1. Перетащите действующее лицо Продавец (*Salesperson*) с браузера на диаграмму.

2. На панели инструментов нажмите кнопку *Object* (Объект).

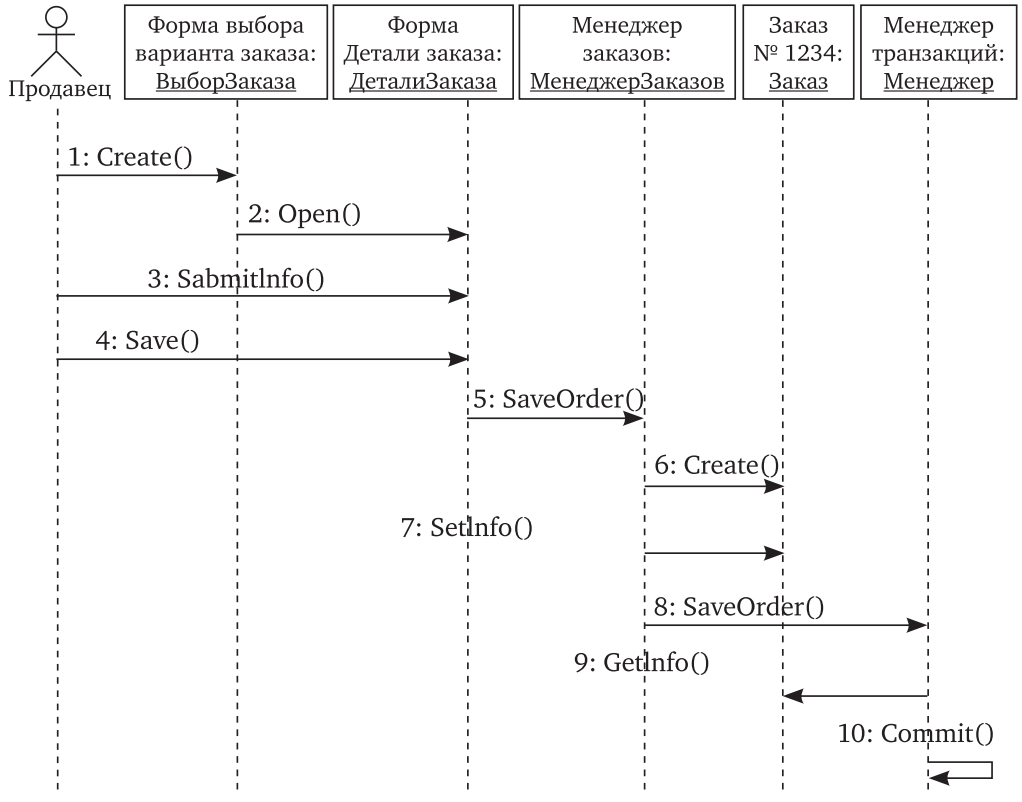

Рис. 3.5. **Диаграмма последовательности с операциями**

3. Щелкните мышью где-нибудь внутри диаграммы, чтобы поместить туда новый объект.

4. Назовите объект «*Order Options Form*» — Выбор варианта заказа.

5. Повторите шаги 3 и 4, чтобы поместить на диаграмму все остальные объекты:

«*Order Detail Form*» — Форма Детали заказа.

«*Order* N1234» — Заказ № 1234.

Добавление сообщений на диаграмму:

1. На панели инструментов нажмите кнопку *Object Link* (Связь объекта).

2. Проведите мышью от действующего лица Продавец к объекту Выбор варианта заказа.

3. Повторите шаги 1 и 2, соединив связями следующие объекты:

Действующее лицо Продавец и объект Детали Заказа.

Объект Выбор варианта заказа и объект Детали заказа.

Объект Детали заказа и объект Заказ № 1234.

4. На панели инструментов нажмите кнопку *Link Message* (Сообщение связи).

5. Щелкните на связи между Продавцом и Выбором варианта заказа.

6. Выделив сообщение, введите его имя «*Create New Order*» — Создать новый заказ.

7. Повторите шаги с 4 по 6, поместив на диаграмму все остальные сообщения, как показано далее:

Open form — Открыть форму (между Выбором варианта заказа и Деталями заказа).

Enter order number, customer, order items — Ввести номер заказа, заказчика и число заказываемых предметов (между Продавцом и Деталями заказа).

Save the order — Сохранить заказ (между Продавцом и Деталями заказа).

Create new, blank order — Создать пустой заказ (между Деталями заказа и Заказом № 1234).

Set the order number, customer, order items — Ввести номер заказа, заказчика и число заказываемых предметов (между Деталями заказа и Заказом № 1234).

Save the order — Сохранить заказ (между Деталями заказа и Заказом № 1234).

Теперь, как и раньше, надо продолжить работу и поместить на диаграмму дополнительные элементы, а также рассмотреть обязанности объектов.

Добавление на диаграмму дополнительных объектов:

1. На панели инструментов нажмите кнопку *Object*.

2. Щелкните мышью где-нибудь на диаграмме, чтобы поместить туда новый объект.

3. Введите имя объекта — *Order Manager* (Управляющий заказами).

4. На панели инструментов нажмите кнопку *Object*.

5. Поместите на диаграмму еще один объект.

6. Введите его имя — *Transaction Manager* (Управляющий транзакциями).

Назначение обязанностей объектам:

1. Выделите сообщение 5 (Создать пустой заказ). Выделяйте слова, а не стрелку.

2. Нажмите комбинацию клавиш CTRL + Del, чтобы удалить это сообщение.

3. Повторите шаги 1 и 2, чтобы удалить сообщения 6 и 7:

Вести номер заказа, заказчика и число заказываемых предметов.

Сохранить заказ.

4. Выделите связь между объектами Детали заказа и Заказ № 1234.

5. Нажмите комбинацию клавиш CTRL + Del, чтобы удалить эту связь.

6. На панели инструментов нажмите кнопку *Object Link* (Связь объекта).

7. Нарисуйте связь между Деталями Заказа и Управляющим заказами.

8. На панели инструментов нажмите кнопку *Object Link* (Связь объекта).

9. Нарисуйте связь между Управляющим заказами и Заказом № 1234.

10. На панели инструментов нажмите кнопку *Object Link* (Связь объекта).

11. Нарисуйте связь между Заказом № 1234 и Управляющим транзакций.

12. На панели инструментов нажмите кнопку *Object Link* (Связь объекта).

13. Нарисуйте связь между Управляющим заказами и Управляющим транзакций.

14. На панели инструментов нажмите кнопку *Link Message* (Сообщение связи).

15. Щелкните на связи между объектами Детали заказа и Управляющим заказами, чтобы ввести новое сообщение.

16. Назовите это сообщение *Save the order* (Сохранить заказ).

17. Повторите шаги 14—16, добавив сообщения с шестого по девятое и назвав их:

Create new, blank order (Создать новый заказ) — между Управляющим заказами и Заказом № 1234.

Set the order number, customer, order items (Вести номер заказа, заказчика и число заказываемых предметов) — между Управляющим заказами и Заказом № 1234.

Save the order (Сохранить заказ) — между Управляющим заказами и Управляющим транзакциями.

Collect order information (Информация о заказе) — между Управляющим транзакциями и Заказом № 1234.

18. На панели инструментов нажмите кнопку *Message to Self* (Сообщение себе).

19. Щелкните на объекте Управляющий транзакциями, добавив к нему рефлексивное сообщение.

20. На панели инструментов нажмите кнопку *Link Message* (Сообщение связи).

21. Щелкните на рефлексивной связи Управляющего транзакциями, чтобы ввести туда сообщение.

22. Назовите новое сообщение *Save the order information to the database* (Сохранить информацию о заказе в базе данных).

Соотнесение объектов с классами (если при разработке диаграммы последовательности классы вы создали):

1. Найдите в браузере класс *OrderOptions*.
2. Перетащите его на объект Выбор варианта заказа на диаграмме.
3. Повторите шаги 1 и 2, соотнеся остальные объекты и соответствующие им классы:

Класс *OrderDetail* соотнесите с объектом Детали заказа.
Класс *OrderMgr* — с объектом Управляющий заказами.
Класс *Order* — с объектом Заказ № 1234.
Класс *TransactionMgr* — с объектом Управляющий транзакциями.

Соотнесение объектов с классами (если при разработке диаграммы последовательности классы не создавались):

1. Щелкните правой кнопкой мыши на объекте Выбор варианта заказа.
2. В открывшемся меню выберите пункт *Open Specification* (Открыть спецификацию).
3. В раскрывающемся списке классов выберите пункт *<New>* (Создать). Появится окно спецификации классов.
4. В поле имени введите имя *OrderOptions* (Выбор заказа).
5. Щелкните на кнопке ОК. Вы вернетесь к окну спецификации объекта.
6. В списке классов выберите теперь класс *OrderOptions*.
7. Щелкните на кнопке ОК, чтобы вернуться к диаграмме. Теперь объект называется *Order Options Form* : *OrderOptions* (Выбор варианта заказа: *OrderOptions*).
8. Для соотнесения остальных объектов с классами повторите шаги с 1 по 7:

Класс *OrderDetail* соотнесите с объектом Детали заказа.
Класс *OrderMgr* — с объектом Управляющий заказами.
Класс *Order* — с объектом Заказ № 1234.
Класс *TransactionMgr* — с объектом Управляющий транзакциями.

Соотнесение сообщений с операциями (если при разработке диаграммы последовательности операции вы создали):

1. Щелкните правой кнопкой на сообщении 1, Создать новый заказ.
2. В открывшемся меню выберите пункт *Open Specification* (Открыть спецификацию).
3. В раскрывающемся списке имен укажите имя операции — *Create* (Создать).
4. Нажмите на кнопку ОК.
5. Повторите этапы с первого по четвертый для соотнесения с операциями остальных сообщений:

Сообщение 2: Открыть соотнесите с операцией *Open*().
Сообщение 3: Ввести номер заказа, заказчика и число заказываемых предметов — с операцией *SubmitInfo*().
Сообщение 4: Сохранить заказ — с операцией *Save*().
Сообщение 5: Сохранить заказ — с операцией *SaveOrder*().
Сообщение 6: Создать пустой заказ — с операцией *Create*().
Сообщение 7: Ввести номер заказа, заказчика и число заказываемых предметов — с операцией *SetInfo*().
Сообщение 8: Сохранить заказ — с операцией *SaveOrder*().
Сообщение 9: Информация о заказе — с операцией *GetInfo*().
Сообщение 10: Сохранить информацию о заказе в базе данных — с операцией *Commit*().

Соотнесение сообщений с операциями (если при разработке диаграммы последовательности операции не создавались):

1. Щелкните правой кнопкой на сообщении 1, Создать новый заказ.
2. В открывшемся меню выберите пункт *<new operation>* (создать операцию). Появится окно спецификации операции.
3. В поле имени введите имя операции — *Create* (Создать).
4. Нажмите на кнопку ОК, чтобы закрыть окно спецификации операции и вернуться на диаграмму.
5. Еще раз щелкните правой кнопкой мыши на сообщении 1.
6. В открывшемся меню выберите пункт *Open Specification* (Открыть спецификацию).
7. В раскрывающемся списке *Name* (имя) укажите имя новой операции.
8. Нажмите на кнопку ОК.
9. Повторите этапы с первого по восьмой, чтобы создать новые операции и соотнести с ними остальные сообщения:

Сообщение 2: Открыть соотнесите с операцией *Open*().

Сообщение 3: Ввести номер заказа, заказчика и число заказываемых предметов — с операцией *SubmitInfo*().

Сообщение 4: Сохранить заказ — с операцией *Save*().

Сообщение 5: Сохранить заказ — с операцией *SaveOrder*().

Сообщение 6: Создать пустой заказ — с операцией *Create*().

Сообщение 7: Ввести номер заказа, заказчика и число заказываемых предметов — с операцией *SetInfo*().

Сообщение 8: Сохранить заказ — с операцией *SaveOrder*().

Сообщение 9: Информация о заказе — с операцией *GetInfo*().

Сообщение 10: Сохранить информацию о заказе в базе данных — с операцией *Commit*.

Упражнение 3. Создание диаграммы классов

Объедините обнаруженные вами классы в пакеты. Создайте диаграмму классов для отображения пакетов, диаграммы классов для представления классов в каждом пакете и диаграмму классов для представления всех классов варианта использования «Ввести новый заказ».

Выполнение упражнения

Настройка:

1. В меню модели выберите пункт *Tools* > *Options* (Инструменты > Параметры).
2. Перейдите на вкладку диаграмм.
3. Убедитесь, что помечен контрольный переключатель *Show Stereotypes* (Показать стереотипы).
4. Убедитесь, что помечены контрольные переключатели *Show All Attributes* (Показать все атрибуты) и *Show All Operations* (Показать все операции).
5. Убедитесь, что не помечены переключатели *Suppress Attributes* (Подавить вывод атрибутов) и *Suppress Operations* (Подавить вывод операций).

Создание пакетов:

1. Щелкните правой кнопкой мыши на логическом представлении браузера.
2. В открывшемся меню выберите пункт *New* > *Package* (Создать > пакет).
3. Назовите новый пакет *Entities* (Сущности).

4. Повторите этапы с первого по третий, создав пакеты *Boundaries* (границы) и *Control* (управление).

Создание главной диаграммы классов:

1. Дважды щелкните на главной диаграмме классов прямо под логическим представлением браузера, чтобы открыть ее.
2. Перетащите пакет *Entities* из браузера на диаграмму.
3. Перетащите пакеты *Boundaries* и *Control* из браузера на диаграмму.

Главная диаграмма классов показана на рис. 3.6.

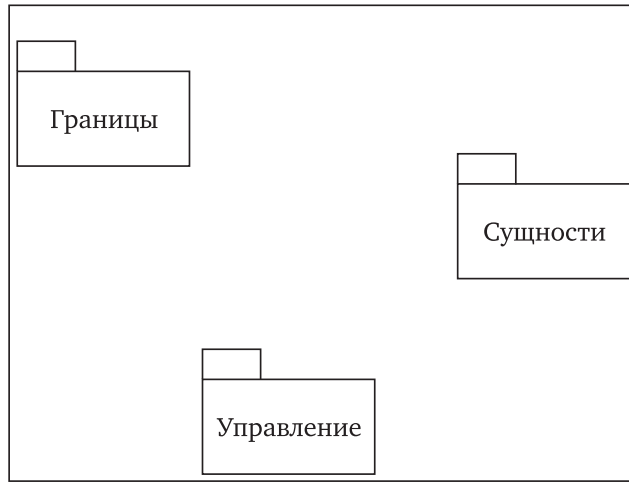

Рис. 3.6. **Главная диаграмма классов системы обработки заказов**

Создание диаграммы классов для сценария «Ввести новый заказ» со всеми классами:

1. Щелкните правой кнопкой мыши на логическом представлении браузера.
2. В открывшемся меню выберите пункт *New > Class Diagram* (Создать > Диаграмму классов).
3. Назовите новую диаграмму Классов *Add New Order* (Введение нового заказа).
4. Щелкните в браузере на этой диаграмме дважды, чтобы открыть ее.
5. Перетащите из браузера все классы (*OrderOptions, OrderDetail, Order, OrderMgr* и *TransactionMgr*).

Диаграмма классов показана на рис. 3.7.

Добавление стереотипов к классам:

1. Щелкните правой кнопкой мыши на классе *OrderOptions* диаграммы.
2. В открывшемся меню выберите пункт *Open Specification* (Открыть спецификацию).

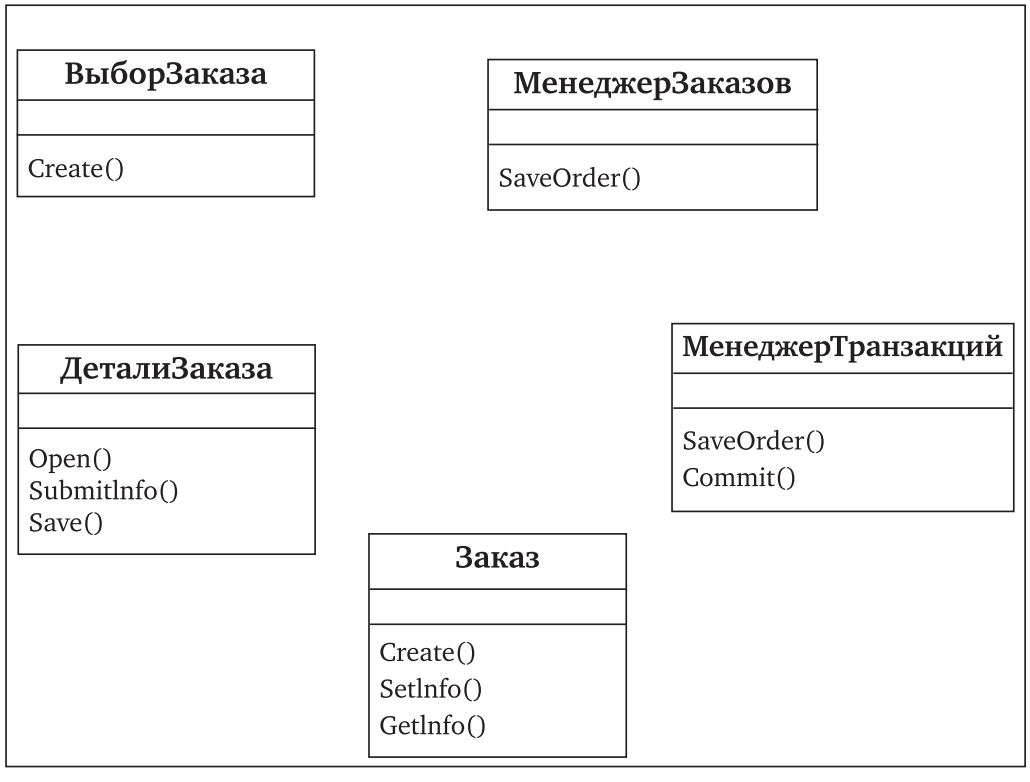

Рис. 3.7. **Диаграмма классов *Add New Order***

3. В поле стереотипа введите слово *Boundary*.
4. Нажмите на кнопку ОК.
5. Щелкните правой кнопкой мыши на классе *OrderDetail* диаграммы.
6. В открывшемся меню выберите пункт *Open Specification* (Открыть спецификацию).
7. В раскрывающемся списке в поле стереотипов теперь будет стереотип *Boundary*. Укажите его.
8. Нажмите на кнопку ОК.
9. Повторите шаги 1—4, связав классы *OrderMgr* и *TransactionMgr* со стереотипом *Control*, а класс *Order* — со стереотипом *Entity*.

Соответствующая диаграмма классов показана на рис. 3.8.

Объединение классов в пакеты:

1. Перетащите в браузере класс *OrderOptions* на пакет *Boundaries*.
2. Перетащите класс *OrderDetail* на пакет *Boundaries*.
3. Перетащите классы *OrderMgr* и *TransactionMgr* на пакет *Control*.
4. Перетащите класс *Order* на пакет *Entities*.

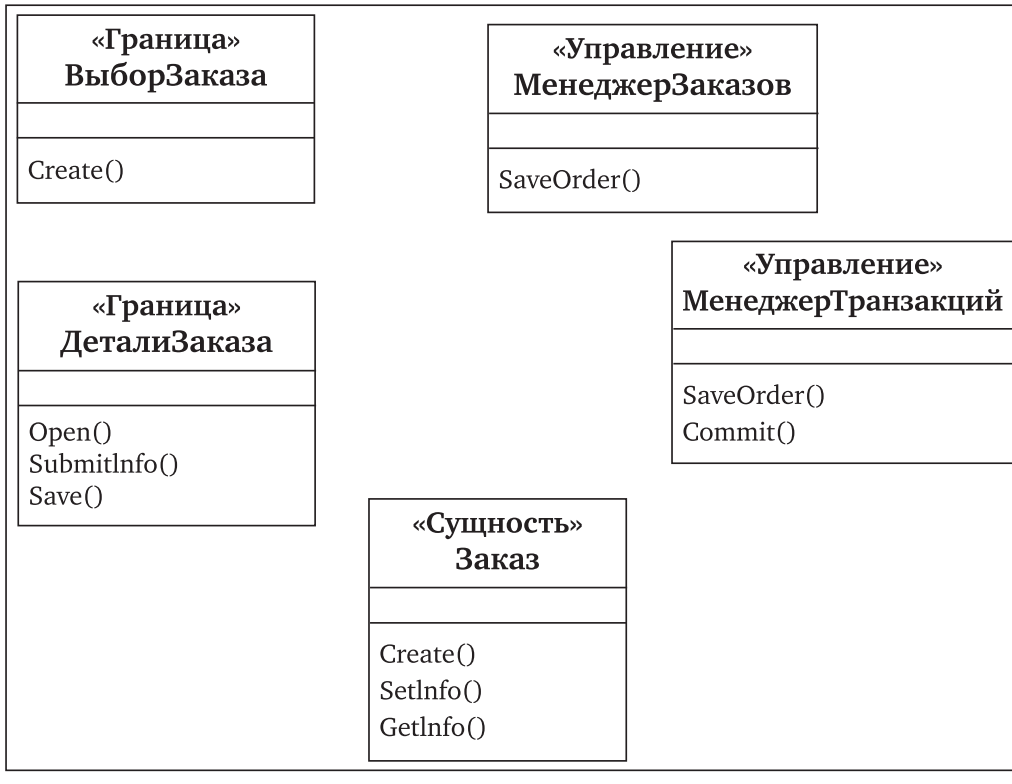

Рис. 3.8. **Стереотипы классов для варианта использования «Ввести новый заказ»**

Добавление диаграмм классов к каждому пакету:

1. Щелкните правой кнопкой на пакете *Boundaries* браузера.
2. В открывшемся меню выберите пункт *New* > *Class Diagram* (Создать > Диаграмму классов).
3. Введите имя новой диаграммы — *Main* (Главная).
4. Дважды щелкните мышью на этой диаграмме, чтобы открыть ее.
5. Перетащите на нее из браузера классы *OrderOptions* и *OrderDetail*.
6. Закройте диаграмму.
7. Щелкните правой кнопкой на пакете *Entities* браузера.
8. В открывшемся меню выберите пункт *New* > *Class Diagram* (Создать > Диаграмму классов).
9. Введите имя новой диаграммы — *Main* (Главная).
10. Дважды щелкните мышью на этой диаграмме, чтобы открыть ее.
11. Перетащите на нее из браузера класс *Order*.
12. Закройте диаграмму.

13. Щелкните правой кнопкой на пакете *Control* браузера.

14. В открывшемся меню выберите пункт *New > Class Diagram* (Создать > Диаграмму классов).

15. Введите имя новой диаграммы — *Main* (Главная).

16. Дважды щелкните мышью на этой диаграмме, чтобы открыть ее.

17. Перетащите на нее из браузера классы *OrderMgr* и *TransactionMgr*.

18. Закройте диаграмму.

Упражнение 4. Добавление атрибутов и операций

Добавим атрибуты и операции к классам диаграммы классов *Add New Order*. Для атрибутов и операций используем специфические для языка особенности. Установим параметры таким образом, чтобы показывать все атрибуты, все операции и их сигнатуры. Видимость покажем с помощью нотации UML.

Выполнение упражнения

Настройка:

1. В меню модели выберите пункт *Tools > Options*.

2. Перейдите на вкладку *Diagram*.

3. Убедитесь, что переключатели *Show Visibility, Show Stereotypes, Show Operation Signatures, Show All Attributes* и *Show All Operations* помечены.

4. Убедитесь, что переключатели *Suppress Attributes* и *Suppress Operations* не помечены.

5. Перейдите на вкладку *Notation*.

6. Убедитесь, что переключатель *Visibility as Icons* не помечен.

Модификация диаграммы последовательности (добавление нового объекта):

1. Найдите в браузере построенную ранее диаграмму последовательности.

2. Щелкните на ней дважды, чтобы ее открыть.

3. Поместите на диаграмму новый объект *OrderItem* (ПозицияЗаказа).

Добавление нового класса:

1. Найдите в браузере диаграмму классов *Add New Order* варианта использования «Ввести новый заказ».

2. Щелкните на ней дважды, чтобы ее открыть.

3. Поместите на диаграмму новый класс *OrderItem* (ПозицияЗаказа).

4. Назначьте этому классу стереотип *Entity*.

5. В браузере перетащите класс в пакет *Entities*.

Добавление атрибутов:

1. Щелкните правой кнопкой мыши на классе *Order* (Заказ).

2. В открывшемся меню выберите пункт *New Attribute* (Создать атрибут).

3. Введите новый атрибут *OrderNumber* : *Integer* (НомерЗаказа)

4. Нажмите клавишу *Enter*.

5. Введите следующий атрибут *CustomerName* : *String* (НаименованиеЗаказчика).

6. Повторите этапы 4 и 5, добавив атрибуты *OrderDate* : *Date* (ДатаЗаказа) и *OrderFillDate* : *Date* (ДатаЗаполненияЗаказа).

7. Щелкните правой кнопкой мыши на классе *OrderItem*.

8. В открывшемся меню выберите пункт *New Attribute* (Создать атрибут).

9. Введите новый атрибут *ItemID* : *Integer* (ИдентификаторПредмета).

10. Нажмите клавишу *Enter*.

11. Введите следующий атрибут *ItemDescription* : *String* (ОписаниеПредмета).

Подробное описание операций с помощью диаграммы Классов:

1. Щелкните мышью на классе *Order*, выделив его таким способом.

2. Щелкните на этом классе еще один раз, чтобы переместить курсор внутрь.

3. Отредактируйте операцию *Create*(), чтобы она имела следующий вид: *Create*() : *Boolean*.

4. Отредактируйте операцию *SetInfo*(), чтобы она выглядела следующим образом: *SetInfo* (*OrderNum* : *Integer*, *Customer* : *String*, *OrderDate* : *Date*, *FillDate* : *Date*) : *Boolean*.

5. Отредактируйте операцию *GetInfo*(), приведя ее к виду: *GetInfo*() : *String*.

Подробное описание операций с помощью браузера:

1. Найдите в браузере класс *OrderItem*.

2. Чтобы раскрыть этот класс, щелкните на значке «+» рядом с ним. В браузере появятся его атрибуты и операции.

3. Дважды щелкните на операции *GetInfo*(), чтобы открыть окно ее спецификации.

4. В раскрывающемся списке *Return Type* (тип возвращаемого значения класс) укажите *String*.

5. Щелкните на кнопке ОК, закрыв окно спецификации операции.

6. Дважды щелкните в браузере на операции *SetInfo* класса *OrderItem*, чтобы открыть окно ее спецификации.

7. В раскрывающемся списке *Return Type* укажите *Boolean*.

8. Перейдите на вкладку *Detail* (Подробно).

9. Щелкните правой кнопкой мыши на белом поле в области аргументов, чтобы добавить туда новый параметр.

10. В открывшемся меню выберите пункт *Insert*. *Rational Rose* добавит туда аргумент под названием *argname*.

11. Щелкните один раз на этом слове, чтобы выделить его, и измените имя аргумента на *ID*.

12. Щелкните на колонке *Type*, открыв раскрывающийся список типов. В нем выберите тип *Integer*.

13. Щелкните на колонке *Default*, чтобы добавить значение аргумента по умолчанию. Введите туда число 0.

14. Нажмите на кнопку ОК, закрыв окно спецификации операции.

15. Дважды щелкните на операции *Create*() класса *OrderItem*, чтобы открыть окно ее спецификации.

16. В раскрывающемся списке *Return Type* укажите *Boolean*.

17. Нажмите на кнопку ОК, закрыв окно спецификации операции.

Подробное описание операций с помощью любого из описанных методов:

1. Используя браузер или диаграмму Классов, введите следующую сигнатуру операций класса *OrderDetail*: *Open*() : *Boolean SubmitInfo*() : *Boolean Save*() : *Boolean*.

2. Используя браузер или диаграмму Классов, введите следующую сигнатуру операций класса *OrderOptions*: *Create*() : *Boolean*.

3. Используя браузер или диаграмму Классов, введите следующую сигнатуру операций класса *OrderMgr*: *SaveOrder* (*OrderID* : *Integer*) : *Boolean*.

4. Используя браузер или диаграмму Классов, введите следующую сигнатуру операций класса *TransactionMgr*: *SaveOrder* (*OrderID* : *Integer*) : *Boolean Commit*() : *Integer*.

Упражнение 5. Добавление связей

Добавим связи к классам, принимающим участие в варианте использования «Ввести новый заказ».

Выполнение упражнения

Настройка:

1. Найдите в браузере диаграмму классов *Add New Order*.
2. Дважды щелкните на ней, чтобы открыть ее.
3. Проверьте, имеется ли на панели инструментов диаграммы кнопка *Unidirectional Association*. Если ее нет, продолжайте настройку, выполнив шаги 4 и 5. Если есть, приступайте к выполнению самого упражнения.
4. Щелкните правой кнопкой мыши на панели инструментов диаграммы и в открывшемся меню выберите пункт *Customize*.
5. Добавьте на панель кнопку, называющуюся *Create A Unidirectional Association*.

Добавление ассоциаций:

1. Нажмите кнопку панели инструментов *Unidirectional Association*.
2. Нарисуйте ассоциацию от класса ВыборЗаказа (*OrderOptions*) к классу ДеталиЗаказа (*OrderDetail*).
3. Повторите этапы 1 и 2, создав еще ассоциации:

От класса *OrderDetail* к классу МенеджерЗаказов (*OrderMgr*).

От класса *OrderMgr* к классу Заказ (*Order*).

От класса *OrderMgr* к классу МенеджерТранзакций (*TransactionMgr*).

От класса *TransactionMgr* к классу *Order.*

От класса *TransactionMgr* к классу ПозицияЗаказа (*OrderItem*).

От класса *Order* к классу *OrderItem*.

4. Щелкните правой кнопкой мыши на однонаправленной ассоциации между классами *OrderOptions* и *OrderDetail* со стороны класса *OrderOptions*,
5. В открывшемся меню выберите пункт *Multiplicity > Zero or One*.
6. Щелкните правой кнопкой мыши на другом конце однонаправленной ассоциации.
7. В открывшемся меню выберите пункт *Multiplicity > Zero or One*.

8. Повторите шаги 4—7, добавив на диаграмму значения множественности для остальных ассоциаций, как показано на рис. 3.9.

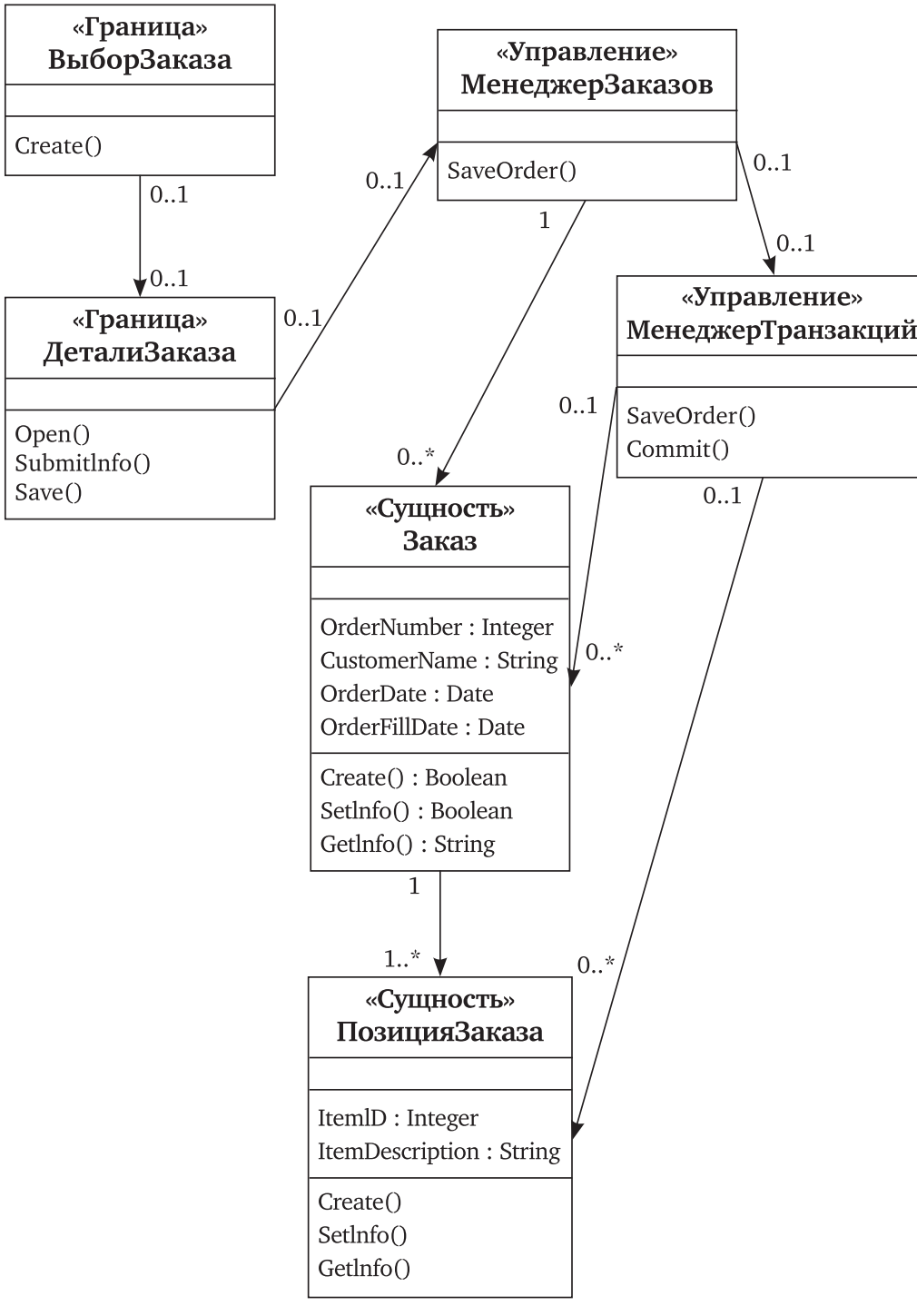

Рис. 3.9. **Ассоциации варианта использования «Ввести новый заказ»**

Упражнение 6. Создание диаграммы состояний

Разработайте диаграмму состояний для класса *Order* (рис. 3.10).

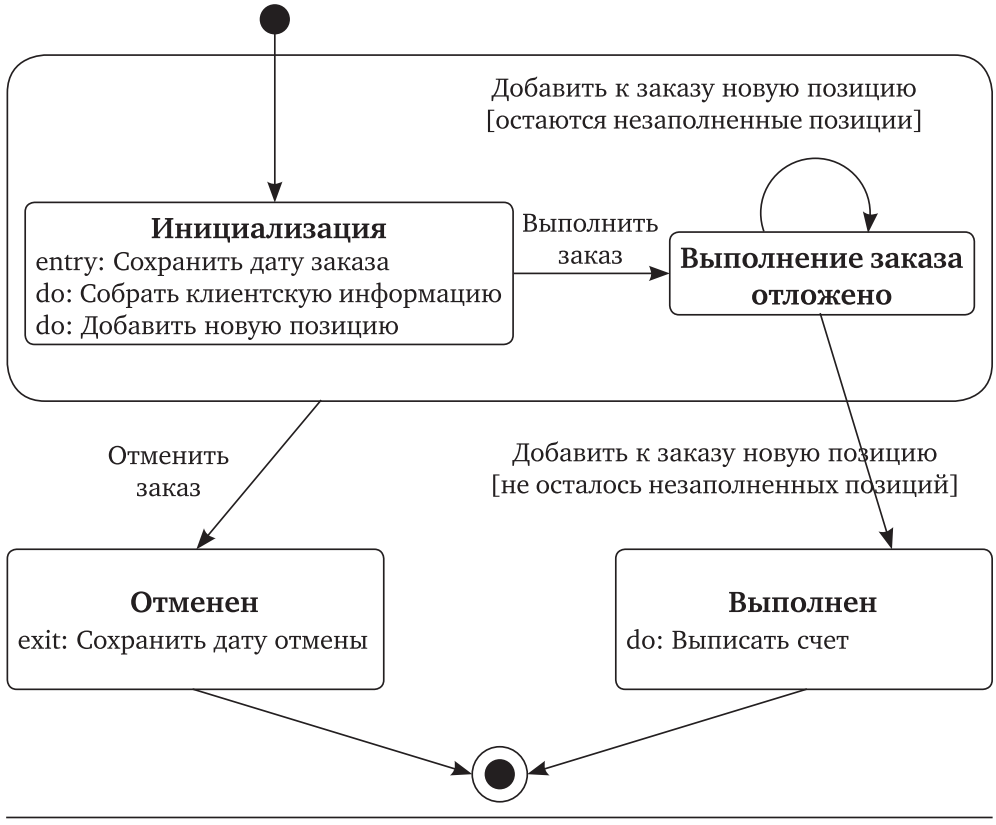

Рис. 3.10. **Диаграмма состояний для класса *Order***

Выполнение упражнения

Создание диаграммы:

1. Найдите в браузере класс *Order*.
2. Щелкните на классе правой кнопкой мыши и в открывшемся меню укажите пункт *New > Statechart Diagram*.

Добавление начального и конечного состояний:

1. На панели инструментов нажмите кнопку *Start State* (Начальное состояние).
2. Поместите это состояние на диаграмму.
3. На панели инструментов нажмите кнопку *End State* (Конечное состояние).
4. Поместите это состояние на диаграмму.

Добавление суперсостояния:

1. На панели инструментов нажмите кнопку *State* (Состояние).

2. Поместите это состояние на диаграмму.

Добавление оставшихся состояний:

1. На панели инструментов нажмите кнопку *State* (Состояние).
2. Поместите это состояние на диаграмму.
3. Назовите состояние *Cancelled* (Отменен).
4. На панели инструментов нажмите кнопку *State* (Состояние).
5. Поместите это состояние на диаграмму.
6. Назовите состояние *Filled* (Выполнен).
7. На панели инструментов нажмите кнопку *State* (Состояние).
8. Поместите это состояние на диаграмму внутрь суперсостояния.
9. Назовите состояние *Initialization* (Инициализация).
10. На панели инструментов нажмите кнопку *State* (Состояние).
11. Поместите это состояние на диаграмму внутрь суперсостояния.
12. Назовите состояние *Pending* (Выполнение заказа приостановлено).

Подробное описание состояний:

1. Дважды щелкните на состоянии *Initialization* (Инициализация).
2. Щелкните правой кнопкой мыши на окне *Actions* (Действия).
3. В открывшемся меню выберите пункт *Insert* (Вставить).
4. Дважды щелкните мышью на новом действии.
5. Назовите его *Store Order Date* (Сохранить дату заказа).
6. Убедитесь, что в окне *When* (Когда) указан пункт *On Entry* (На входе).
7. Повторите шаги 3—7, добавив следующие действия:

Collect Customer Info (Собрать клиентскую информацию), в окне *When* указать пункт *Do*.

Add Order Items (Добавить к заказу новые графы), в окне *When* указать *Do*.

8. Нажмите на кнопки ОК два раза, чтобы закрыть спецификацию.
9. Дважды щелкните на состоянии *Cancelled* (Отменен).
10. Повторите шаги 2—7, добавив действие

Store Cancellation Data (Сохранить дату отмены), указать пункт *On Exit* (на выходе).

11. Нажмите на кнопки OK два раза, чтобы закрыть спецификацию.

12. Дважды щелкните на состоянии *Filled* (Выполнен).

13. Повторите шаги 2—7, добавив действие

Bill Customer (Выписать счет), указать пункт *Do*.

14. Нажмите на кнопки OK два раза, чтобы закрыть спецификацию.

Добавление переходов:

1. На панели инструментов нажмите кнопку *Transition* (Переход).

2. Щелкните мышью на начальном состоянии.

3. Проведите линию перехода к состоянию *Initialization* (Инициализация).

4. Повторите этапы с первого по третий, создав следующие переходы:

От состояния *Initialization* (Инициализация) к состоянию *Pending* (Выполнение заказа приостановлено).

От состояния *Pending* (Выполнение заказа приостановлено) к состоянию *Filled* (Выполнен).

От суперсостояния к состоянию *Cancelled* (Отменен).

От состояния *Cancelled* (Отменен) к конечному состоянию.

От состояния *Filled* (Выполнен) к конечному состоянию.

5. На панели инструментов нажмите кнопку *Transition to Self* (Переход к себе).

6. Щелкните на состоянии *Pending* (Выполнение заказа приостановлено).

Подробное описание переходов:

1. Дважды щелкните на переходе от состояния *Initialization* (Инициализация) к состоянию *Pending* (Выполнение заказа приостановлено), открыв окно его спецификации.

2. В поле *Event* (Событие) введите фразу *Finalize order* (Выполнить заказ).

3. Щелкните на кнопке OK, закрыв окно спецификации.

4. Повторите этапы с первого по третий, добавив событие *Cancel Order* (Отменить заказ) к переходу между суперсостоянием и состоянием *Cancelled* (Отменен).

5. Дважды щелкните на переходе от состояния *Pending* (Выполнение заказа приостановлено) к состоянию *Filled* (Выполнен), открыв окно его спецификации.

6. В поле *Event* (Событие) введите фразу *Add Order Item* (Добавить к заказу новую позицию).

7. Перейдите на вкладку *Detail* (Подробно).

8. В поле *Condition* (Условие) введите *No unfilled items remaining* (Не осталось незаполненных позиций).

9. Щелкните на кнопке OK, закрыв окно спецификации.

10. Дважды щелкните мышью на рефлексивном переходе (*Transition to Self*) состояния *Pending* (Выполнение заказа приостановлено).

11. В поле *Event* (Событие) введите фразу *Add Order Item* (Добавить к заказу новую позицию).

12. Перейдите на вкладку *Detail* (Подробно).

13. В поле *Condition* (Условие) введите *Unfilled items remaining* (Остаются незаполненные позиции).

14. Щелкните на кнопке OK, закрыв окно спецификации.

Упражнение 7. Создание диаграмм компонентов системы

На данный момент уже определены все классы, требуемые для варианта использования «Ввести новый заказ». По мере реализации других вариантов использования на диаграмму следует добавлять новые компоненты.

Завершив анализ и проектирование системы, выберем в качестве языка программирования C++ и для каждого класса создадим соответствующие этому языку компоненты.

На рис. 3.11 показана главная диаграмма компонентов всей системы. Внимание на ней уделяется пакетам создаваемых компонентов.

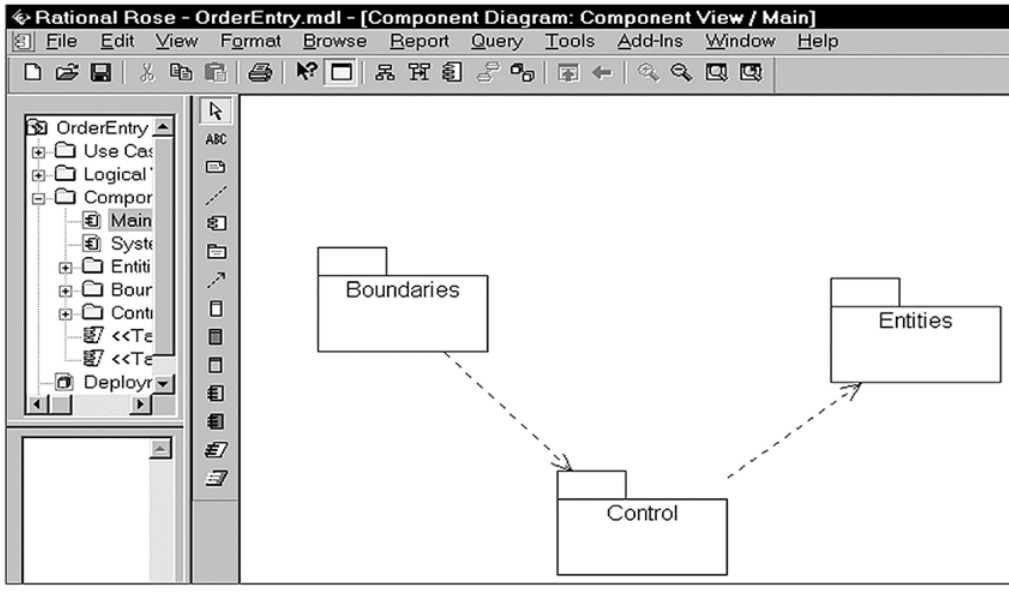

Рис. 3.11. **Главная диаграмма компонентов системы**

На рис. 3.12 изображены все компоненты пакета *Entities*. Эти компоненты содержат классы пакета *Entities* логического представления системы.

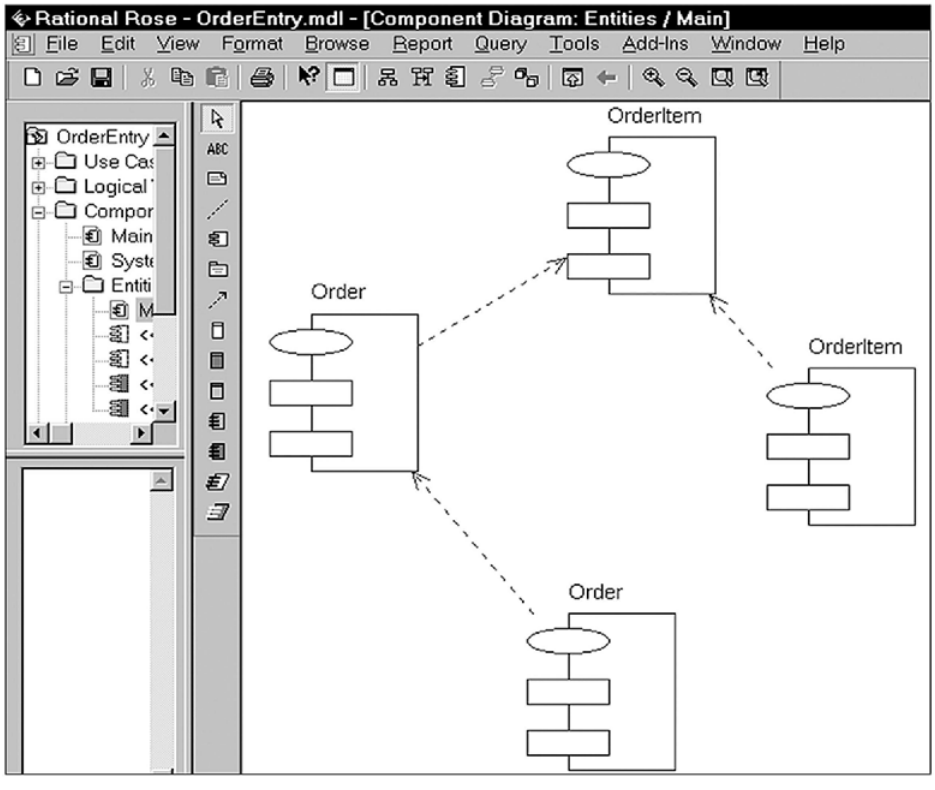

Рис. 3.12. **Диаграмма компонентов пакета *Entities***

На рис. 3.13 показаны компоненты пакета *Control*. Они содержат классы пакета *Control* логического представления системы.

На рис. 3.14 показаны компоненты пакета *Boundaries*. Они также соответствуют классам одноименного пакета логического представления системы.

На рис. 3.15 показаны все компоненты системы. Назовем эту диаграмму диаграммой компонентов системы. На ней можно видеть все зависимости между всеми компонентами проектируемой системы.

Выполнение упражнения

Создание пакетов компонентов:

1. Щелкните правой кнопкой мыши на представлении компонентов в браузере.
2. В открывшемся меню выберите пункт *New* > *Package* (Создать > пакет).
3. Назовите этот пакет *Entities* (Сущности).

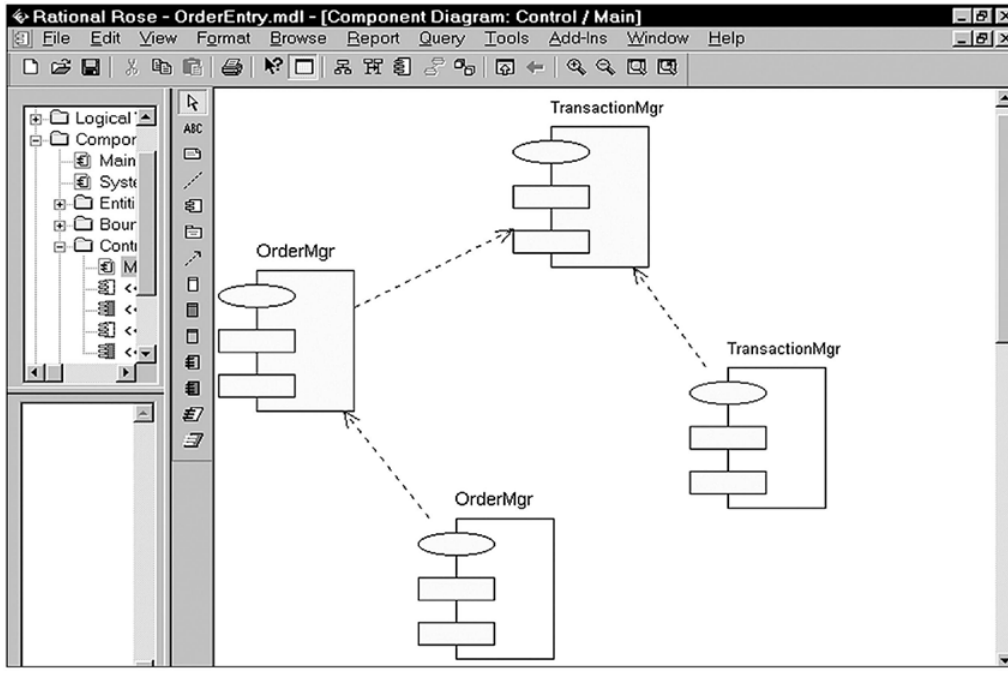

Рис. 3.13. **Диаграмма компонентов пакета *Control***

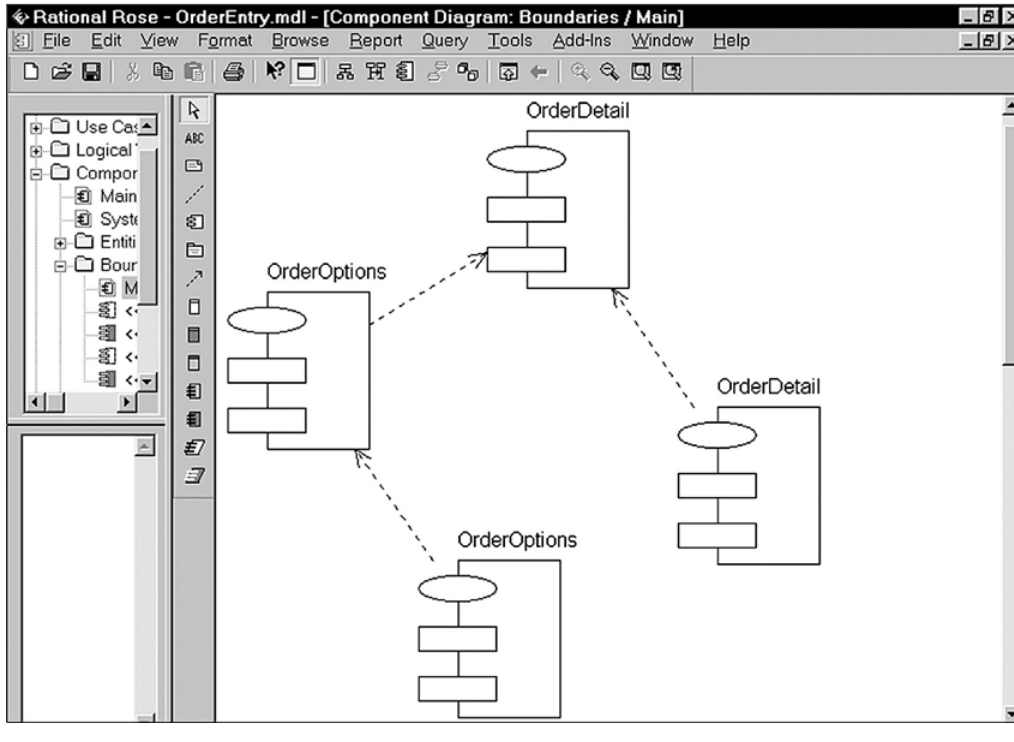

Рис. 3.14. **Диаграмма компонентов пакета *Boundaries***

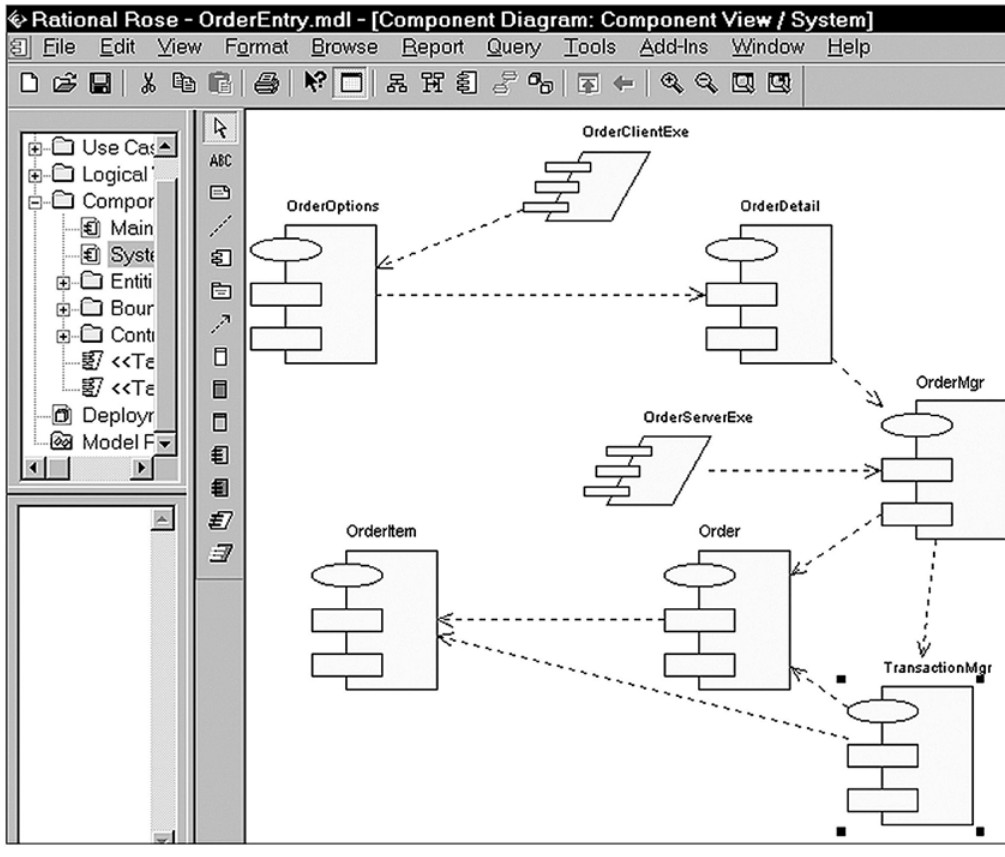

Рис. 3.15. **Диаграмма компонентов системы**

4. Повторите этапы с первого по третий, создав пакеты *Boundaries* (Границы) и *Control* (Управление).

Добавление пакетов на главную диаграмму компонентов:

1. Откройте главную диаграмму компонентов, дважды щелкнув на ней.

2. Перетащите пакеты *Entities, Boundary* и *Control* из браузера на главную диаграмму.

Рисование зависимостей между пакетами:

1. На панели инструментов нажмите кнопку *Dependency* (Зависимость).

2. Щелкните мышью на упаковке *Boundaries* Главной диаграммы компонентов.

3. Проведите линию зависимости до упаковки *Control*.

4. Повторите шаги 1—3, проведя еще линию зависимости от пакета *Control* до пакета *Entities*.

Добавление компонентов к пакетам и рисование зависимостей:

1. Дважды щелкните мышью на пакете *Entities* главной диаграммы компонентов, открыв главную диаграмму компонентов этого пакета.

2. На панели инструментов нажмите кнопку *Package Specification* (Спецификация пакета).

3. Поместите спецификацию пакета на диаграмму.

4. Введите имя спецификации пакета *OrderItem*.

5. Повторите шаги 2—4, добавив спецификацию пакета *Order*.

6. На панели инструментов нажмите кнопку *Package Body* (Тело пакета).

7. Поместите его на диаграмму.

8. Введите имя тела пакета *OrderItem*.

9. Повторите шаги 6—8, добавив тело пакета *Order*.

10. На панели инструментов нажмите кнопку *Dependency* (Зависимость).

11. Щелкните мышью на теле пакета *OrderItem*.

12. Проведите линию зависимости от него к спецификации пакета *OrderItem*.

13. Повторите этапы 10—12, добавив линию зависимости между телом пакета *Order* и спецификацией пакета *Order*.

14. Повторите этапы 10—12, добавив линию зависимости от спецификации пакета *Order* к спецификации пакета *OrderItem*.

15. С помощью описанного метода создайте следующие компоненты и зависимости:

Для пакета *Boundaries*:

Спецификацию пакета *OrderOptions.*

Тело пакета *OrderOptions*.

Спецификацию пакета *OrderDetail*.

Тело пакета *OrderDetail*.

Зависимости в пакете *Boundaries*:

От тела пакета *OrderOptions* до спецификации пакета *OrderOptions*.

От тела пакета *OrderDetail* до спецификации пакета *OrderDetail*.

От спецификации пакета *OrderOptions* до спецификации пакета *OrderDetail*.

Для пакета *Control*:

Спецификацию пакета *OrderMgr*.

Тело пакета *OrderMgr*.

Спецификацию пакета *TransactionMgr*.

Тело пакета *TransactionMgr*.

Зависимости в пакете *Control*:

От тела пакета *OrderMgr* до спецификации пакета *OrderMgr*.

От тела пакета *TransactionMgr* до спецификации пакета *TransactionMgr*.

От спецификации пакета *OrderMgr* до спецификации пакета *TransactionMgr*.

Создание диаграммы компонентов системы:

1. Щелкните правой кнопкой мыши на представлении компонентов в браузере.
2. В открывшемся меню выберите пункт *New > Component Diagram*.
3. Назовите новую диаграмму *System*.
4. Дважды щелкните на этой диаграмме.

Размещение компонентов на диаграмме компонентов системы:

1. Если это еще не было сделано, разверните в браузере пакет компонентов *Entities*, чтобы открыть его.
2. Щелкните мышью на спецификации пакета *Order* в пакете компонентов *Entities*.
3. Перетащите эту спецификацию на диаграмму.
4. Повторите этапы 2 и 3, поместив на диаграмму спецификацию пакета *OrderItem*.
5. С помощью этого метода поместите на диаграмму следующие компоненты:

Из пакета компонентов *Boundaries*:

Спецификацию пакета *OrderOptions*.

Спецификацию пакета *OrderDetail*.

Из пакета компонентов *Control*:

Спецификацию пакета *OrderMgr*.

Спецификацию пакета *TransactionMgr*.

6. На панели инструментов нажмите кнопку *Task Specification* (Спецификация задачи).
7. Поместите спецификацию задачи на диаграмму и назовите ее *OrderClientExe*.
8. Повторите этапы 6 и 7 для спецификации задачи *OrderServerExe*.

Добавление оставшихся зависимостей на диаграмму компонентов системы:

Уже существующие зависимости будут автоматически показаны на диаграмме компонентов системы после добавления

туда соответствующих компонентов. Теперь надо добавить остальные зависимости.

1. На панели инструментов нажмите кнопку *Dependency* (Зависимость).
2. Щелкните на спецификации пакета *OrderDetail*.
3. Проведите линию зависимости к спецификации пакета *OrderMgr*.
4. Повторите этапы 1—3, создав следующие зависимости:

От спецификации пакета *OrderMgr* к спецификации пакета *Order*.
От спецификации пакета *TransactionMgr* к спецификации пакета *OrderItem*.
От спецификации пакета *TransactionMgr* к спецификации пакета *Order*.
От спецификации задачи *OrderClientExe* к спецификации пакета *OrderOptions*.
От спецификации задачи *OrderServerExe* к спецификации пакета *OrderMgr*.

Соотнесение классов с компонентами:

1. В логическом представлении браузера найдите класс *Order* пакета *Entities*.
2. Перетащите этот класс на спецификацию пакета компонента *Order* в представлении компонентов браузера. В результате класс *Order* будет соотнесен со спецификацией пакета компонента *Order*.
3. Перетащите класс *Order* на тело пакета компонента *Order* в представлении компонентов браузера. В результате класс *Order* будет соотнесен с телом пакета компонента *Order*.
4. Повторите этапы 1—3, соотнеся с классами следующие компоненты:

Класс *OrderItem* со спецификацией пакета *OrderItem*.
Класс *OrderItem* с телом пакета *OrderItem*.
Класс *OrderOptions* со спецификацией пакета *OrderOptions*.
Класс *OrderOptions* с телом пакета *OrderOptions*.
Класс *OrderDetail* со спецификацией пакета *OrderDetail*.
Класс *OrderDetail* с телом пакета *OrderDetail*.
Класс *OrderMgr* со спецификацией пакета *OrderMgr*.
Класс *OrderMgr* с телом пакета *OrderMgr*.
Класс *TransactionMgr* со спецификацией пакета *TransactionMgr*.
Класс *TransactionMgr* с телом пакета *TransactionMgr*.

Упражнение 8. Создание диаграммы размещения

Разработайте диаграмму размещения для системы обработки заказов (рис. 3.16).

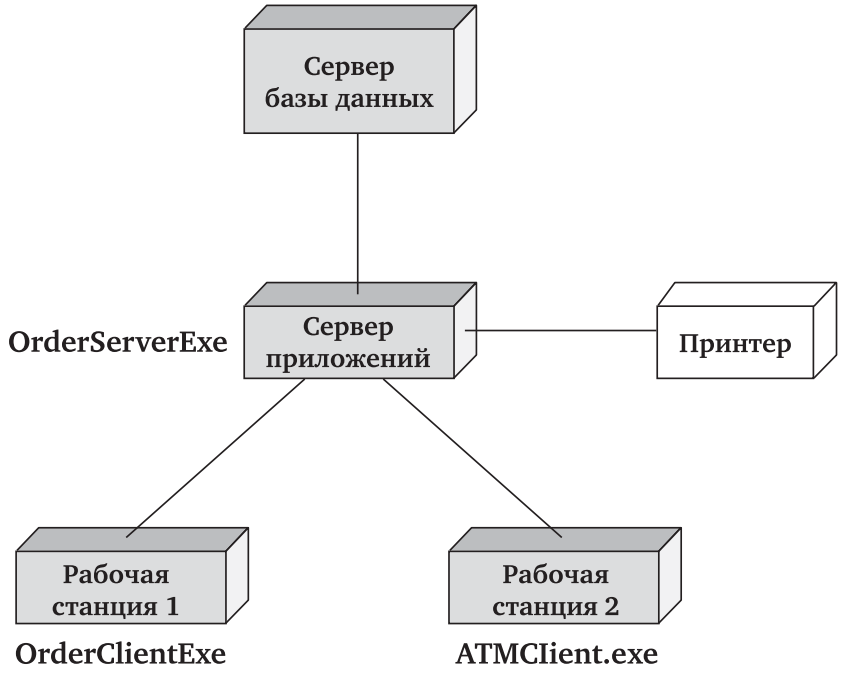

Рис. 3.16. **Диаграмма размещения для системы обработки заказов**

Выполнение упражнения

Добавление узлов к диаграмме размещения:

1. Дважды щелкните мышью на представлении размещения в браузере, чтобы открыть диаграмму размещения.
2. На панели инструментов нажмите кнопку *Processor* (Процессор).
3. Щелкните на диаграмме, поместив туда процессор.
4. Введите имя процессора «Сервер базы данных».
5. Повторите шаги 2—4, добавив следующие процессоры:

Сервер приложения.
Клиентская рабочая станция № 1.
Клиентская рабочая станция № 2.

6. На панели инструментов нажмите кнопку *Device* (Устройство).
7. Щелкните на диаграмме, поместив на нее устройство.
8. Назовите его «Принтер».

Добавление связей:

1. На панели инструментов нажмите кнопку *Connection* (Связь).
2. Щелкните на процессоре «Сервер базы данных».
3. Проведите линию связи к процессору «Сервер приложения».
4. Повторите этапы 1—3, добавив следующие связи:

От процессора «Сервер приложения» к процессору «Клиентская рабочая станция № 1».

От процессора «Сервер приложения» к процессору «Клиентская рабочая станция № 2».

От процессора «Сервер приложения» к устройству «Принтер».

Добавление процессов:

1. Щелкните правой кнопкой мыши на процессоре «Сервер приложения» в браузере.
2. В открывшемся меню выберите пункт *New > Process* (Создать > Процесс).
3. Введите имя процесса *OrderServerExe*.
4. Повторите этапы 1—3, добавив еще процессы:

На процессоре «Клиентская рабочая станция № 1» — процесс *OrderClientExe*.

На процессоре «Клиентская рабочая станция № 2» — процесс *ATMClientEXE*.

Показ процессов на диаграмме:

1. Щелкните правой кнопкой мыши на процессоре «Сервер приложения».
2. В открывшемся меню выберите пункт *Show Processes* (Показать процессы).
3. Повторите этапы 1 и 2, показав процессы на следующих процессорах:

Клиентская рабочая станция № 1.

Клиентская рабочая станция № 2.

Упражнение 9. Генерация кода C++

В предыдущих упражнениях была создана модель для системы обработки заказов (*Order Entry*). Теперь сгенерируем код C++ для этой системы. При этом воспользуемся диаграммой компонентов системы (рис. 3.17).

Для генерации кода необходимо выполнить описанные ниже шаги.

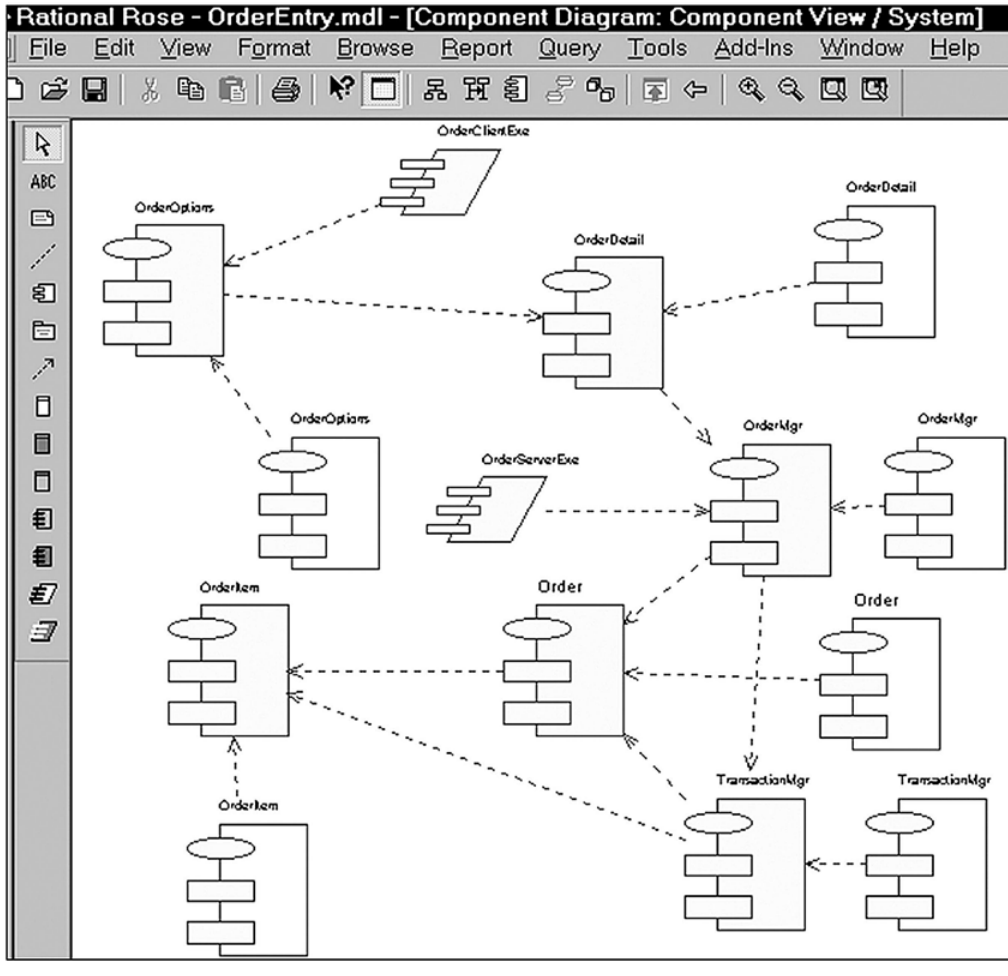

Рис. 3.17. **Диаграмма компонентов системы *Order Entry***

Выполнение упражнения

Ввод тел пакетов на диаграмму компонентов системы:

1. Откройте диаграмму компонентов системы.
2. Выберите в браузере *Entities*: тело пакета *Order*.
3. «Перетащите» тело пакета *Order* на диаграмму компонентов системы.
4. Повторите шаги 2 и 3 для следующих компонентов:
   Entities: тело пакета *OrderItem*.
   Boundaries: тело пакета *OrderOptions*.
   Boundaries: тело пакета *OrderDetail*.
   Control: тело пакета *TransactionMgr*.
   Control: тело пакета *OrderMgr*.

Установка языка C++:

1. Откройте спецификацию компонента *Order* (спецификацию пакета) в пакете компонентов *Entities*.

2. Выберите в качестве языка C++.
3. Повторите шаги 1 и 2 для следующих компонентов:
   Entities: тело пакета *Order*.
   Entities: спецификация пакета *OrderItem*.
   Entities: тело пакета *OrderItem*.
   Boundaries: спецификация пакета *OrderOptions*.
   Boundaries: тело пакета *OrderOptions*.
   Boundaries: спецификация пакета *OrderDetail*.
   Boundaries: тело пакета *OrderDetail*.
   Control: спецификация пакета *TransactionMgr*.
   Control: тело пакета *TransactionMgr*.
   Control: спецификация пакета *OrderMgr*.
   Control: тело пакета *OrderMgr*.
   Спецификация задачи *OrderClientExe*.
   Спецификация задачи *OrderServerExe*.

Генерация кода C++:

1. Откройте диаграмму компонентов системы.
2. Выберите все объекты на диаграмме компонентов системы.
3. Выберите *Tools* > C++ > *Code Generation* в меню.
4. Выберите каталог для генерации кода.
5. Щелкните по кнопке OK и выполните генерацию в окне генерации кода.
6. Просмотрите результаты генерации (меню *Tools* > C++ > *Browse Header* и *Tools* > C++ > *Browse Body*).

Библиографический список

1. *Басс, Л.* Архитектура программного обеспечения на практике / Л. Басс, П. Клементс, Р. Кацман ; пер. с англ. — СПб. : Питер, 2006. — 575 с.
2. *Боггс, У.* UML и Rational Rose 2002 / У. Боггс, М. Боггс ; пер. с англ. — М. : Изд. «Лори». 2004. — 510 с.
3. *Буч, Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / Г. Буч ; пер. с англ. — М. : Бином; СПб. : Невский диалект, 1999. — 320 с.
4. *Вендров, А. М.* Проектирование программного обеспечения экономических информационных систем / А. М. Вендров. — М. : Финансы и статистика, 2005. — 544 с.
5. *Кватрани, Т.* Визуальное моделирование с помощью Rational Rose 2002 и UML / Т. Кватрани ; пер. с англ. — М. : Издательский дом «Вильямс», 2003. — 192 с.
6. *Кролл, П.* Rational Unified Process — это легко. Руководство по RUP / П. Кролл, Ф. Крачтен ; пер. с англ. — М. : КУДИЦ-ОБРАЗ, 2004. — 432 с.
7. *Леоненков, А. В.* Самоучитель UML / А. В. Леоненков. — СПб. : БВХ — Петербург, 2004. — 432 с.
8. *Мацяшек, Л.* Анализ требовании и проектирование систем. Разработка информационных систем с использованием UML / Л. Мацяшек ; пер. с англ. — М. : Издательский дом «Вильямс», 2002. — 432 с.
9. *Рамбо, Дж.* UML: специальный справочник / Дж. Рамбо ; пер. с англ. — СПб. : Питер, 2002. — 656 с.
10. *Соммервилл, И.* Инженерия программного обеспечения / И. Соммервилл ; пер. с англ. — М. : Изд. дом «Вильямс», 2002. — 624 с.

Заключение

Программная инженерия — это область деятельности, основанная на интеграции принципов математики, информатики и компьютерных наук с инженерными подходами. В то же время это дисциплина, изучающая применение строгого систематического инженерного подхода к разработке, эксплуатации и сопровождению программного обеспечения.

Основа объектной технологии разработки программных проектов — процесс, который должен четко определять и структурировать технологию инженерии программного обеспечения. Методология объектно-ориентированного анализа и проектирования тесно связана с концепцией автоматизированной разработки программного обеспечения (*Computer Aided Software Engineering* — CASE) и визуального моделирования программных систем.

В соответствии с современными принципами программной инженерии при разработке систем должны реализовываться следующие положения:

- итеративная разработка;
- управление требованиями;
- модульная архитектура системы;
- визуальное моделирование;
- непрерывная качественная оценка продукта.

Следует подчеркнуть, что согласно этим положениям современные процессы разработки программных систем должны быть итеративными с пошаговым наращиванием возможностей системы.

При таких процессах модели разрабатываемых систем уточняются и преобразуются на всех этапах. В результате успешных итераций добавляются новые детали, при необходимости вводятся изменения и усовершенствования. Выпуски программных модулей с наращенными возможностями обеспечивают обратную связь с пользователями, необходимую для продолжения разработки программных систем.

Программно-инструментальная триада инженерного подхода к разработке программных систем — это процесс разработки, CASE-средства, язык визуального моделирования.

Визуальное моделирование как составляющая проектирования программных систем может быть использовано во всех фазах проекта, а именно: начальной фазе (анализ и определение требований), фазе проектирования, фазе конструирования, фазе ввода в действие программного продукта.

В начальной фазе некоторые задачи включают в себя определение вариантов использования и действующих лиц. *IBM Rational Rose* можно применять для документирования вариантов использования и действующих лиц, а также для создания диаграмм, показывающих связи между ними.

В фазе проектирования осуществляется уточнение и детализация требований к системе. UML-диаграммы последовательности и кооперации позволяют проиллюстрировать поток обработки данных при его детализации. Уточнение предполагает подготовку проекта системы для передачи разработчикам, которые начнут ее конструирование. В среде *IBM Rational Rose* это выполняется путем создания UML-диаграмм классов и UML-диаграмм состояния.

В фазе конструирования пишется большая часть кода проекта. Чтобы показать зависимости между компонентами на этапе компиляции, создаются UML-диаграммы компонентов. После выбора языка программирования можно осуществить генерацию скелетного кода для каждого компонента. По завершении работы над кодом модель можно привести в соответствие с ним с помощью обратного проектирования.

В фазе ввода в действие программного продукта осуществляется обновление моделей в *Rational Rose*, вносятся изменения в UML-диаграммы компонентов и размещения.

Визуальные модели используют не только разработчики. Они применяются также в других ситуациях:

- с помощью диаграмм вариантов использования потребители и менеджеры проекта получают общее представление о системе и могут принять решение о сфере ее применения;
- с помощью диаграмм вариантов использования и документации менеджеры проекта могут разделить проект на отдельные управляемые задачи;
- из документации по вариантам использования аналитики и потребители могут понять, что будет делать готовая система;

- изучив документацию, технические писатели могут приступить к написанию руководства для пользователей и к подготовке планов по их обучению;
- из диаграмм последовательности и кооперации аналитики и разработчики определяют, насколько логично работает система, понимают ее объекты и сообщения между ними;
- с помощью документации по вариантам использования, а также диаграмм последовательности и кооперации специалисты по контролю качества могут получить информацию, требующуюся им для написания тестовых сценариев;
- с помощью диаграмм классов и состояний разработчики получают представление о фрагментах системы и их взаимодействии друг с другом;
- из диаграмм компонентов и размещения эксплуатационный персонал может узнать, какие пусковые файлы и другие компоненты будут созданы, а также где в сети они должны быть размещены;
- с помощью модели в целом вся команда участников проекта может отслеживать реализацию исходных требований до кода.

Модели, созданные в *IBM Rational Rose*, весьма выгодно иметь и для уже существующих приложений. Если сделано изменение в модели, *IBM Rational Rose* позволяет модифицировать код для его реализации. Если же был изменен код, можно автоматически обновить соответствующим образом и модель. Благодаря этому удается поддерживать соответствие между моделью и кодом, уменьшая риск «старения» модели.

Таким образом, визуальное моделирование с использованием CASE-технологии позволяет осуществлять анализ и проектирование программных систем до написания кода. С помощью готовой модели выявляются недостатки проекта на ранних стадиях разработки, что позволяет минимизировать затраты на их исправление и, соответственно, снизить общую стоимость проекта.

Рекомендуемая литература

1. *Вендров, А. М.* Практикум по проектированию программного обеспечения экономических информационных систем : учеб. пособие / А. М. Вендров. — М. : Финансы и статистика, 2006.

2. *Грекул, В. И.* Проектирование информационных систем: учебник и практикум для академического бакалавриата / В. И. Грекул, Н. Л. Коровкина, Г. А. Левочкина. — М. : Издательство Юрайт, 2017.

3. *Зыков, С. В.* Программирование. Объектно-ориентированный подход: учебник и практикум для академического бакалавриата / С. В. Зыков. — М. : Издательство Юрайт, 2017.

4. *Йордан, Э.* Объектно-ориентированный анализ и проектирование систем : пер. с англ. / Э. Йордан, К. Аргила. — М. : Лори, 2014.

5. *Лаврищева, Е. М.* Программная инженерия. Парадигмы, технологии и CASE-средства : учебник для вузов / Е. М. Лаврищева. — 2-е изд., испр. — М. : Издательство Юрайт, 2018.

6. *Ларман, К.* Применение UML 2.0 и шаблонов проектирования : пер. с англ. / К. Ларман. — 3-е изд. — М. : Вильямс, 2016.

7. *Макконнелл, С.* Профессиональная разработка программного обеспечения : пер. с англ. / С. Макконнелл. — СПб. : СимволПлюс, 2015.

8. *Мацяшек, Л.* Анализ и проектирование информационных систем с помощью UML 2.0 : пер. с англ. / Л. Мацяшек. — М. : Вильямс, 2016.

9. Проектирование информационных систем : учебник и практикум для академического бакалавриата / Д. В. Чистов, П. П. Мельников, А. В. Золотарюк, Н. Б. Ничепорук ; под общ. ред. Д. В. Чистова. — М. : Издательство Юрайт, 2017.

10. *Тузовский, А. Ф.* Объектно-ориентированное программирование : учеб. пособие для прикладного бакалавриата / А. Ф. Тузовский. — М. : Издательство Юрайт, 2017.

Новые издания по дисциплине «Программная инженерия» и смежным дисциплинам

Альсова, О. К. Имитационное моделирование систем в среде Extendsim : учеб. пособие для академического бакалавриата / О. К. Альсова. — 2-е изд. — М. : Издательство Юрайт, 2018.

Загорулько, Ю. А. Искусственный интеллект. Инженерия знаний : учеб. пособие для вузов / Ю. А. Загорулько, Г. Б. Загорулько. — М. : Издательство Юрайт, 2018.

Лаврищева, Е. М. Программная инженерия и технологии программирования сложных систем : учебник для вузов / Е. М. Лаврищева. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2018.

Лаврищева, Е. М. Технология разработки и моделирования вариантов программных систем : монография для вузов / Е. М. Лаврищева. — М. : Издательство Юрайт, 2017.

Советов, Б. Я. Моделирование систем : учебник для академического бакалавриата / Б. Я. Советов, С. А. Яковлев. — 7-е изд. — М. : Издательство Юрайт, 2019.

Советов, Б. Я. Моделирование систем. Практикум : учеб. пособие для бакалавров / Б. Я. Советов, С. А. Яковлев. — 4-е изд., перераб. и доп. — М. : Издательство Юрайт, 2019.

Наши книги можно приобрести:

Учебным заведениям и библиотекам:
в отделе по работе с вузами
тел.: (495) 744-00-12, e-mail: vuz@urait.ru

Частным лицам:
список магазинов смотрите на сайте urait.ru
в разделе «Частным лицам»

Магазинам и корпоративным клиентам:
в отделе продаж
тел.: (495) 744-00-12, e-mail: sales@urait.ru

Отзывы об издании присылайте в редакцию
e-mail: red@urait.ru

Новые издания и дополнительные материалы доступны
в электронной библиотечной системе «Юрайт»
biblio-online.ru

Учебное издание

Черткова Елена Александровна

ПРОГРАММНАЯ ИНЖЕНЕРИЯ. ВИЗУАЛЬНОЕ МОДЕЛИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

Учебник для СПО

Формат $60\times90\,^{1}/_{16}$.
Гарнитура «Charter». Печать цифровая.
Усл. печ. л. 9,19.

ООО «Издательство Юрайт»
111123, г. Москва, ул. Плеханова, д. 4а.
Тел.: (495) 744-00-12. E-mail: izdat@urait.ru, www.urait.ru