Учебник для вузов

ПИТЕР

СТАНДАРТ ТРЕТЬЕГО ПОКОЛЕНИЯ

С. А. Орлов Б. Я. Цилькер

Технологии разработки программного обеспечения

Современный курс по программной инженерии

4-е издание

ДОПУЩЕНО МИНИСТЕРСТВОМ ОБРАЗОВАНИЯ И НАУКИ РФ

Сергей Александрович Орлов, Борис Яковлевич Цилькер

Технологии разработки программного обеспечения: Учебник для вузов. 4-е издание. Стандарт третьего поколения

Рецензенты:

Филиппович Ю. Н., к. т. н, доцент Московского государственного университета печати;

Ревунков Г. И., к. т. н, доцент Московского государственного технического университета им. Н. Э. Баумана.

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Художественный редактор	*Л. Адуевская*
Корректор	*В. Листова*
Верстка	*Л. Харитонов*

ББК 32.973.2-018-02я7
УДК 004.413(075)

Орлов С. А., Цилькер Б. Я.

О-66 Технологии разработки программного обеспечения: Учебник для вузов. 4-е изд. Стандарт третьего поколения. — СПб.: Питер, 2012. — 608 с.: ил.

ISBN 978-5-459-01101-2

Учебник посвящен систематическому изложению принципов, моделей, методов и метрик, используемых в инженерном цикле разработки сложных программных продуктов. Изложены классические основы программной инженерии, показаны последние научные и практические достижения, характеризующие динамику развития этой области; продемонстрирован комплексный подход к решению наиболее важных вопросов, возникающих в программных проектах.

Допущено Министерством образования и науки Российской Федерации в качестве учебника для студентов высших учебных заведений, обучающихся по специальности «Программное обеспечение вычислительной техники и автоматизированных систем» направлений подготовки дипломированных специалистов «Информатика и вычислительная техника».

ISBN 978-5-459-01101-2

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная
Подписано в печать 01.02.12. Формат 70×100/16. Усл. п. л. 49.020. Тираж 2000. Заказ 27222.
Отпечатано по технологии CtP в ОАО «Первая Образцовая типография», обособленное подразделение «Печатный двор»
197110, Санкт-Петербург, Чкаловский пр., 15.

Краткое оглавление

Оглавление

Предисловие к четвертому изданию

Время бежит чрезвычайно быстро, а в такой динамической области знаний, как программная инженерия, оно спрессовывается в мгновения. Прошло семь лет с момента третьего издания, и вот, уважаемый читатель, вы держите в руках четвертое издание учебника.

Учебная дисциплина «Технологии разработки программного обеспечения» играет особую роль в линейке дисциплин, посвященных различным аспектам программирования, она замыкает эту линейку и интегрирует знания всех дисциплин-предшественниц.

Первое и, пожалуй, ключевое слово в названии дисциплины — технология. Оно подчеркивает аналогию между созданием программного продукта и промышленным производством. Оно отражает современную тенденцию — ввести дисциплину, организацию, инструментирование в такой, казалось бы, свободный творческий процесс, как программирование. Слово фиксирует ту точку зрения, что программирование, несмотря на интеллектуальность и творческий характер этой деятельности, нуждается в организации и регламентации, наборе соглашений и правил, не говоря уж об инструментальном обеспечении. Сейчас это кажется тривиальным утверждением, но в 60-е годы XX века такую точку зрения приходилось отстаивать в жарких спорах и дискуссиях.

Сам русский термин «технология программирования» был введен (в «боях» и «схватках») русским академиком Андреем Петровичем Ершовым. Это было очень давно, в те времена термин «программирование» Андрей Петрович толковал в обобщенном смысле, подразумевая все виды деятельности, выполняемые в ходе создания программных систем. И это был революционный прорыв. На западе для определения этой деятельности использовался (и тогда, и сейчас) термин «engineering», вобравший в себя как технологический, так и созидательный аспект деятельности. С тех пор много воды утекло... С современных позиций обобщенный термин, применимый к созданию программных систем, обозначают как «разработка». Справедлива формула: «разработка = анализ + проектирование + программирование (кодирование) + тестирование + отладка». Иногда сюда же включают и «сопровождение». А чтобы подчеркнуть промышленно-производственный аспект, говорят о «технологии разработки».

Целью дисциплины «Технологии разработки программного обеспечения» является обучение основным принципам и методам, используемым на различных этапах разработки программного обеспечения сложных компьютерных систем, а также обучение организации процессов программной разработки.

В государственном образовательном стандарте высшего профессионального образования содержание дисциплины определено следующим образом:

- программные продукты (изделия);
- жизненный цикл ПО;
- метрология и качество ПО;
- критерии качества: сложность, корректность, надежность, трудоемкость;
- измерения и оценка качества ПО;
- процесс производства ПО: методы, технология и инструментальные средства;
- тестирование и отладка;
- документирование;
- проектирование программного обеспечения;
- технологический цикл разработки программных систем;
- коллективная работа по созданию программ;
- организация процесса разработки и инструментальные средства поддержки;
- автоматизация проектирования программных продуктов;
- принципы построения, структура и технология использования САПР ПО.

Все эти вопросы освещены в данном учебнике, то есть учебник отвечает всем требованиям образовательного стандарта.

Что нового в этом издании?

Во-первых, полностью переработана структура и содержание учебника. Существенно большее внимание уделяется описанию фундаментальных положений программной инженерии, учебник отражает наиболее впечатляющие научные и инженерные новации последних лет. Нам представляется, что новая последовательность изложения материала гармонизирована с реальным порядком изучения дисциплины в высших учебных заведениях. Значительно расширено изложение вопросов менеджмента программных проектов, работы с требованиями, управления изменениями, управления конфигурацией программных изделий, обеспечения качества ПО. Ощутимо выросло количество примеров и упражнений по всему жизненному циклу программной разработки. Достаточно последовательно и подробно рассмотрены разнообразные принципы разработки и аппарат количественной оценки качества программных решений.

Во-вторых, в качестве инструментария для создания всех артефактов визуального моделирования применяется язык нового поколения UML 2.3, а в качестве среды для автоматизации разработки — CASE-система IBM Rational Software Architect.

В-третьих, добавлено описание средств параллельного программирования языка Ada 2005, мощность и выразительность которых вне конкуренции.

В-четвертых, исправлены имевшиеся ошибки и опечатки. Увы, время от времени приходится вспоминать пословицу, что «каждая последняя ошибка является предпоследней...», поэтому авторы будут признательны всем читателям, кто заметит неточности и опечатки и сообщит о них редакции.

В заключение хочется поблагодарить всех читателей, приславших благожелательные отзывы и конструктивные замечания.

Введение

Известно, что основной задачей первых трех десятилетий компьютерной эры являлось развитие аппаратных компьютерных средств. Это было обусловлено высокой стоимостью обработки и хранения данных. В 80-е годы успехи микроэлектроники привели к резкому увеличению производительности компьютера при значительном снижении стоимости.

Основной задачей 90-х годов и начала XXI века стало совершенствование качества компьютерных приложений, возможности которых целиком определяются программным обеспечением (ПО).

Современный персональный компьютер теперь имеет производительность большой ЭВМ 80-х годов. Сняты практически все аппаратные ограничения на решение задач. Оставшиеся ограничения приходятся на долю ПО.

Чрезвычайно актуальными стали следующие проблемы:

- аппаратная сложность опережает наше умение строить ПО, использующее потенциальные возможности аппаратуры;
- наше умение строить новые программы отстает от требований к новым программам;
- нашим возможностям эксплуатировать существующие программы угрожает низкое качество их разработки.

Ключом к решению этих проблем является грамотная организация процесса создания ПО, реализация технологических принципов промышленной разработки программных систем (ПС).

Настоящий учебник посвящен систематическому изложению принципов, моделей и методов (формирования требований, анализа, проектирования и тестирования), используемых в инженерном цикле разработки сложных программных продуктов.

В основу материала положен многолетний опыт постановки и преподавания авторами дисциплин по программной инженерии. Базовый курс «Технология разработки программного обеспечения» прослушали больше двух тысяч студентов, работающих теперь в инфраструктуре мировой программной индустрии самых разных стран и на самых разных континентах.

Авторы стремились к достижению четырех целей:

- изложить классические основы, отражающие накопленный отечественный и мировой опыт программной инженерии;

- показать последние научные и практические достижения, характеризующие динамику развития в области Software Engineering,
- обеспечить комплексный охват наиболее важных вопросов, возникающих в больших и средних программных проектах,
- обобщить и отразить 20-летний университетский опыт преподавания соответствующих дисциплин.

Компьютерные науки вообще и программная инженерия в частности — очень популярные и стремительно развивающиеся области знаний. Обоснование простое: человеческое общество XXI века — информационное общество. Об этом говорят цифры: в ведущих странах занятость населения в информационной сфере составляет 60%, а в сфере материального производства — 40%. Именно поэтому специальности направления «Компьютерные науки и информационные технологии» гарантируют приобретение наиболее престижных, дефицитных и высокооплачиваемых профессий. Так считают во всех развитых странах мира. Ведь не зря утверждают: «Кто владеет информацией — тот владеет миром!»

Поэтому понятно то пристальное внимание, которое уделяет компьютерному образованию мировое сообщество, понятно стремление унифицировать и упорядочить знания, необходимые специалисту этого направления. Одним из результатов такой работы являются международный стандарт по компьютерному образованию Computing Curricula 2005, состоящий из пяти томов (отметим том Computer Science Curriculum 2008, а также том Software Engineering 2004), и международный стандарт по программной инженерии IEEE Computer Society Guide to the Software Engineering Body of Knowledge SWEBOK 2004.

Содержание данного учебника отвечает рекомендациям этих стандартов. Учебник состоит из 19 глав и трех приложений.

Первая глава посвящена базовым понятиям программной инженерии. Здесь приведена официальная классификация процессов программной инженерии, подробно рассматриваются основные идеи, содержание и оценка качества классических, современных и перспективных процессов разработки ПО. Спектр категорий обсуждаемых процессов достаточно широк: от дисциплинирующих «тяжеловесных» процессов (линейных и циклических) до облегченных гибких процессов.

Вторая глава знакомит с вопросами руководства программными проектами — основными понятиями и спецификой процесса руководства, планированием проекта, управлением рисками, персоналом, документацией и конфигурацией программного обеспечения.

Третья глава рассматривает комплексную оценку при планировании программного проекта. Вводятся размерно-ориентированные и функционально-ориентированные метрики затрат, обсуждается методика их применения, описывается наиболее популярная модель для оценки затрат — COCOMO II. Приводятся примеры предварительной оценки программного проекта и анализа чувствительности проекта к изменению условий разработки.

Предметом внимания четвертой главы являются формирование и анализ требований заказчика. Описываются разновидности требований, предъявляемых к программным системам, обсуждаются их характеристики и поясняются основные

процессы для работы с требованиями: формирование требований, анализ требований, управление изменениями требований.

Пятая глава поясняет классические методы анализа требований, ориентированные на процедурную реализацию программных систем. Здесь излагаются наиболее популярные, прошедшие проверку временем методы. Изложение носит исторический характер.

Шестая глава отведена основам и месту проектирования в жизненном цикле разработки программных систем. Подробно разъясняется архитектурное проектирование с применением паттернов. Определяются принципы, средства и характеристики проектирования: разделение понятий, модульность, информационная закрытость, пошаговая детализация, аспекты, рефакторинг, сложность, связность, сцепление и метрики для их оценки.

Седьмая глава — это обзор классических, процедурных методов проектирования ПО. Они рассматриваются как исторические корни современных методов проектирования.

Восьмая глава раскрывает принципы объектно-ориентированного представления программных систем — особенности их абстрагирования, инкапсуляции, модульности, иерархической организации. Здесь изучаются характеристики основных строительных элементов объектно-ориентированного ПО — объектов и классов, а также отношения между ними. Далее дается сжатое изложение базовых понятий языка визуального моделирования — UML, описывается его современная версия 2.3 и комментируются механизмы расширения языка.

Девятая глава содержит материал по современным методам и средствам формирования исходных требований заказчика и анализа детальных требований в объектно-ориентированной среде. В качестве средств для формирования требований обсуждаются диаграмма Use Case и диаграмма деятельности, а в качестве средств для анализа требований — диаграмма коммуникации и диаграмма последовательности. Детально поясняется методика оценки затрат на программный проект, ориентированная на обработку диаграмм Use Case. Дополнительно описываются диаграммы конечных автоматов — мощный инструмент для моделирования объектов, управляемых событиями.

Десятая глава освещает широкий круг вопросов проектирования в объектно-ориентированном стиле: принципы детального проектирования, средства языка UML для архитектурного и детального проектирования; паттерн-ориентированное проектирование. Иллюстрируются модель промышленного подхода к компонентной упаковке программного кода, базовые идеи аспектно-ориентированного подхода. Определяются принципы проектирования пользовательского интерфейса и метрики практичности интерфейса, излагаются вопросы развертывания ПО на аппаратных средствах компьютерных систем.

Одиннадцатая глава представляет особенности разработки баз данных в рамках объектно-ориентированной программной системы.

В двенадцатой главе разъясняется метрический аппарат для оценки качества объектно-ориентированных проектных решений: метрики оценки объектно-ориентированной связности, сцепления; аспектно-ориентированные метрики;

широко известные наборы метрик Чидамбера и Кемерера, Фернандо Абреу, Лоренца и Кидда; описывается методика их применения.

Тринадцатая глава решает задачу презентации унифицированного процесса разработки объектно-ориентированных программных систем, на конкретных примерах обучает методике применения этого процесса. Кроме того, здесь рассматривается XP-процесс экстремальной разработки, Scrum-процесс. Применение XP-процесса также иллюстрируется примером.

Четырнадцатая глава определяет базовые понятия структурного тестирования программного обеспечения (по принципу «белого ящика») и знакомит с наиболее популярными методиками данного вида тестирования: тестирования базового пути, тестирования ветвей и операторов отношений, тестирования потоков данных, тестирования циклов.

Пятнадцатая глава вводит в круг понятий функционального тестирования ПО и описывает конкретные способы тестирования — путем разбиения по эквивалентности, анализа граничных значений, построения диаграмм причин–следствий.

Шестнадцатая глава ориентирована на комплексное изложение содержания процесса тестирования: тестирование модулей, тестирование интеграции модулей в программную систему; тестирование правильности, при котором проверяется соответствие системы требованиям заказчика; системное тестирование, при котором проверяется корректность встраивания ПО в цельную компьютерную систему. Здесь же рассматривается организация отладки ПО (с целью устранения выявленных при тестировании ошибок).

Семнадцатая глава обучает особенностям объектно-ориентированного тестирования, проведению такого тестирования на уровне визуальных моделей, уровне методов, уровне классов и уровне взаимодействия классов. Здесь же описывается методика предваряющего тестирования, применяемая в ходе экстремальной разработки и известная также под именем test-first design или test-driven development.

Восемнадцатая глава отражает проблематику оценки качества программных систем. Здесь речь идет о понятии «качество ПО» и всех аспектах его обеспечения: целях, факторах, количественном подходе к определению уровня качества, приводится вся панорама деятельности по обеспечению качества. Отдельно обсуждаются технические проверки и аудиты, инспектирование, верификация и валидация, а также оформление плана по обеспечения качества ПО.

Девятнадцатая глава демонстрирует возможности применения CASE-системы IBM Rational Software Architect к решению задач автоматизации формирования требований, анализа, проектирования и программирования программного продукта.

В приложениях описываются расчетные таблицы для модели затрат COCOMO II, перечисляются термины языка UML и унифицированного процесса разработки, приводится справочный материал по языку программирования Ada 2005.

Учебник предназначен для студентов инженерного, бакалаврского и магистерского уровней компьютерных специальностей, может быть полезен преподавателям, разработчикам промышленного программного обеспечения, менеджерам программных проектов.

Вот и все. Насколько удалась эта работа — судить вам, уважаемый читатель.

Благодарности

Прежде всего, наши слова искренней любви родителям.

Самые теплые слова благодарности нашим семьям, родным, любимым и близким людям. Без их долготерпения, внимания, поддержки, доброжелательности и сердечной заботы эта книга никогда бы не была написана.

Выход в свет этой работы был бы невозможен вне творческой атмосферы, бесчисленных вопросов и положительной обратной связи, которую создавали наши многочисленные студенты.

Авторы искренне признательны талантливым сотрудникам издательства «Питер».

И конечно, огромное спасибо нашим коллегам, всем людям, которые принимали участие в путешествии по городам, улицам и бесконечным переулкам страны ПРОГРАММНАЯ ИНЖЕНЕРИЯ.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Одиннадцатую главу, а также приложения Б и В в формате PDF можно загрузить с сайта издательства.

На веб-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Глава 1

Организация процесса разработки

В этой главе определяются базовые понятия программной инженерии (инженерии программного обеспечения). Как и в любой инженерной дисциплине, основными составляющими программной инженерии являются продукты (программные системы) и процессы, обеспечивающие создание продуктов. Данная глава посвящена процессам. Здесь рассматриваются основные подходы к организации процесса разработки. В главе приводятся примеры классических, современных и перспективных процессов разработки, обсуждаются модели качества процесса разработки.

Основные понятия программной инженерии

Современное человеческое общество все больше и больше зависит от компьютерных систем, управляемых программным обеспечением (ПО). Эти системы весьма настойчиво проникают во все сферы человеческой жизни. Они буквально окружают человека и на работе, и дома, и на отдыхе.

Известно, что аппаратура современных компьютерных систем очень сложна, и вместе с тем считают: сложность программного обеспечения превосходит аппаратную сложность более чем на порядок. Ф. Брукс, известнейший авторитет в данной области, утверждает: «Сложность ПО является существенным, а не второстепенным свойством» [6].

С одной стороны, ПО абстрактно и нематериально. Оно не имеет физической природы и присущих этой природе ограничений: кажется человеку-творцу очень податливой «глиной», из которой можно «вылепить» все, что угодно. С другой стороны, сложность программного обеспечения может превосходить возможности человеческого разума.

Следует различать понятия «программа» и «программное обеспечение». Государственный стандарт 19781-90 и международный стандарт ISO/IEC 2382/1-93 определяют, что ПО включает в себя не только программы, но и всю сопутствующую документацию, а также конфигурационные данные, необходимые для правильной работы программ. Чтобы подчеркнуть наличие многих элементов и от-

дать дань сложности ПО, его часто называют программной системой (ПС). Итак, программные системы состоят из совокупности программ, файлов конфигурации, необходимых для установки этих программ, и документации, которая описывает организацию системы и объясняет пользователям порядок работы с системой.

Создается ПО промышленным способом, коллективом профессионалов-инженеров и продается пользователям в виде программных продуктов. Разрабатываются программные продукты в рамках программных проектов.

Программный проект (project) — это временное предприятие, предназначенное для создания уникальных продуктов, услуг или результатов [14]. Временный характер проекта подчеркивает, что у любого проекта есть определенное начало и завершение. Завершение наступает, когда достигнуты цели проекта или признано, что цели проекта не могут быть достигнуты или исчезла необходимость в проекте. Характеристика «временный», как правило, не относится к создаваемому в ходе проекта продукту, услуге или результату. Большинство проектов предпринимается для достижения устойчивого, длительного результата — время жизни конечного программного продукта может существенно превышать время жизни программного проекта. В состав программного проекта входят как люди (разработчики), так и необходимые материальные ресурсы.

ВНИМАНИЕ

Увы, русскоязычному сообществу не повезло: схожее название (проектирование) имеет один из шагов деятельности программного проекта. Так же называется и выполняемая на этом шаге работа. Суть проектирования состоит в получении эскиза, концептуального решения требуемого ПО, представляемого обычно с помощью рисунков и пояснений к ним. В английской же терминологии здесь употребляется другой термин — design. Будьте внимательны и не называйте проектированием все шаги работы программного проекта.

Термин *программная инженерия* или *инженерия программного обеспечения*[1] впервые был использован в 1968 году в качестве темы конференции, посвященной вопросам максимальной загрузки самых мощных (по тем временам) компьютеров. Там же было дано определение этого термина, не утратившее своей актуальности и в настоящее время:

- программная инженерия (инженерия программного обеспечения) — система инженерных принципов для создания экономичного ПО, которое надежно и эффективно работает в реальных компьютерах [93].

Определение заложило основы новой научно-практической дисциплины: нужно было создать систему инженерных принципов, применимую к разработке ПО, обеспечить экономичность и надежность строительства, а также эффективную работоспособность конечного продукта на различных реальных машинах.

Спустя четверть века международный терминологический стандарт ISO/IEC 2382/1-93 дает более развернутую формулировку:

- программная инженерия — систематическое применение научных и технологических знаний, методов и практического опыта к проектированию, реализации,

[1] Это более точный, хотя и более длинный, перевод английского термина *software engineering*.

тестированию и документированию программного обеспечения в целях оптимизации его производства, поддержки и качества.

Данная формулировка вводит много понятий, смысл которых будет поясняться на протяжении всего учебника. Подчеркнем главное: программная инженерия обеспечивает управляемый, комплексный подход к созданию сложного программного продукта коллективом инженеров, определяя все шаги этой деятельности — от начальной идеи до прекращения использования продукта клиентами. При этом гармонизируется взаимодействие разных специалистов, гарантируя синергетический эффект работы коллектива.

Буквальный русский перевод термина *software engineering* нельзя назвать шедевром: звучит он как-то не по-русски. Поэтому долгое время в русскоязычном пространстве бытовал термин *технология разработки ПО*. Однако такой перевод сужал рамки понятия, оставляя за бортом многие аспекты дисциплины. Именно поэтому в национальных стандартах утвердился перевод *программная инженерия*.

Программная инженерия — сравнительно молодая дисциплина, ее возраст чуть больше сорока лет. Первые двадцать лет (70–80-е годы) в ней доминировал классический, процедурный подход, а во вторые двадцать лет (1990–2000-е годы) — объектно-ориентированный подход к разработке ПО.

Различают методы, средства и процессы программной инженерии.

ПРИМЕЧАНИЕ

Процессом здесь называют набор взаимосвязанных работ, которые преобразуют исходные данные в выходные результаты.

Методы обеспечивают решение широкого спектра технических задач; например, назовем следующие задачи разработки:

- планирование и оценка программного проекта;
- анализ требований к компьютерной системе в целом и к программному обеспечению в частности;
- проектирование структур программ (и структур данных), входящих в состав ПО;
- конструирование программного текста (другие названия: кодирование, программирование, реализация);
- тестирование (выявление ошибок в созданных программах);
- сопровождение ПО, уже используемого заказчиками.

Средства (утилиты) программной инженерии обеспечивают автоматизированную или автоматическую поддержку методов. В целях совместного применения утилиты могут объединяться в системы автоматизированной разработки ПО. Такие системы принято называть CASE-системами. Аббревиатура CASE расшифровывается как Computer Aided Software Engineering (программная инженерия с компьютерной поддержкой).

Процессы являются «клеем», который соединяет методы и утилиты так, что они обеспечивают непрерывную технологическую цепочку разработки. Процессы определяют:

- порядок применения методов и утилит;
- формирование отчетов, форм по соответствующим требованиям;
- контроль, который помогает обеспечивать качество и координировать изменения;
- формирование «вех», по которым руководители оценивают прогресс.

Реальные процессы достаточно сложны, поэтому в теории программной инженерии предлагаются модели — упрощенные и формализованные описания процессов создания ПО. Современная программная инженерия обеспечивает представительный набор моделей процессов, каждая из моделей имеет свои достоинства и недостатки. Работая с этими моделями, следует помнить, что они являются лишь абстрактными представлениями реальной последовательности шагов строительства ПО.

Вместе с тем, применение моделей процессов гарантирует систематический, упорядоченный подход к промышленной разработке. использованию и сопровождению ПО. Фактически эти модели вносят в программные проекты организующее инженерное начало, необходимость которого трудно переоценить.

Прежде чем перейти к рассмотрению наиболее популярных моделей процессов, обсудим современную классификацию процессов программной инженерии.

Официальная классификация процессов программной инженерии

Классификацию процессов программной инженерии задают международный стандарт ISO/IEC 12207-95 «Information Technology – Software Life Cycle Processes» и его российский аналог ГОСТ Р ИСО/МЭК 12207-99.

ПРИМЕЧАНИЕ

В этих стандартах процессы привязаны к основному понятию программной инженерии — жизненному циклу программного обеспечения (ЖЦ ПО). В авторитетном словаре программной инженерии IEEE Std 610.12-90 «IEEE Standard Glossary of Software Engineering Terminology» записано: жизненный цикл 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.

Работы, которые могут выполняться в жизненном цикле ПО, распределены по пяти основным, восьми вспомогательным и четырем организационным процессам.

Основные процессы жизненного цикла

Основные процессы жизненного цикла состоят из пяти процессов, которые реализуются под управлением основных сторон, вовлеченных в жизненный цикл ПО. Под основной стороной понимают одну из тех организаций, которые инициируют или выполняют разработку, эксплуатацию или сопровождение программных продуктов. Основными сторонами являются заказчик, поставщик, разработчик,

оператор и персонал сопровождения программных продуктов. Основными процессами считают:

1. Процесс заказа (acquisition process). Определяет работы заказчика, то есть организации, которая приобретает ПО.
2. Процесс поставки (supply process). Определяет работы поставщика, то есть организации, которая поставляет ПО заказчику.
3. Процесс разработки (development process). Определяет работы разработчика, то есть организации, которая создает программный продукт.
4. Процесс эксплуатации (operation process). Определяет работы оператора, то есть организации, эксплуатирующей вычислительную систему.
5. Процесс сопровождения (maintenance process). Определяет работы сопровождающей организации, которая предоставляет услуги по сопровождению программного продукта, состоящие в контролируемом изменении программного продукта с целью сохранения его исходного состояния и функциональных возможностей. Данный процесс охватывает перенос ПО в другую операционную среду и снятие ПО с эксплуатации.

Вспомогательные процессы жизненного цикла

Вспомогательные процессы жизненного цикла состоят из восьми процессов. Вспомогательный процесс считается целенаправленной составной частью другого процесса, обеспечивающей успешную реализацию и качество выполнения программного проекта. Вспомогательный процесс при необходимости инициируется и используется другим процессом. Вспомогательными процессами являются:

1. Процесс документирования (documentation process). Определяет работы по описанию информации, формируемой в процессе жизненного цикла.
2. Процесс управления конфигурацией (configuration management process). Определяет работы по управлению конфигурацией (конфигурация ПО — это совокупность функциональных и физических характеристик, установленных в технической документации и реализованных в ПО). Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях жизненного цикла.
3. Процесс обеспечения качества (quality assurance process). Определяет работы по объективному обеспечению того, чтобы программный продукт соответствовал установленным требованиям и создавался в рамках утвержденных планов. В качестве методов обеспечения качества могут использоваться совместные оценки, аудиторские проверки, верификация и аттестация.
4. Процесс верификации (verification process). Определяет работы (заказчика, поставщика или независимой стороны) по верификации (проверке реализации конкретных требований) программных продуктов по мере реализации программного проекта.
5. Процесс аттестации (validation process). Определяет работы (заказчика, поставщика или независимой стороны) по аттестации (проверке полноты реализации всех требований) программных продуктов программного проекта.

6. Процесс совместной проверки (joint review process). Определяет работы по оценке состояния и результатов какой-либо работы. Данный процесс может использоваться двумя любыми сторонами, когда одна из сторон (проверяющая) проверяет другую сторону (проверяемую) на совместном совещании.
7. Процесс аудита (audit process). Определяет работы по выявлению соответствия требованиям, планам и договору. Данный процесс может использоваться двумя сторонами, когда одна из сторон (проверяющая) контролирует программные продукты или работы другой стороны (проверяемой). Аудит иначе называют ревизией, проводимой независимым лицом для выявления реального положения дел.
8. Процесс решения проблемы (problem resolution process). Определяет процесс анализа и устранения проблем (включая несоответствия), независимо от их характера и источника, которые были обнаружены во время осуществления разработки, эксплуатации, сопровождения или других процессов.

Организационные процессы жизненного цикла

Организационные процессы жизненного цикла применяются для объединения взаимосвязанных процессов и персонала, а также для постоянного совершенствования результатов объединения. Существуют четыре разновидности организационных процессов. Организационными процессами являются:

1. Процесс управления (management process). Определяет основные работы по управлению, включая управление проектом при реализации процессов жизненного цикла.
2. Процесс создания инфраструктуры (infrastructure process). Определяет работы по выбору и поддержке средств и действий, обеспечивающих жизненный цикл.
3. Процесс усовершенствования (improvement process). Определяет основные работы, которые организация (заказчика, поставщика, разработчика, оператора, персонала сопровождения или администратора другого процесса) выполняет при создании, оценке, контроле и усовершенствовании выбранных процессов жизненного цикла.
4. Процесс обучения (training process). Определяет работы по соответствующему обучению персонала.

Базис процессов разработки ПО

Будем считать, что модели процессов состоят из таких строительных элементов, как виды деятельности, действия и задачи. *Деятельность* — самый крупный элемент, который ориентирован на достижение весомой цели (например, обеспечение взаимодействия с заинтересованными в проекте лицами) и применяется независимо от прикладной области, размера проекта, сложности затрат или степени строгости использования «арсенала» программной инженерии. Деятельность состоит из действий. *Действие* — средний элемент, охватывает набор задач, которые производят этапный рабочий продукт (например, модель результатов проектирования).

Задача — самый мелкий элемент. Задача фокусируется на маленькой, но хорошо определенной цели (например, на проведении тестирования модуля), которая приводит к ощутимому реальному результату.

Модель процесса программной инженерии не является застывшим описанием порядка строительства ПО. Скорее, это адаптивное руководство, позволяющее людям (команде проекта) выполнять работу, указывая или выбирая подходящий набор рабочих действий и задач. Цель — создавать ПО за приемлемое время и с достаточным качеством, удовлетворяющим тех, кто спонсирует его создание и кто будет использовать его.

Р. Прессман предлагает определить в составе базиса процессов такие виды деятельности, которые применимы ко всем программным проектам, независимо от их размера или сложности [88]. Кроме того, целесообразно дополнить базис набором видов защитной деятельности, которые, как мы видели в предыдущем разделе, реально используются в процессах разработки. Обобщенный базис процессов для программной инженерии включает пять видов основной деятельности:

- *Подготовка.* К любой технической работе нужно правильно подготовиться. Подготовка заключается в тесном сотрудничестве с заказчиком и другими заинтересованными лицами. Главное — понять цели заинтересованных лиц в отношении продукта и проекта, собрать их требования к характеристикам и функциям ПО.
- *Планирование.* По понятным причинам опытный путешественник всегда берет в дорогу карту. Программный проект подобен сложному путешествию, и деятельность планирования создает карту, упрощающую «путешествие» команды. Карта называется планом программного проекта. План определяет порядок инженерной работы. Он описывает технические задачи, которые надо выполнять, наиболее вероятные факторы риска, подстерегающие команду, требуемые ресурсы, рабочие продукты (модели, документы, данные, отчеты, формы и т. д.), которые будут созданы, и расписание работы.
- *Моделирование.* В любой человеческой деятельности каждый день работают с моделями. Почему? Ответ в том, что модель — упрощенное представление реальности. Модели «с высоты птичьего полета» наглядно демонстрируют желаемую структуру и поведение системы (обычно визуально). Модели позволяют лучше понять общую картину — эскиз будущего решения. При необходимости эскиз дополняется деталями. Так формируется окончательный способ решения проблемы. Обычно моделирование включает в себя два действия: анализ и проектирование. Модель анализа улучшает понимание требований к ПО, а модель проектирования показывает эскиз структуры и поведения ПО, выполняющего эти требования.
- *Конструирование.* Эта деятельность объединяет два действия: генерацию программного кода ПО (ручную или автоматическую) и тестирование, которое требуется для обнаружения ошибок в коде.
- *Развертывание.* ПО (окончательный вариант или частично завершенная версия) поставляется заказчику, который оценивает полученный продукт и обеспечивает обратную связь, дающую возможность улучшения продукта.

Эти пять видов основной деятельности могут использоваться как в ходе разработки малых, простых программ, так и при создании больших и сложных компьютерных систем. В каждом случае детали процесса (действия, задачи, порядок их выполнения и взаимодействия) будут меняться, но виды деятельности останутся одинаковыми.

Для многих программных проектов перечисленные виды основной деятельности применяются итеративно, по мере развития проекта. То есть подготовка, планирование, моделирование, конструирование и развертывание применяются повторно, в ходе ряда итераций проекта. Каждая итерация проекта производит для заинтересованных лиц версию ПО с частичной реализацией характеристик и функций. По мере производства очередной версии возможности программного продукта возрастают.

Виды основной деятельности по разработке дополняются набором видов защитной деятельности. Виды защитной деятельности «пронизывают» весь программный проект и помогают его команде управлять прогрессом, контролировать развитие, качество, изменения и риск. Перечислим типичные виды защитной деятельности:

- *Отслеживание (трассировка) и контроль программного проекта* — позволяют команде проекта оценивать степень выполнения плана проекта и предоставляют необходимую информацию для модификации расписания.
- *Управление риском* — оценка риска, которая может влиять на результат проекта, или качество продукта; выполнение действий, компенсирующих недопустимую степень риска.
- *Обеспечение качества ПО* — определяет и проводит действия, требуемые для поддержания качества ПО.
- *Технические проверки* — оценивают рабочие продукты программного проекта; цель — обнаружить и удалить ошибки до того, как они распространятся на следующую деятельность.
- *Измерение* — выполняет и накапливает измерения процесса, проекта и продукта, обеспечивающие создание таких характеристик ПО, которые соответствуют нуждам заинтересованных лиц. Используется в сочетании с другими видами основной и защитной деятельности.
- *Управление конфигурацией ПО* — управляет воздействием изменений на ход разработки в течение всего программного процесса.
- *Управление повторной используемостью* — определяет критерии для повторного использования рабочего продукта (включая программные компоненты) и задает механизмы для получения повторно используемых компонентов.
- *Подготовка и производство рабочего продукта* — включает действия, требуемые для создания рабочих продуктов (таких как модели, документы, логи, формы и листинги).

Процесс разработки не является застывшим предписанием, которому команда проекта должна следовать догматически. Напротив, он должен быть гибким и адаптивным (к проблеме, к проекту, к команде, к культуре организации). Поэтому процесс, адаптированный для одного проекта, может существенно отличаться от

процесса, адаптированного для другого проекта. Различия, например, включают в себя:

- Общую последовательность (поток) видов деятельности, действий и задач, а также внутренние зависимости между ними.
- Порядок, в котором определены действия и задачи внутри каждого вида деятельности.
- Порядок выявления и востребованности промежуточных рабочих продуктов.
- Метод применения действий по обеспечению качества.
- Метод применения действий по отслеживанию и контролю проекта.
- Общая степень детализации и строгости, с которой описан процесс.
- Степень привлечения к проекту заказчика и других заинтересованных лиц.
- Степень самостоятельности команды проекта.
- Степень подробности описания организации команды и ролей сотрудников.

Еще раз заметим, если модели процессов применяются догматически и без адаптации, они могут повысить уровень бюрократии программного проекта и непреднамеренно создать трудности для всех заинтересованных лиц.

Модель «классический жизненный цикл»

Старейшей моделью процесса разработки ПО является классический жизненный цикл (автор Уинстон Ройс, 1970) [89].

Очень часто классический жизненный цикл называют каскадной или водопадной моделью, подчеркивая, что разработка рассматривается как последовательность этапов, причем переход на следующий, иерархически нижний этап происходит только после полного завершения работ на текущем этапе (рис. 1.1).

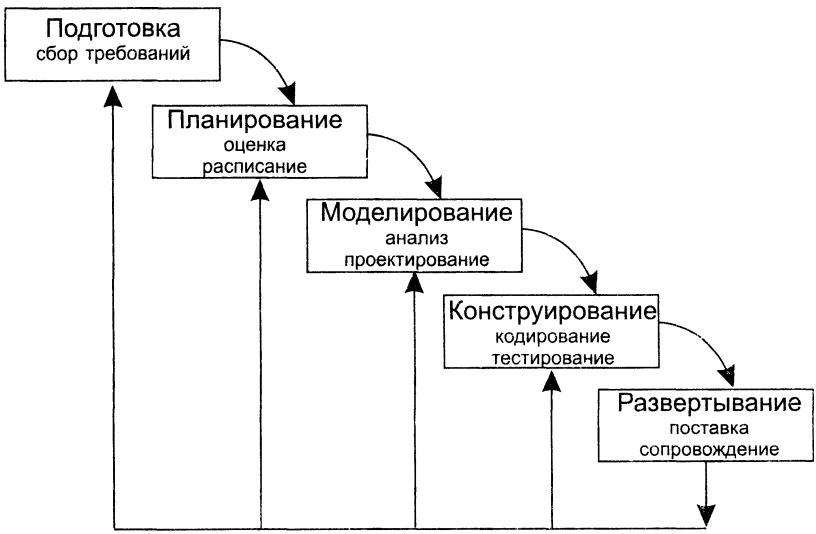


Рис. 1.1. Классический жизненный цикл разработки ПО

Охарактеризуем содержание основных этапов.

Подразумевается, что разработка начинается с заключения контракта с заказчиком (подготовка) и проходит через планирование, моделирование (анализ, проектирование), конструирование (кодирование, тестирование) и развертывание (поставка заказчику, сопровождение). При этом используемые действия аналогичны типовым действиям стандартного инженерного цикла.

Подготовка обеспечивает активное взаимодействие с потенциальным заказчиком. Помимо оформления контракта, здесь собираются и формируются требования, определяющие характеристики и функции будущей программной системы. Этот сбор требует интенсивного диалога с заказчиком и другими заинтересованными в создании системы лицами. Фактически эти требования определяют полное задание на разработку.

Планирование. На этом этапе решаются задачи планирования программного проекта. В ходе планирования проекта определяются объем будущих работ и их риск, необходимые трудозатраты, формируются рабочие задачи и расписание (план-график работ).

Моделирование посвящено выполнению двух действий — анализу требований и проектированию. Результаты этих действий — модели — обычно записываются на графическом языке моделирования, языке картинок.

Анализ требований обрабатывает набор требований к ПО, сформированный на этапе подготовки. Уточняются и детализируются функции, характеристики и интерфейс ПО. Все текстовые определения и модели документируются в *спецификации анализа*.

Проектирование состоит в создании представлений:

- архитектуры ПО;
- структурной и поведенческой организации частей архитектуры ПО;
- входного и выходного интерфейса частей архитектуры (входных и выходных форм данных).

Архитектура ПО определяет организационную структуру программной системы, задает ее разбиение на части, связи между этими частями, механизмы взаимодействия и основные руководящие принципы для детализации дальнейшего проектирования системы.

Архитектура — это множество значимых решений относительно принципов построения программной системы. При проектировании архитектуры выбирают структурные элементы и интерфейсы, с помощью которых они связаны между собой. Архитектура фиксирует:

- крупномасштабную организацию структурных элементов и топологию их связей;
- поведение, описываемое кооперацией этих элементов;
- важные механизмы, доступные во всей системе;
- архитектурный стиль, который управляет организацией элементов системы.

Например, решение создавать программный продукт в виде системы, имеющей два уровня, каждый из которых содержит свои подсистемы, определенным образом сообщающиеся между собой, относится к архитектуре. Архитектура программной

системы затрагивает не только поведение и структуру системы, но также ее использование, функциональность, производительность, устойчивость, возможность повторного применения, способность к восстановлению функций, экономические и технические ограничения и компромиссы, а также вопросы эстетических предпочтений [91].

Исходные данные для проектирования содержатся в *спецификации анализа*. По сути, в ходе проектирования выполняется трансляция требований к ПО во множество проектных представлений. При решении задач проектирования основное внимание уделяется качеству будущего программного продукта.

Конструирование — этот этап включает в себя действия кодирования и тестирования.

Кодирование, иначе называемое программированием или реализацией, — состоит в переводе результатов проектирования в текст на языке программирования.

Тестирование — выполнение программы для выявления ошибок в функциях, логике и форме реализации программного продукта. Если ошибки выявлены, запускается отладка, цель которой — устранить ошибки.

Развертывание — последний этап классического жизненного цикла нацелен на два действия: *поставку* разработанного продукта заказчику и сопровождение процесса эксплуатации этого продукта.

Сопровождение — это внесение изменений в эксплуатируемое ПО. Цели изменений:

- исправление ошибок;
- адаптация к изменениям внешней для ПО среды;
- усовершенствование ПО по требованиям заказчика.

Адаптация обычно требуется, если заказчик хочет использовать продукт с другой операционной системой, а усовершенствование состоит или в расширении функциональности полюбившегося продукта, или в изменении каких-то характеристик (например, ускорения реакции на запросы пользователя).

Согласно статистическим данным, 65% сопровождения связано с усовершенствованием ПО, 18% отводится на адаптацию и 17% связано с исправлением ошибок. Из этого можно заключить, что исправление ошибок не является самой распространенной работой сопровождения. Львиная толика сопровождения ориентирована на модернизацию ПО. Поэтому сопровождение само по себе является естественным продолжением разработки системы со своими этапами подготовки, планирования, моделирования и конструирования.

Сопровождение ПО заключается в повторном применении одного (или нескольких) из предшествующих шагов (этапов) жизненного цикла к существующему ПО, но не в разработке нового ПО. Такая возможность показана на рис. 1.1 стрелками обратных связей.

Как и любая инженерная схема, модель классического жизненного цикла имеет достоинства и недостатки.

Достоинства классического жизненного цикла: дает план и временной график по всем этапам проекта, упорядочивает ход разработки.

Недостатки классического жизненного цикла:

1) реальные проекты часто требуют отклонения от стандартной последовательности шагов;
2) цикл основан на точной формулировке исходных требований к ПО (реально в начале проекта требования заказчика определены лишь частично);
3) результаты проекта доступны заказчику только в конце работы.

Макетирование

Достаточно часто заказчик не может сформулировать подробные требования по вводу, обработке или выводу данных для будущего программного продукта. С другой стороны, разработчик может сомневаться в приспосабливаемости продукта под операционную систему, форме диалога с пользователем или в эффективности реализуемого алгоритма. В этих случаях целесообразно использовать макетирование.

Основная цель макетирования: снять неопределенности в требованиях заказчика.

Макетирование (прототипирование) — это процесс создания модели требуемого программного продукта.

Модель может принимать одну из трех форм:

1) бумажный макет или макет на основе ПК (изображает или рисует человеко-машинный диалог);
2) работающий макет (выполняет некоторую часть требуемых функций);
3) существующая программа (характеристики которой затем должны быть улучшены).

Как показано на рис. 1.2, макетирование основывается на многократном повторении итераций, в которых участвуют заказчик и разработчик.

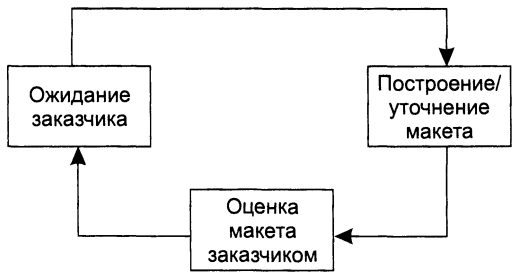


Рис. 1.2. Макетирование

Последовательность действий при макетировании представлена на рис. 1.3.

Макетирование начинается со сбора и уточнения требований к создаваемому ПО. Разработчик и заказчик встречаются и определяют все цели ПО, устанавливают, какие требования известны, а какие предстоит доопределить.

Затем выполняется быстрое проектирование. В нем внимание сосредотачивается на тех характеристиках ПО, которые должны быть видимы пользователю.

Быстрое проектирование приводит к построению макета.

Макет оценивается заказчиком и используется для уточнения требований к ПО.

Итерации повторяются до тех пор, пока макет не выявит все требования заказчика и тем самым не даст возможность разработчику понять, что должно быть сделано.

Достоинство макетирования: обеспечивает определение полных требований к ПО.

Недостатки макетирования:

- заказчик может принять макет за продукт;
- разработчик может принять макет за продукт.

Поясним суть недостатков. Когда заказчик видит работающую версию ПО, он перестает сознавать, что детали макета скреплены «жевательной резинкой и проволокой»; он забывает, что в погоне за работающим вариантом оставлены нерешенными вопросы качества и удобства сопровождения ПО. Когда заказчику говорят, что продукт должен быть перестроен, он начинает возмущаться и требовать, чтобы макет «в три приема» был превращен в рабочий продукт. Очень часто это отрицательно сказывается на управлении разработкой ПО.

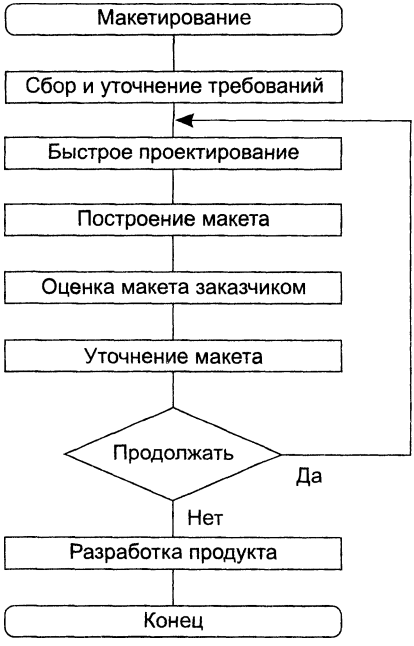


Рис. 1.3. Последовательность действий при макетировании

С другой стороны, для быстрого получения работающего макета разработчик часто идет на определенные компромиссы. Могут использоваться не самый подходящий язык программирования или операционная система. Для простой

демонстрации возможностей может применяться неэффективный алгоритм. Спустя некоторое время разработчик забывает о причинах, по которым эти средства не подходят. В результате далеко не идеальный выбранный вариант интегрируется в систему.

Очевидно, что преодоление этих недостатков требует борьбы с житейским соблазном — принять желаемое за действительное.

Стратегии разработки ПО

Существуют три стратегии разработки ПО:

- *однократный проход* (водопадная стратегия) — линейная последовательность этапов разработки;
- *инкрементная стратегия.* В начале процесса определяются все пользовательские и системные требования, оставшаяся часть разработки выполняется в виде последовательности версий. Первая версия реализует часть запланированных возможностей, следующая версия реализует дополнительные возможности и т. д., пока не будет получена полная система;
- *эволюционная стратегия.* Система также строится в виде последовательности версий, но в начале процесса определены не все требования. Требования уточняются в результате разработки версий.

Характеристики стратегий разработки ПО в соответствии с требованиями стандартов IEEE/EIA 12207.2-1997 и ГОСТ Р ИСО/МЭК ТО 15271-2002 приведены в табл. 1.1.

Таблица 1.1. Характеристики стратегий разработки

Стратегия разработки	В начале процесса определены все требования?	Множество циклов разработки?	Промежуточное ПО распространяется?
Однократный проход	Да	Нет	Нет
Инкрементная (запланированное улучшение продукта)	Да	Да	Может быть
Эволюционная	Нет	Да	Да

Инкрементная модель

Инкрементная модель является классическим примером инкрементной стратегии разработки (рис. 1.4). Она объединяет элементы последовательной водопадной модели с итерационной философией макетирования.

Каждая линейная последовательность здесь вырабатывает поставляемый инкремент (версию) ПО. Например, ПО для обработки слов в 1-м инкременте реализует функции базовой обработки файлов, функции редактирования и документирования; во 2-м инкременте — более сложные возможности редактирования и документирования; в 3-м инкременте — проверку орфографии и грамматики; в 4-м инкременте — возможности компоновки страницы.

Первый инкремент приводит к получению базового продукта, реализующего базовые требования (правда, многие вспомогательные требования остаются нереализованными).

План следующего инкремента предусматривает модификацию базового продукта, обеспечивающего дополнительные характеристики и функциональность.

По своей природе инкрементный процесс итеративен, но в отличие от макетирования инкрементная модель обеспечивает на каждом инкременте работающий продукт.

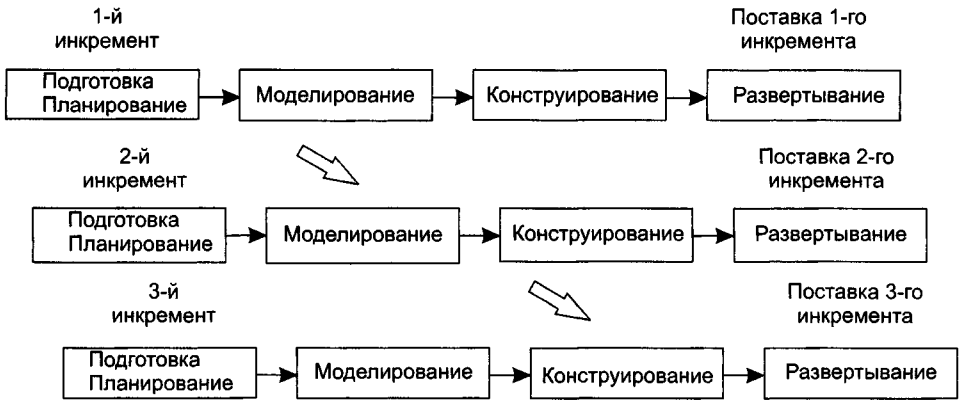


Рис. 1.4. Инкрементная модель

Забегая вперед, отметим, что современная реализация инкрементного подхода — экстремальное программирование XP (Кент Бек, 1999) [25]. Оно ориентировано на очень малые приращения функциональности.

Спиральная модель

Спиральная модель — классический пример применения эволюционной стратегии разработки.

Спиральная модель (автор Барри Боэм, 1988) базируется на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент — анализ риска, отсутствующий в этих моделях [35].

Как показано на рис. 1.5, модель определяет четыре действия, представляемые четырьмя квадрантами спирали.

1. Подготовка — сбор требований и ограничений.
2. Планирование — формирование плана проекта и анализ риска.
3. Моделирование и конструирование — подготовка моделей и реализация продукта следующего уровня.
4. Развертывание — оценка заказчиком текущей версии продукта.

Разработка здесь отображается движением по разворачивающейся спирали (по часовой стрелке), причем проект стартует в первом квадранте. Интегрирующий аспект спиральной модели очевиден при учете радиального измерения спирали.

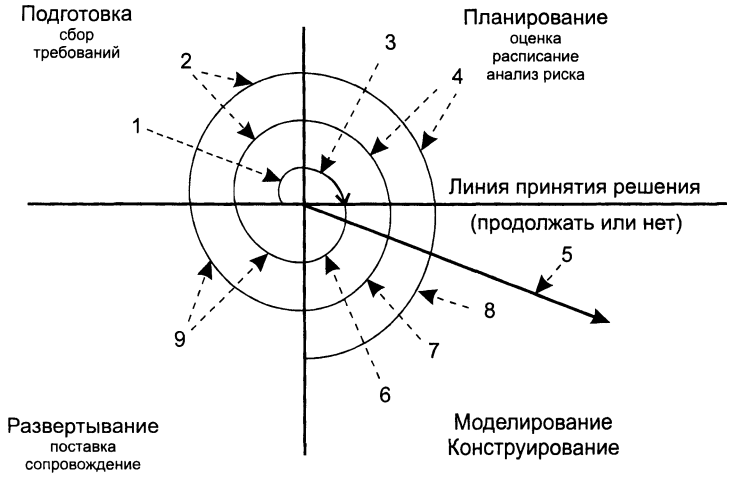


Рис. 1.5. Спиральная модель: начальный сбор требований проекта (*1*); та же работа, но на основе рекомендаций заказчика (*2*); планирование проекта и анализ риска на основе начальных требований (*3*); планирование и анализ риска на основе реакции заказчика (*4*); переход к комплексной системе (*5*); начальный макет системы (*6*); версия системы следующего уровня (*7*); разработанная система (*8*); оценивание заказчиком (*9*)

С каждой итерацией по спирали (продвижением от центра к периферии) строятся все более полные версии ПО.

В первом витке спирали определяются начальные требования и ограничения, составляется начальный план (с учетом распознания и анализа начального риска). Если анализ риска показывает неопределенность требований, на помощь разработчику и заказчику приходит макетирование (используемое в квадранте моделирования и конструирования). Для дальнейшего определения проблемных и уточненных требований может быть использовано моделирование. Заказчик оценивает инженерную работу и вносит предложения по модификации (квадрант развертывания). Следующая фаза подготовки, планирования и анализа риска базируется на предложениях заказчика. В каждом цикле по спирали результаты анализа риска формируются в виде «продолжать, не продолжать». Если риск слишком велик, проект может быть остановлен.

В большинстве случаев движение по спирали продолжается, с каждым шагом продвигая разработчиков к более общей версии системы. В каждом цикле по спирали требуется создание версии продукта (нижний правый квадрант), которое может быть реализовано моделированием-конструированием или макетированием. Заметим, что количество «созидательных» действий, происходящих в правом нижнем квадранте, возрастает по мере продвижения от центра спирали (мы предполагаем, что оно пропорционально длине траектории, проходимой в третьем квадранте).

Достоинства спиральной модели:

1) наиболее реально (в виде эволюции) отображает разработку программного обеспечения;

2) позволяет явно учитывать риск на каждом витке эволюции разработки;
3) включает возможность оценки системы в итерационную структуру разработки;
4) использует моделирование для уменьшения риска и совершенствования программного изделия.

Недостатки спиральной модели:

1) повышенные требования к заказчику;
2) трудности контроля и управления временем разработки.

Компонентно-ориентированная модель

Компонентно-ориентированная модель является развитием спиральной модели и тоже основывается на эволюционной стратегии разработки. В этой модели модифицируется содержание квадранта моделирования-конструирования — оно отражает тот факт, что в современных условиях новая разработка должна основываться на повторном использовании существующих программных компонентов (рис. 1.6).

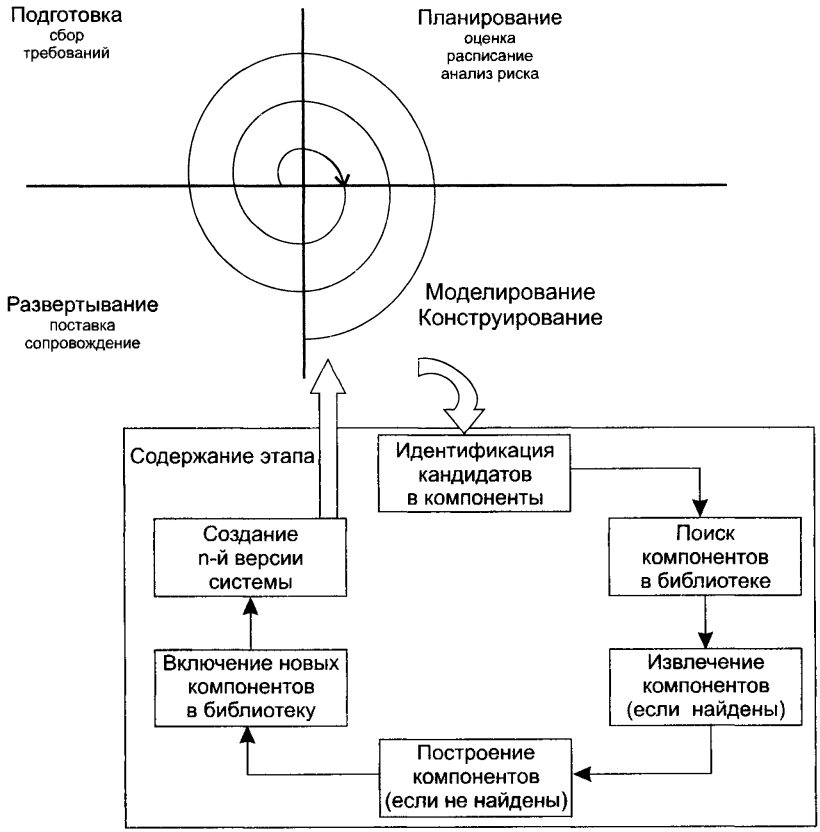


Рис. 1.6. Компонентно-ориентированная модель

Программные компоненты, созданные в реализованных программных проектах, хранятся в библиотеке. В новом программном проекте, исходя из требований заказчика, выявляются кандидаты в компоненты. Далее проверяется наличие этих кандидатов в библиотеке. Если они найдены, то компоненты извлекаются из библиотеки и используются повторно. В противном случае создаются новые компоненты, они применяются в проекте и включаются в библиотеку.

Достоинства компонентно-ориентированной модели:

1) уменьшает на 30% время разработки программного продукта;
2) уменьшает стоимость программной разработки до 70%;
3) увеличивает в полтора раза производительность разработки.

Тяжеловесные и облегченные процессы

В 21 веке потребности общества в программном обеспечении информационных технологий достигли экстремальных значений. Программная индустрия буквально «захлебывается» от потока самых разнообразных заказов. «Больше процессов разработки, хороших и разных!» — скандируют заказчики. «Сейчас, сейчас! Только об этом и думаем!» — отвечают разработчики.

Традиционно для упорядочения и ускорения программных разработок предлагались строго упорядочивающие тяжеловесные (heavyweight) процессы. В этих процессах прогнозируется весь объем предстоящих работ, поэтому они называются прогнозирующими (predictive) процессами. Порядок, который должен выполнять при этом человек-разработчик, чрезвычайно строг — «шаг вправо, шаг влево — виртуальный расстрел!» Иными словами, человеческие слабости в расчет не принимаются, а объем необходимой документации способен отнять покой и сон у «совестливого» разработчика.

В последние годы появилась группа новых, облегченных (lightweight) процессов [49]. Теперь их называют подвижными (agile) или гибкими процессами [22, 45, 57]. Они привлекательны отсутствием бюрократизма, характерного для тяжеловесных (прогнозирующих) процессов. Новые процессы должны воплотить в жизнь разумный компромисс между слишком строгой дисциплиной и полным ее отсутствием. Иначе говоря, порядка в них достаточно для того, чтобы получить разумную отдачу разработчиков. Гибкие процессы требуют меньшего объема документации и ориентированы на человека. В них явно указано на необходимость использования природных качеств человеческой натуры (а не на применение действий, направленных наперекор этим качествам).

Более того, гибкие процессы учитывают особенности современного заказчика, а именно частые изменения его требований к программному продукту. Известно, что для прогнозирующих процессов частые изменения требований подобны смерти. В отличие от них гибкие процессы адаптируют изменения требований и даже выигрывают от этого. Словом, гибкие процессы имеют адаптивную природу.

Таким образом, в современной инфраструктуре программной инженерии существует два семейства процессов разработки:

- семейство прогнозирующих (тяжеловесных) процессов;
- семейство адаптивных (гибких, облегченных) процессов.

У каждого семейства есть свои достоинства, недостатки и область применения:

- адаптивный процесс используют при частых изменениях требований, малочисленной группе высококвалифицированных разработчиков и грамотном заказчике, который согласен участвовать в разработке;
- прогнозирующий процесс применяют при фиксированных требованиях и многочисленной группе разработчиков разной квалификации.

XP-процесс

Экстремальное программирование (eXtreme Programming, XP) — облегченный (гибкий) процесс (или методология), главный автор которого — Кент Бек (1999) [26, 27]. XP-процесс ориентирован на группы малого и среднего размера, строящие программное обеспечение в условиях неопределенных или быстро изменяющихся требований. XP-группу образуют до 10 сотрудников, которые размещаются в одном помещении.

Основная идея XP — устранить высокую стоимость изменения, характерную для приложений с использованием объектов, паттернов[1] и реляционных баз данных. Поэтому XP-процесс должен быть высоко динамичным процессом. XP-группа имеет дело с изменениями требований на всем протяжении итерационного цикла разработки, причем цикл состоит из очень коротких итераций. Четырьмя базовыми действиями в XP-цикле являются: планирование, проектирование, кодирование и тестирование. Динамизм обеспечивается с помощью пяти характеристик: *непрерывная связь с заказчиком* (и в пределах группы), *простота* (всегда выбирается минимальное решение), *быстрая обратная связь* (с помощью модульного и функционального тестирования), *смелость* в проведении профилактики возможных проблем, *уважение* членов команды друг к другу и к своей работе. Кент Бек называет эти характеристики ценностями своей методологии.

Большинство идей, поддерживаемых в XP (минимальность, простота, эволюционный цикл разработки, малая длительность итерации, участие пользователя, человечность, экономичность и т. д.), продиктованы здравым смыслом и применяются в любом упорядоченном процессе. Просто в XP эти принципы, как показано в табл. 1.2, достигают «экстремальных значений». Тот, кто принимает принцип «минимального решения» за хакерство, ошибается, в действительности XP — строго упорядоченный процесс. Простые решения, имеющие высший приоритет, в настоящее время рассматриваются как наиболее ценные части системы, в отличие от проектных решений, которые пока не нужны, а могут (в условиях изменения требований и операционной среды) и вообще не понадобиться.

Базис первой версии XP (1999-го года) формировали двенадцать методик, во второй версии (2004-го года) их количество возросло до двадцати четырех. Перечислим наиболее характерные из XP-методик.

1. Игра планирования (Planning game) — быстрое определение области действия следующей версии путем объединения деловых приоритетов и технических

[1] Паттерн является решением типичной проблемы в определенном контексте.

оценок. Заказчик формирует область действия, приоритетность и сроки с точки зрения бизнеса, а разработчики оценивают и прослеживают продвижение (прогресс) в разработке.

2. Частая смена версий (Small releases) — быстрый запуск в производство простой системы. Новые версии реализуются в очень коротком (двухнедельном) цикле.

Таблица 1.2. Экстремумы в экстремальном программировании

Практика здравого смысла	XP Экстремум	XP Реализация
Проверки кода	Код проверяется все время	Парное программирование
Тестирование	Тестирование выполняется все время, даже с помощью заказчиков	Тесты модулей, тесты приемки
Проектирование	Проектирование является частью ежедневной деятельности каждого разработчика	Рефакторинг
Простота	Для системы выбирается простейшее проектное решение, поддерживающее ее текущую функциональность	Самая простая вещь, которая могла бы работать
Архитектура	Каждый постоянно работает над уточнением архитектуры	Метафора
Тестирование интеграции	Интегрируется и тестируется несколько раз в день	Непрерывная интеграция
Короткие итерации	Итерации являются предельно короткими, продолжаются секунды, минуты, часы, а не недели, месяцы или годы	Игра планирования

3. Инкрементное проектирование (Incremental design) — не следует заниматься подробным проектированием всей системы в самом начале работ. Вместо этого разработчики стараются как можно скорее создавать программный код, чтобы получать отзывы заказчика и улучшать систему по ходу дела. Конечно, чтобы написать хороший код, система должна быть хорошо спроектирована. Однако проектирование (наращивание) функциональности системы должно производиться маленькими порциями, чередуясь с кодированием. Не нужно торопить события. Это, кстати, очень полезно для устранения ненужного дублирования.
4. Простое проектирование (Simple design) — проектирование выполняется настолько просто, насколько это возможно в данный момент.
5. Тестируй, а затем кодируй (Test-first-coding) — вначале разработчики создают тесты для модулей, а затем пишут программный код модулей, одновременно заказчики составляют тесты приемки для проверки правильности функций модулей со своей точки зрения.
6. Рефакторинг (Refactoring) — система реструктурируется, но ее поведение не изменяется; цель — устранить дублирование, улучшить взаимодействие, упростить систему или добавить в нее гибкость.
7. Парное программирование (Pair programming) — весь код пишется двумя программистами, работающими на одном компьютере.

8. Коллективное владение кодом (Collective ownership) — любой разработчик может улучшать любой программный код системы в любое время.
9. Непрерывная интеграция (Continuous integration) — система интегрируется и строится много раз в день, по мере завершения каждой задачи. Непрерывное регрессионное тестирование, то есть повторение предыдущих тестов, гарантирует, что изменения требований не приведут к регрессу функциональности.
10. Локальный заказчик (On-site customer) — в группе все время должен находиться представитель заказчика, действительно готовый отвечать на вопросы разработчиков.

Игра планирования и частая смена версий зависят от заказчика, обеспечивающего набор «историй» (коротких описаний), характеризующих работу, которая будет выполняться для каждой версии системы. Версии генерируются каждые две недели, поэтому разработчики и заказчик должны прийти к соглашению, какие истории будут осуществлены в пределах двух недель. Полную функциональность, требуемую заказчиком, характеризует пул историй; но для следующей двухнедельной итерации из пула выбирается подмножество историй, наиболее важных для заказчика. В любое время в пул могут быть добавлены новые истории, таким образом, требования могут быстро изменяться. Однако процессы двухнедельной генерации основаны на наиболее важных функциях, входящих в текущий пул, следовательно, изменчивость управляется. Локальный заказчик обеспечивает поддержку этого стиля итерационной разработки.

Глобальное «видение» полномасштабного продукта обеспечивает «метафора» — экстремально упрощенный вариант архитектуры. XP подчеркивает желательность проектирования при минимизации проектной документации. Точнее говоря, XP предлагает непрерывное перепроектирование (с помощью рефакторинга), при котором нет нужды в детализированной документации по проектированию, а для инженеров сопровождения единственным надежным источником информации является программный код. Обычно после написания кода документация по проектным решениям выбрасывается. Документация по проектированию сохраняется только в том случае, когда заказчик временно теряет способность придумывать новые истории. Тогда систему помещают в «нафталин» и пишут руководство страниц на пять-десять по нафталиновому варианту системы. Использование рефакторинга приводит к реализации простейшего решения, удовлетворяющего текущую потребность. Изменения в требованиях заставляют отказываться от всех «общих решений».

Парное программирование — одна из наиболее спорных методик в XP, оно влияет на ресурсы, что важно для менеджеров, решающих, будет ли проект использовать XP. Может показаться, что парное программирование удваивает ресурсы, но исследование доказали: парное программирование приводит к повышению качества и уменьшению времени цикла. Для согласованной группы затраты увеличиваются на 15%, а время цикла сокращается на 40–50%. Для интернет-среды увеличение скорости продаж покрывает приращения затрат. Сотрудничество улучшает процесс решения проблем, улучшение качества существенно снижает затраты сопровождения, которые превышают стоимость дополнительных ресурсов по всему циклу разработки.

Коллективное владение означает, что любой разработчик может изменять любой фрагмент кода системы в любое время. Непрерывная интеграция, непрерывное регрессионное тестирование и парное программирование XP обеспечивают защиту от возникающих при этом проблем.

«Тестируй, а затем кодируй» — эта фраза выражает акцент XP на тестировании. Она отражает принцип, по которому сначала планируется тестирование, а тесты создаются для еще несуществующего программного кода, то есть разрабатываются параллельно анализу требований. Размышление о тестировании в начале цикла жизни — хорошо известная практика разработки ПО (правда, редко осуществляемая практически).

Основным средством управления XP является количественный показатель, например «скорость проекта» — количество историй заданного размера, которые могут быть реализованы в итерации.

При принятии XP рекомендуется осваивать его методики по одной, каждый раз выбирая методику, ориентированную на самую трудную проблему группы. Конечно, все эти методики являются «не более чем правилами» — группа может в любой момент поменять их (если ее сотрудники достигли принципиального соглашения по поводу внесенных изменений). Защитники XP признают, что XP оказывает сильное социальное воздействие и не каждый может принять его. Вместе с тем, XP — это методология, обеспечивающая преимущества только при использовании законченного набора базовых методик.

Рассмотрим структуру «идеального» XP-процесса. Основным структурным элементом процесса является XP-реализация, в которую многократно вкладывается базовый элемент — XP-итерация. В состав XP-реализации и XP-итерации входят три фазы — подготовка, планирование, конструирование. Подготовка — это поиск новых требований (историй, задач), которые должна выполнять система. Планирование — выбор для реализации конкретного подмножества из всех возможных требований. Конструирование — проведение разработки, воплощение плана в жизнь.

XP рекомендует: первая реализация должна иметь длительность 2–6 месяцев, продолжительность остальных реализаций — около двух месяцев, каждая итерация длится приблизительно две недели, а численность группы разработчиков не превышает 10 человек. XP-процесс для проекта с семью реализациями, осуществляемый за 15 месяцев, показан на рис. 1.7.

Процесс инициируется начальной фазой подготовки.

Фаза подготовки, с которой начинается любая реализация и итерация, имеет клапан «пропуска», на этой фазе принимается решение о целесообразности дальнейшего продолжения работы.

Предполагается, что длительность первой реализации составляет 3 месяца, длительность второй — седьмой реализаций — 2 месяца. Вторая — седьмая реализации образуют период сопровождения, характеризующий природу XP-проекта. Каждая итерация длится две недели, за исключением тех, которые относят к поздней стадии реализации — «запуску в производство» (в это время темп итерации ускоряется).

Наиболее трудна первая реализация — пройти за три месяца от обычного старта (скажем, отдельный сотрудник не зафиксировал никаких требований, не определены ограничения) к поставке заказчику системы промышленного качества очень сложно.

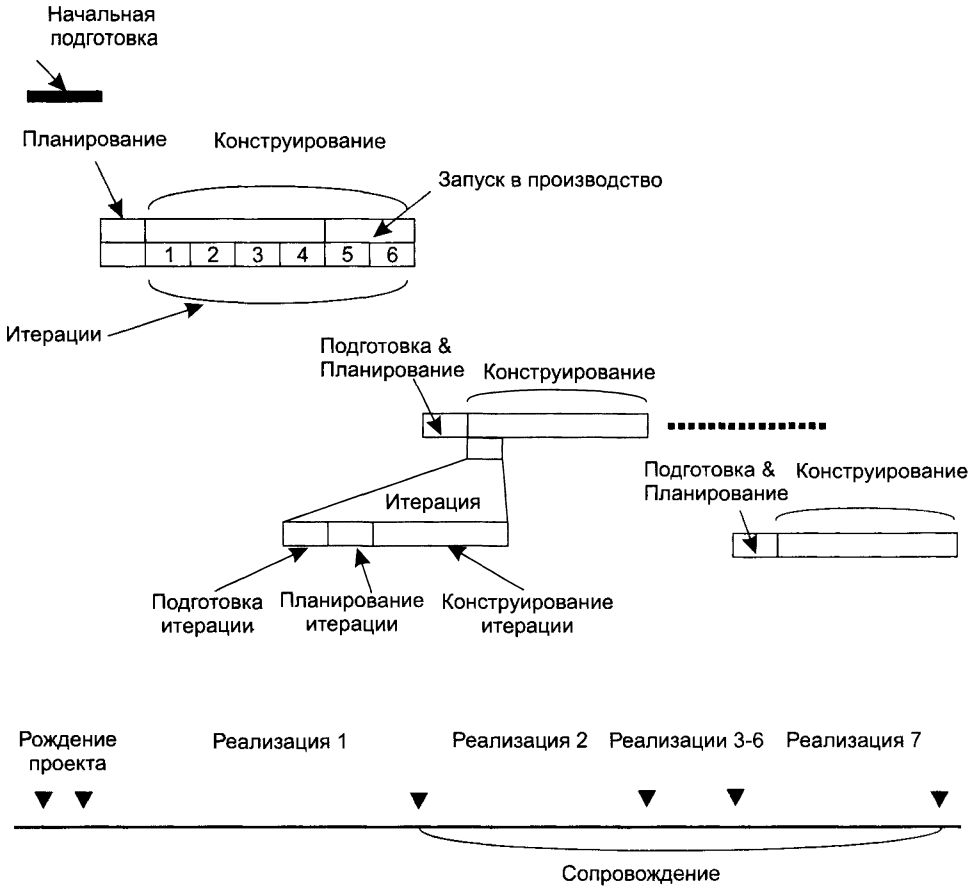


Рис. 1.7. Идеальный XP-процесс

Модели качества процессов разработки

В современных условиях, условиях жесткой конкуренции, очень важно гарантировать высокое качество вашего процесса разработки ПО. Такую гарантию дает сертификат качества процесса, подтверждающий его соответствие принятым международным стандартам. Каждый такой стандарт фиксирует свою модель обеспечения качества. Наиболее авторитетны модели стандартов ISO 9001:2000, ISO/IEC 15504 и модель зрелости процесса разработки ПО (Capability Maturity Model – CMM) Института программной инженерии при американском университете Карнеги–Меллон.

Модель стандарта ISO 9001:2000 ориентирована на процессы разработки из любых областей человеческой деятельности. Стандарт ISO/IEC 15504 специализируется на процессах программной разработки и отличается более высоким уровнем детализации. Достаточно сказать, что объем этого стандарта превышает 500 страниц. Значительная часть идей ISO/IEC 15504 взята из модели CMM.

Базовым понятием модели CMM считается *зрелость* компании [85. 86]. Незрелой называют компанию, где процесс разработки ПО и принимаемые решения зависят только от таланта конкретных разработчиков. Как следствие, здесь высока вероятность превышения бюджета или срыва сроков окончания проекта.

Напротив, в зрелой компании работают ясные процедуры управления проектами и построения программных продуктов. По мере необходимости эти процедуры уточняются и развиваются. Оценки длительности и затрат разработки точны, основываются на накопленном опыте. Кроме того, в компании имеются и действуют корпоративные стандарты на процессы взаимодействия с заказчиком, процессы анализа, проектирования, программирования, тестирования и внедрения программных продуктов. Все это создает среду, обеспечивающую качественную разработку программного обеспечения.

Таким образом, модель CMM фиксирует критерии для оценки зрелости компании и предлагает рецепты для улучшения существующих в ней процессов. Иными словами, в ней не только сформулированы условия, необходимые для достижения минимальной организованности процесса, но и даются рекомендации по дальнейшему совершенствованию процессов.

Рис. 1.8. Пять уровней зрелости модели CMM

Очень важно отметить, что модель CMM ориентирована на построение системы постоянного улучшения процессов. В ней зафиксированы пять уровней зрелости (рис. 1.8) и предусмотрен плавный, поэтапный подход к совершенствованию процессов — можно поэтапно получать подтверждения об улучшении процессов после каждого уровня зрелости.

Начальный уровень (уровень 1) означает, что процесс в компании не формализован. Он не может строго планироваться и отслеживаться, его успех носит

случайный характер. Результат работы целиком и полностью зависит от личных качеств отдельных сотрудников. При увольнении таких сотрудников проект останавливается.

Для перехода на **повторяемый** уровень (уровень 2) необходимо внедрить формальные процедуры для выполнения основных элементов процесса разработки. Результаты выполнения процесса соответствуют заданным требованиям и стандартам. Основное отличие от уровня 1 состоит в том, что выполнение процесса планируется и контролируется. Применяемые средства планирования и управления дают возможность повторения ранее достигнутых успехов.

Следующий, **определенный** уровень (уровень 3) требует, чтобы все элементы процесса были определены, стандартизованы и задокументированы. Основное отличие от уровня 2 заключается в том, что элементы процесса уровня 3 планируются и управляются на основе единого стандарта компании. Качество разрабатываемого ПО уже не зависит от способностей отдельных личностей.

С переходом на **управляемый** уровень (уровень 4) в компании принимаются количественные показатели качества как программных продуктов, так и процесса. Это обеспечивает более точное планирование проекта и контроль качества его результатов. Основное отличие от уровня 3 состоит в более объективной, количественной оценке продукта и процесса.

Высший, **оптимизирующий** уровень (уровень 5) подразумевает, что главной задачей компании становится постоянное улучшение и повышение эффективности существующих процессов, ввод новых технологий. Основное отличие от уровня 4 заключается в том, что технология создания и сопровождения программных продуктов планомерно и последовательно совершенствуется.

Каждый уровень CMM характеризуется *областью ключевых процессов* (ОКП), причем считается, что каждый последующий уровень включает в себя все характеристики предыдущих уровней. Иначе говоря, для 3-го уровня зрелости рассматриваются ОКП 3-го уровня, ОКП 2-го уровня и ОКП 1-го уровня. Область ключевых процессов образуют процессы, которые при совместном выполнении приводят к достижению определенного набора целей. Например, ОКП 5-го уровня образуют процессы:

- предотвращения дефектов;
- управления изменениями технологии;
- управления изменениями процесса.

Если все цели ОКП достигнуты, компании присваивается сертификат данного уровня зрелости. Если хотя бы одна цель не достигнута, то компания не может соответствовать данному уровню CMM.

Контрольные вопросы и упражнения

1. Дайте определение программной инженерии.
2. Что называют программным проектом?
3. Дайте развернутую характеристику официальной классификации процессов программной инженерии. Что такое жизненный цикл ПО?

4. В чем заключается основная идея выделения видов основной деятельности для базиса процессов? Охарактеризуйте каждый из этих видов.
5. Какую специфику имеют виды защитной деятельности?
6. Почему архитектура ПО считается сложным понятием? Обоснуйте эту точку зрения.
7. Какие этапы классического жизненного цикла вы знаете?
8. Охарактеризуйте содержание этапов классического жизненного цикла.
9. Объясните достоинства и недостатки классического жизненного цикла.
10. Чем отличается классический жизненный цикл от макетирования?
11. Какие существуют формы макетирования?
12. Чем отличаются друг от друга стратегии разработки ПО?
13. Укажите сходства и различия классического жизненного цикла и инкрементной модели.
14. Объясните достоинства и недостатки инкрементной модели.
15. Укажите сходства и различия спиральной модели и классического жизненного цикла.
16. В чем состоит главная особенность спиральной модели?
17. Чем отличается компонентно-ориентированная модель от спиральной модели и классического жизненного цикла?
18. Перечислите достоинства и недостатки компонентно-ориентированной модели.
19. Чем отличаются тяжеловесные процессы от облегченных процессов?
20. Чем отличаются тяжеловесные процессы от прогнозирующих процессов?
21. Чем отличаются гибкие процессы от облегченных процессов?
22. Перечислите достоинства и недостатки тяжеловесных процессов.
23. Перечислите достоинства и недостатки облегченных процессов.
24. Приведите примеры тяжеловесных процессов.
25. Приведите примеры облегченных процессов.
26. Перечислите характеристики XP-процесса.
27. Перечислите методики XP-процесса.
28. В чем состоит главная особенность XP-процесса?
29. Охарактеризуйте содержание игры планирования в XP-процессе.
30. Охарактеризуйте назначение метафоры в XP-процессе.
31. Какова особенность проектирования в XP-процессе?
32. Какова особенность программирования в XP-процессе?
33. Что такое рефакторинг?
34. Что такое коллективное владение?
35. Какова особенность тестирования в XP-процессе?

36. Чем отличается XP-реализация от XP-итерации?
37. Чем XP-реализация похожа на XP-итерацию?
38. Какова длительность XP-реализации?
39. Какова длительность XP-итерации?
40. Какова максимальная численность группы XP-разработчиков?
41. Какие модели качества процессов разработки вы знаете?
42. Охарактеризуйте модель CMM.
43. Охарактеризуйте уровень зрелости знакомой вам фирмы.
44. Выберите подходящий процесс разработки для перечисленных ниже программных приложений. Обоснуйте свой выбор.
    - ○ Система решения квадратных уравнений.
    - ○ Система определения оценки по результатам ответа на три экзаменационных вопроса.
    - ○ Информационная система института.
45. Какую модель процесса разработки применяют на знакомой вам фирме? На ваш взгляд, это правильное решение? Если бы вы были топ-менеджером на этой фирме, какую бы предложили модель процесса? Свое предложение обоснуйте.

Глава 2

Руководство программным проектом

В этой главе детально рассматриваются вопросы руководства программным проектом. Читатель знакомится с основными понятиями процесса руководства. Далее здесь описываются пять видов защитной деятельности: планирование проекта, управление риском, управление персоналом, управление документацией и управление конфигурацией.

Основные понятия руководства проектом

Руководство программным проектом является защитной деятельностью программной инженерии, пронизывающей все виды основной деятельности — подготовку, планирование, моделирование, конструирование и развертывание (рис. 2.1). Мало того, именно руководство программным проектом интегрирует в жизненный цикл ПО все остальные виды защитной деятельности.

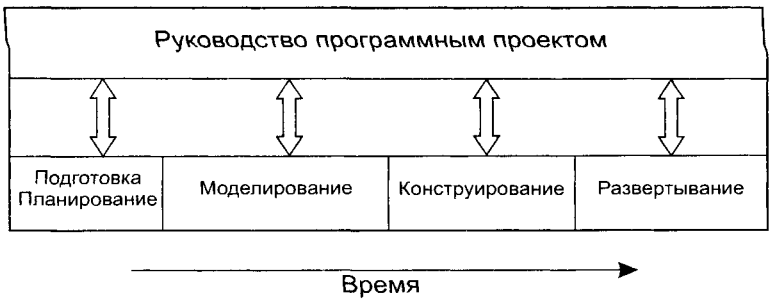


Рис. 2.1. Руководство в процессе разработки ПО

Руководство применяется ко всем четырем «П» разработки: *персоналу* (тем, кто это делает), *процессу* (порядку, в котором это делается), *проекту* (среде, в которой это делается), *продукту* (итогу всех дел).

Цель любого программного *проекта* состоит в производстве определенного программного *продукта*. Понятие «продукт» задает не только текст на языке программирования и откомпилированный двоичный код. Например, туда включаются документация, отчеты по промежуточным итогам, результаты проверок, оценки качества и т. д. Эти элементы обычно называют *артефактами*. То, как в рамках проекта создается продукт, представляет собой *процесс*. Поскольку критичным для успеха дела является взаимодействие членов команды, в рассмотрение включается *персонал*.

Персонал должен быть организован как эффективная команда, в которой обеспечено эффективное взаимодействие и которая мотивирована на выполнение работы с высоким качеством. Требования к продукту должны быть переданы (без искажений) от заказчика к разработчикам, правильно разбиты на части и распределены для обработки между инженерами команды. Процесс должен быть адаптирован как к персоналу, так и к продукту. Должна быть правильно сконфигурирована среда для выполнения процесса, разумно выбран такой набор рабочих задач, который гарантирует оптимальную загрузку персонала. Наконец, организация проекта должна быть нацелена на успешную работу команды.

Руководство проектом заключается в управлении производством продукта в рамках отведенных средств и времени. Поскольку для этого требуются человеческие ресурсы, то для управления проектом необходимы не только технические и организационные навыки, но еще и искусство управления людьми. Управление проектом — очень сложное занятие, привлекающее широкий спектр многоплановых знаний и навыков.

Для проведения успешного проекта нужно понять объем предстоящих работ, возможный риск, требуемые ресурсы, предстоящие задачи, прокладываемые вехи, необходимые усилия (стоимость), план работ, которому желательно следовать. Руководство программным проектом обеспечивает такое понимание. Оно начинается перед технической работой, продолжается по мере развития ПО от идеи к реальности и достигает наивысшего уровня к концу работ [52, 88, 94].

Проблемы управления программными проектами начали проявляться уже в 60–70-х годах 20-го столетия. Именно в это время заговорили о провалах многих программных проектов. Причем подавляющее большинство провалов связывалось с крупными недостатками в руководстве разработкой.

Обычно применяемые методики заимствовали опыт управления проектами из других инженерных областей и «пробуксовывали» при создании программного обеспечения. В чем причина? Ответ прост: разработка ПО существенно отличается от разработки обычных, физических продуктов. Приведем примеры.

Менеджер авиационного проекта физически наблюдает результаты производства самолета и легко замечает, если работа отстает от графика. Напротив, менеджер программного проекта не может увидеть рост программного продукта — он нематериален, а значит ему нужны иные средства для наблюдения и контроля.

Большие программные проекты чаще всего оригинальны, то есть сильно отличаются от прошлых проектов и являются инновационными. Поэтому, чтобы уменьшить промахи в планировании, менеджеры должны полагаться лишь на личный

опыт и «чутье». Но стремительные и революционные изменения в компьютерных средствах и технологиях быстро сводят их к нулю.

Таким образом, отличительной особенностью многих программных проектов является предрасположенность к нарушению ограничений по бюджету и времени. И эту черту надо обязательно учитывать при организации руководства.

Обсудим характерные точки руководства проектом.

Начало проекта

Перед планированием проекта следует:

- установить цели и проблемную (предметную) область проекта;
- обсудить альтернативные решения;
- выявить технические и управленческие ограничения.

Без этой информации нельзя обоснованно оценить стоимость, реально распределить задачи проекта, составить такой план руководства проектом, который обеспечивает всестороннее отображение состояния дел.

Цели и область проекта определяются разработчиком и заказчиком. Цели идентифицируют задачи проекта (без обсуждения способов их решения). Проблемная область задает основные функции ПО, их количественные границы, а также иные характеристики.

ПРИМЕЧАНИЕ

Другое название проблемной области — предметная область. Предметная область (domain) — это область знаний или деятельности с характерными понятиями и терминами, которыми владеют профессионалы, работающие в данной области.

Далее обсуждаются альтернативные решения. Менеджеры и инженеры выбирают лучший подход с точки зрения функциональных ограничений, бюджетных ограничений, возможностей персонала, технических интерфейсов и т. д.

Измерения, меры и метрики

Измерения помогают понять как процесс разработки продукта, так и сам продукт. Измерения процесса производятся в целях его улучшения, измерения продукта — для повышения его качества. В результате измерения определяется *мера* — количественная характеристика какого-либо свойства объекта. Путем непосредственных измерений могут определяться только опорные свойства объекта. Все остальные свойства оцениваются в результате вычисления тех или иных функций от значений опорных характеристик. Вычисления этих функций проводятся по формулам, которые дают числовые значения и называются *метриками*.

В *IEEE Standard Glossary of Software Engineering Terms* метрика определена как мера степени обладания свойством, имеющая числовое значение. В программной инженерии понятия *мера* и *метрика* очень часто рассматривают как синонимы.

Типичные вопросы в этой области:

- Как выбрать наиболее подходящие метрики для процессов и продуктов?
- Как использовать полученные данные?

- Корректно ли использовать измерения для сравнения людей, процессов, продуктов?

Процесс оценки

При планировании программного проекта надо оценить людские ресурсы (в человеко-месяцах), продолжительность (в календарных датах), стоимость (в тыс. $). Обычно исходят из прошлого опыта. Если новый проект по размеру и функциям похож на предыдущий проект, вполне вероятно, что потребуются такие же ресурсы, время и деньги. Ну, а если такого аналога нет? Кроме того, может оказаться, что прошлого опыта недостаточно. Словом, надо изучать способы оценки.

Анализ риска

На этой стадии исследуется область неопределенности, имеющаяся в наличии перед созданием программного продукта. Анализируется ее влияние на проект. Действительно ли поняты пожелания заказчика? Можно ли реализовать требуемые функции в рамках ограничений проекта? Нет ли скрытых от внимания трудных технических проблем? Не станут ли изменения, проявившиеся в ходе проекта, причиной недопустимого отставания по срокам? В результате принимается решение: выполнять проект или нет. Специальный аппарат анализа позволяет атаковать риск прежде, чем он атакует программный проект.

Планирование

В каждом программном проекте существует планирование, но не все планы одинаковы. Предусмотрена ли в плане возможность усовершенствования? Не допускает ли план работу в режиме «ошпаренной кошки»? Имеется ли резерв времени на непредвиденный случай? Как измеряется прогресс? По формуле «что еще сделать?», или есть набор хорошо расставленных вех — контрольных рубежей.

Суть планирования заключается в следующем. Определяется набор задач проекта. Устанавливаются связи между задачами, оценивается сложность каждой задачи. Определяются людские и другие ресурсы. Создается сетевой график задач, проводится его временная разметка.

Трассировка и контроль

Каждая задача, помеченная в плане, отслеживается руководителем проекта. При отставании в решении задачи применяются утилиты повторного планирования. С помощью утилит определяется влияние этого отставания на промежуточную веху и общее время разработки. Под *вехой* понимается временная метка, к которой привязано подведение промежуточных итогов.

В результате повторного планирования:

- могут быть перераспределены ресурсы;
- могут быть реорганизованы задачи;
- могут быть пересмотрены выходные обязательства.

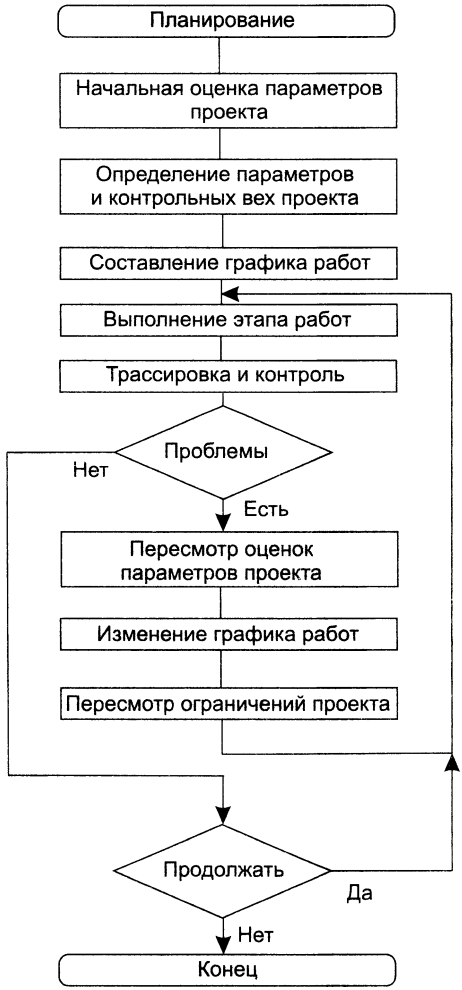


Рис. 2.2. Последовательность действий при планировании

Планирование программного проекта

Эффективность руководства программным проектом целиком определяется правильностью планирования работ, которые необходимы для его выполнения. План помогает предвидеть возможные проблемы разработки ПО и ввести защитные меры для их предупреждения и решения. План создается на начальном этапе проекта (в ходе основной деятельности «планирование) и рассматривается менеджерами и инженерами-разработчиками как руководящий документ, выполнение которого должно привести к успешному завершению проекта. Этот начальный план должен максимально подробно описывать все этапы работы проекта.

Как показано на рис. 2.2, планирование является многошаговым итерационным процессом. Очень важно, чтобы план регулярно пересматривался — ведь по мере

работы в проект непрерывно поступают новые сведения. Важными факторами, которые должны учитываться при разработке плана, являются контрактные обязательства фирмы, требования заказчика, бюджетные и временные ограничения. Их изменения должны оперативно отражаться в плане работ программного проекта.

Планирование начинается с оценки предметной области программной системы, ее размера, трудозатрат и времени на разработку. Формируется команда исполнителей, распределяются их функции. При этом учитываются ограничения по времени, бюджету, наличию и возможностям сотрудников, материально-техническому обеспечению. Затем определяются этапы разработки, контрольные вехи проекта и перечень артефактов — результатов каждого этапа. Напомним, в состав артефактов входит документация, макеты, отчеты, листинги, модели, программный код версий продукта и т. д. Составляется начальный план-график (расписание) работ команды. Далее начинается итерационная часть планирования. Выполняется текущий этап проекта. Его ход отслеживается и контролируется. Выявляются расхождения между реальным и плановым ходом работ.

Если зафиксированы внутренние проблемы или заказчик изменил/расширил список требований, возможен пересмотр первоначальных оценок проекта. Такой пересмотр может привести к модификации начального (или уже промежуточного) графика работ. Если изменения влияют на сроки завершения или стоимость проекта, с заказчиком согласовываются новые ограничения проекта. После этого продолжают проект — переходят к следующему этапу работы.

Конечно, опытные менеджеры закладывают в график резервы на случай неприятностей. Иными словами, в планах фиксируются пессимистические графики работ. Но, разумеется, невозможно построить план, учитывающий все реальные проблемы, поэтому и возникает необходимость периодической коррекции этого руководящего документа.

Структура плана управления программным проектом

План управления проектом должен ясно показать ресурсы, необходимые для воплощения проекта, разделение работ на этапы и временной график выполнения этих этапов. План управления проектом должен быть составлен так, чтобы каждый знал, что и когда ему надо делать. Существует множество стандартов для таких планов. Но в любом случае большинство планов содержат следующие разделы.

1. Введение.

1.1. Обзор проекта. Должен определять проект, но не пытаться охватить все требования к нему. Сами требования приводятся в *Спецификации требований к программному обеспечению.*

1.2. Результирующие артефакты проекта. Список всех документов, исходных файлов и конечных программных продуктов, которые должны быть созданы.

1.3. Развитие плана. Направления ожидаемого расширения и изменения.

1.4. Ссылочные материалы.

1.5. Определения и аббревиатуры.

2. Организация проекта.

2.1. Модель процесса. Ссылаются на тип процесса разработки, который будет использован (например, водопадный, спиральный, инкрементальный).

2.2. Организационная структура. Описывается внутренняя организация команды.

2.3. Организационные рамки и взаимосвязи. Пути возможного взаимодействия между организациями. Все это зависит от заинтересованных в проекте сторон. Например, здесь определяется, каким образом будет осуществляться взаимодействие между отделом разработки и маркетинговым, будут ли это регулярные встречи или переписка по электронной почте и т. д.

2.4. Ответственность за проект. Определяет границы ответственности, то есть кто за что отвечает. Например, за что несет ответственность координатор повышения эффективности команды при горизонтальной организации? Отвечает ли он за общий успех проекта, предоставляет ли персональные рекомендации или занимается только руководством?

3. Анализ рисков.

3.1. Цели и приоритеты. Провозглашается рабочая философия проекта.

3.2. Допущения, зависимости и ограничения.

3.3. Управление рисками.

3.4. Механизмы мониторинга и контроля. Определяют, кто будет управлять, контролировать и (или) осуществлять проверку проекта, а также предписывает, как и когда это должно быть сделано.

3.5. План расстановки кадров.

4. Технический процесс.

4.1. Методы, инструменты и технологии. Накладываются ограничения на языки и используемые инструменты. Может содержать информацию о повторно используемых требованиях и использовании таких техник, как образцы проектирования.

4.2. Документация программного обеспечения.

4.3. Функции сопровождения проекта. Описаны действия для поддержания процесса разработки, такие как *управление конфигурацией* и *обеспечение качества*. Если же функция поддержки представлена в различных документах (например, в плане управления конфигурациями или в плане качества), то в этом пункте будут ссылки на эти документы. В противном случае здесь полностью специфицируются функции поддержки.

5. Распределение работ, график и бюджет.

5.1. Распределение работ. Описывает то, как работа должна распределяться и предоставляться после выполнения. Поскольку программная архитектура еще не утверждена, первая версия этого пункта будет поверхностной. Детали появляются в последующих версиях плана.

5.2. Зависимости.

5.3. Потребности в ресурсах. Оцениваются трудозатраты, аппаратное и программное обеспечение, необходимые для сборки и технической поддержки продукта. Могут быть приведены результаты оценки стоимости. Этот пункт уточняется и детализируется с каждой итерацией.

5.4. Выделение бюджета и ресурсов. Распределяются ресурсы между различными частями проекта в течение всего его жизненного цикла. Приводятся оценки стоимости человеко-дней, могут указываться оценки стоимости аппаратуры и программного обеспечения.

5.5. План-график. Содержит расписание, определяющее, как и когда должны быть выполнены различные этапы процесса.

По мере выполнения проекта план должен регулярно пересматриваться. Одни разделы плана (например, график работ) меняются часто, другие более стабильны. Для внесения изменений в план требуется специальная защитная деятельность, ориентированная на обновление документов.

Структура графика работ программного проекта

Составление графика — одна из самых ответственных работ менеджера проекта. Здесь менеджер оценивает длительность проекта, определяет ресурсы, необходимые для реализации рабочих задач, и разворачивает последовательность задач во времени. Как правило, это действие выполняется с помощью специализированной утилиты планирования проекта.

Планирующие утилиты позволяют:

- определить критический путь (цепочку задач, задающих длительность всего проекта);
- определить длительность критического пути;
- установить для каждой задачи наиболее вероятную временную оценку (по прикладной статистической модели);
- вычислить границы, определяющие временное окно для отдельной задачи.

Результат работы такой утилиты представляется в виде сетевой диаграммы. Типовая сетевая диаграмма приведена на рис. 2.3. Она создается на основе иерархической структуры распределения работ, называемой WBS — Work Breakdown Structure.

Первыми выполняемыми задачами являются сбор требований и анализ требований. Они закладывают фундамент для последующих параллельных задач.

Сбор требований проводится с целью:

1) выяснения потребностей заказчика;
2) оценки выполнимости программной системы;
3) выполнения экономического и технического анализа;
4) определения стоимости и ограничений планирования;
5) создания системной спецификации.

В *системной спецификации* описываются функции, характеристики программной системы, ограничения разработки, входная и выходная информация.

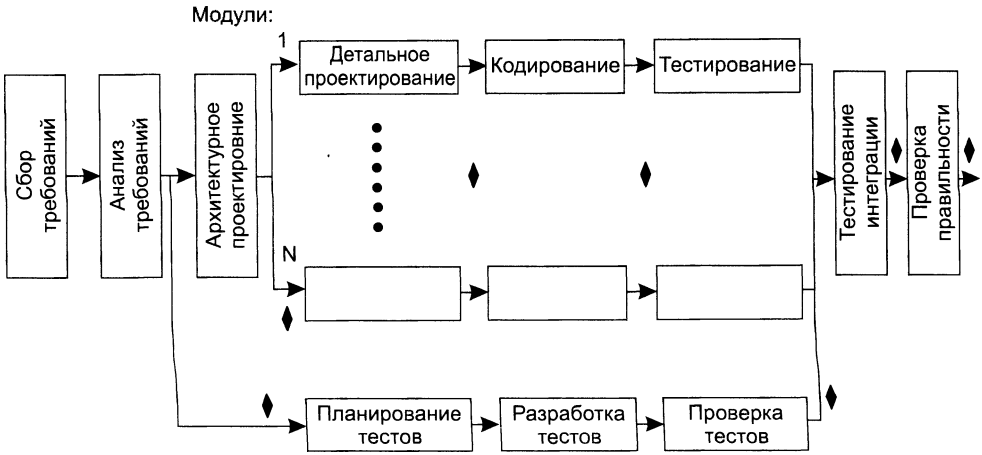


Рис. 2.3. Типовая сетевая диаграмма работ проекта

Анализ требований дает возможность:

1) уточнить функции и характеристики программного продукта;
2) обозначить интерфейс продукта с другими системными элементами;
3) определить проектные ограничения программного продукта;
4) построить модели: процесса, данных, режимов функционирования продукта;
5) создать такие формы представления информации и функций системы, которые можно использовать в ходе проектирования.

Результаты анализа сводятся в *спецификацию анализа*, содержащую конкретизированные требования к программному продукту.

Как видно из типовой структуры, задачи по проектированию и планированию тестов могут быть распараллелены. Благодаря модульной природе ПО для каждого модуля можно предусмотреть параллельный путь для детального (процедурного) проектирования, кодирования и тестирования. После получения всех модулей ПО решается задача тестирования интеграции — объединения элементов в единое целое. Далее проводится тестирование правильности, которое обеспечивает проверку соответствия ПО требованиям заказчика.

Ромбиками на рис. 2.3 обозначены вехи — процедуры контроля промежуточных результатов. Очень важно, чтобы вехи были расставлены через регулярные интервалы (вдоль всего процесса разработки ПО). Это даст возможность менеджеру регулярно получать информацию о текущем положении дел. Вехи распространяются и на документацию как на один из результатов успешного решения задачи.

Параллельность действий повышает требования к планированию. Так как параллельные задачи выполняются асинхронно, планировщик должен определить межзадачные зависимости. Это гарантирует «непрерывность движения к объединению». Кроме того, менеджер проекта должен знать задачи, лежащие на критическом пути. Для того чтобы весь проект был выполнен в срок, необходимо выполнять в срок все критические задачи.

Основной рычаг в планирующих методах — вычисление границ времени выполнения задачи.

Обычно используют следующие оценки:

1. Раннее время начала решения задачи $T_{\min}^{in}$ (при условии, что все предыдущие задачи решены в кратчайшее время).
2. Позднее время начала решения задачи $T_{\max}^{in}$ (еще не вызывает общую задержку проекта).
3. Раннее время конца решения задачи $T_{\min}^{out}$

$$T_{\min}^{out} = T_{\min}^{in} + T_{реш}.$$

4. Позднее время конца решения задачи $T_{\min}^{out}$

$$T_{\max}^{out} = T_{\max}^{in} + T_{реш}.$$

5. Общий резерв — количество избытков и потерь планирования задач во времени, не приводящих к увеличению длительности критического пути $T_{кп}$.

Все эти значения позволяют менеджеру (планировщику) количественно оценить успех в планировании, выполнении задач.

Рекомендуемое правило распределения затрат проекта — 40–20–40:

- на анализ и проектирование приходится 40% затрат (из них на планирование и сбор требований — 5%);
- на кодирование — 20%;
- на тестирование и отладку — 40%.

Данное правило отражает накопленный мировой опыт индустрии программной инженерии и базируется на следующих фактах:

- Работы, связанные со сбором и формализацией требований, определением и детализацией архитектуры ПО, наиболее трудоемки. Они требуют привлечения специалистов высочайшей квалификации, работающих в условиях существенной неопределенности. Ментальные усилия разработчиков здесь максимальны: ведь приходится принимать стратегические решения, сложность которых сравнима со сложностью их творцов.
- Кодирование (программирование) хорошо проработанных проектных решений имеет меньшую сложность. Здесь разработка переходит на тактический уровень. Степень определенности существенно выше, применяются хорошо известные, испытанные приемы и методики.
- Трудоемкость третьего сегмента — тестирования и отладки — тоже очень велика. Это обусловлено доминирующим стилем человеческой деятельности: «методом проб и ошибок». Никого не удивляет, что человек, выполняя работу, часто ошибается. На поиск и устранение ошибок (а это цель данных действий) тратится много усилий.

Управление риском

Словарь русского языка С. И. Ожегова и Н. Ю. Шведовой определяет риск как *«возможность опасности, неудачи»*. Влияние риска вычисляют по выражению:

$$RE = P(UO) \times L(UO),$$

где

- RE — показатель риска (Risk Exposure — подверженность риску);
- $P(UO)$ — вероятность неудовлетворительного результата (Unsatisfactory Outcome);
- $L(UO)$ — потеря при неудовлетворительном результате.

При разработке программного продукта неудовлетворительным результатом может быть: превышение бюджета, низкая надежность, неправильное функционирование и т. д.

Управление риском включает шесть действий:

1. Идентификация риска — выявление элементов риска в проекте.
2. Анализ риска — оценка вероятности и величины потери по каждому элементу риска.
3. Ранжирование риска — упорядочение элементов риска по степени их влияния.
4. Планирование управления риском — подготовка к работе с каждым элементом риска.
5. Разрешение риска — устранение или разрешение элементов риска.
6. Наблюдение риска — отслеживание динамики элементов риска, выполнение корректирующих действий.

Первые три действия относят к этапу оценивания риска, последние три действия — к этапу контроля риска [36].

Идентификация риска

В результате идентификации формируется список элементов риска, специфичных для данного проекта.

Выделяют три категории источников риска: проектный риск, технический риск, коммерческий риск.

Источниками проектного риска являются:

- выбор бюджета, плана, человеческих ресурсов программного проекта;
- формирование требований к программному продукту;
- сложность, размер и структура программного проекта;
- методика взаимодействия с заказчиком.

К источникам технического риска относят:

- трудности проектирования, конструирования, формирования интерфейса, тестирования и сопровождения;
- источность спецификаций;
- техническая неопределенность или отсталость принятого решения.

Главная причина технического риска — реальная сложность проблем выше предполагаемой сложности.

Источники коммерческого риска включают:

- создание продукта, не требующегося на рынке;
- создание продукта, опережающего требования рынка (отстающего от них);
- потерю финансирования.

Лучший способ идентификации — использование проверочных списков риска, которые помогают выявить возможный риск. Например, проверочный список десяти главных элементов программного риска может иметь представленный ниже вид.

1. Дефицит персонала.
2. Нереальные расписание и бюджет.
3. Разработка неправильных функций и характеристик.
4. Разработка неправильного пользовательского интерфейса.
5. Слишком дорогое обрамление.
6. Интенсивный поток изменения требований.
7. Дефицит поставляемых компонентов.
8. Недостатки в задачах, разрабатываемых смежниками.
9. Дефицит производительности при работе в реальном времени.
10. Деформирование научных возможностей.

На практике каждый элемент списка снабжается комментарием — набором методик для предотвращения источника риска.

После идентификации элементов риска следует количественно оценить их влияние на программный проект, решить вопросы о возможных потерях. Эти вопросы решаются на шаге анализа риска.

Анализ риска

В ходе анализа оценивается вероятность возникновения P_i и величина потери L_i для каждого выявленного i-го элемента риска. В результате вычисляется влияние RE_i i-го элемента риска на проект.

Вероятности определяются с помощью экспертных оценок или на основе статистики, накопленной за предыдущие разработки. Итоги анализа, как показано в табл. 2.1, сводятся в таблицу.

Таблица 2.1. Оценка влияния элементов риска

Элемент риска	Вероятность, %	Потери	Влияние риска
1. Критическая программная ошибка	3–5	10	30–50
2. Ошибка потери ключевых данных	3–5	8	24–40
3. Отказоустойчивость недопустимо снижает производительность	4–8	7	28–56
4. Отслеживание опасного условия как безопасного	5	9	45
5. Отслеживание безопасного условия как опасного	5	3	15

Элемент риска	Вероятность, %	Потери	Влияние риска
6. Аппаратные задержки срывают планирование	6	4	24
7. Ошибки преобразования данных приводят к избыточным вычислениям	8	1	8
8. Слабый интерфейс пользователя снижает эффективность работы	6	5	30
9. Дефицит процессорной памяти	1	7	7
10. СУБД теряет данные	2	2	4

Ранжирование риска

Ранжирование заключается в назначении каждому элементу риска приоритета, который пропорционален влиянию элемента на проект. Это позволяет выделить категории элементов риска и определить наиболее важные из них. Например, представленные в табл. 2.1 элементы риска упорядочены по их приоритету.

Для больших проектов количество элементов риска может быть очень велико (30–40 элементов). В этом случае управление риском затруднено. Поэтому к элементам риска применяют принцип Парето 80/20. Опыт показывает, что 80% всего проектного риска приходятся на долю 20% от общего количества элементов риска. В ходе ранжирования определяют эти 20% элементов риска (их называют существенными элементами). В дальнейшем учитывается влияние только существенных элементов риска.

Планирование управления риском

Цель планирования — сформировать набор функций управления каждым элементом риска. Введем необходимые определения.

В планировании используют понятие эталонного уровня риска. Обычно выбирают три эталонных уровня риска: превышение стоимости, срыв планирования, упадок производительности. Они могут быть причиной прекращения проекта. Если комбинация проблем, создающих риск, станет причиной превышения любого из этих уровней, работа будет остановлена. В фазовом пространстве риска эталонному уровню риска соответствует эталонная точка. В эталонной точке решения «продолжать проект» и «прекратить проект» имеют одинаковую силу. На рис. 2.4 показана кривая останова, составленная из эталонных точек.

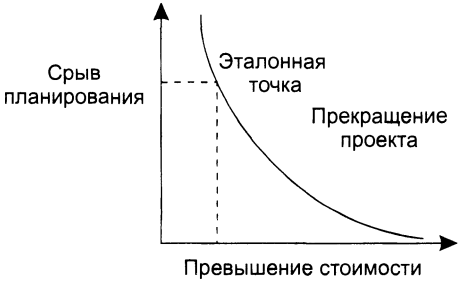


Рис. 2.4. Кривая останова проекта

Ниже кривой располагается рабочая область проекта, выше кривой — запретная область (при попадании в эту область проект должен быть прекращен).

Реально эталонный уровень редко представляется как кривая, чаще это сфера, в которой есть области неопределенности (в них принять решение невозможно).

Теперь рассмотрим последовательность шагов планирования.

1. Исходными данными для планирования является набор четверок: $[R_i, P_i, L_i, RE_i]$, где R_i — i-й элемент риска, P_i — вероятность i-го элемента риска, L_i — потеря по i-му элементу риска, RE_i — влияние i-го элемента риска.
2. Определяются эталонные уровни риска в проекте.
3. Разрабатываются зависимости между каждой четверкой $[R_i, P_i, L_i, RE_i]$ и каждым эталонным уровнем.
4. Формируется набор эталонных точек, образующих сферу останова. В сфере останова предсказываются области неопределенности.
5. Для каждого элемента риска разрабатывается план управления. Предложения плана составляются в виде ответов на вопросы «зачем, что, когда, кто, где, как и сколько».
6. План управления каждым элементом риска интегрируется в общий план программного проекта.

Разрешение и наблюдение риска

Основанием для разрешения и наблюдения является план управления риском. Работы по разрешению и наблюдению производятся с начала и до конца процесса разработки.

Разрешение риска состоит в плановом применении действий по уменьшению риска.

Наблюдение риска гарантирует:

- цикличность процесса слежения за риском;
- вызов необходимых корректирующих воздействий.

Для управления риском используется эффективная методика — «*Отслеживание 10 верхних элементов риска*». Эта методика концентрирует внимание на факторах повышенного риска, экономит много времени, минимизирует «сюрпризы» разработки.

Рассмотрим шаги методики «*Отслеживания 10 верхних элементов риска*».

1. Выполняется выделение и ранжирование наиболее существенных элементов риска в проекте.
2. Производится планирование регулярных просмотров (проверок) процесса разработки. В больших проектах (в группе больше 20 человек) просмотр должен проводиться ежемесячно, в остальных проектах — чаще.
3. Каждый просмотр начинается с обсуждения изменений в 10 верхних элементах риска (их количество может изменяться от 7 до 12). В обсуждении фиксируется текущий приоритет каждого из 10 верхних элементов риска, его приоритет в предыдущем просмотре, частота попадания элемента в список верхних элементов. Если элемент в списке опустился, он по-прежнему нуждается в наблюдении,

но не требует управляющего воздействия. Если элемент поднялся в списке или только появился в нем, то элемент требует повышенного внимания. Кроме того, в обзоре обсуждается прогресс в разрешении элемента риска (по сравнению с предыдущим просмотром).

4. Внимание участников просмотра концентрируется на любых проблемах в разрешении элементов риска.

Управление персоналом

Самым ценным ресурсом программного проекта являются, конечно, люди. Вне сомнений, в первую очередь важны технические навыки инженеров-разработчиков. Однако эти навыки необходимо применять для решения проблем в нужное время и в нужном месте. Следовательно, предполагается комбинация двух стилей: работа в команде и лидерство. Организация команды, обеспечивающей эффективную работу, является весьма сложной задачей для менеджера — руководителя проекта. Хорошая команда должна демонстрировать сплав самых разнообразных качеств: профессиональные навыки, опыт, сплоченность, дух товарищества. Структура команды должна стимулировать творческую работу всех и каждого. Выражаясь языком известнейших в этой области авторов Тома Демарко и Тимоти Листера, команда должна пройти кристаллизацию [4]. Они пишут:

«Команда, прошедшая кристаллизацию, — это группа людей, столь сильно связанных, что целое становится больше суммы составляющих его частей. Производительность этой команды выше, чем производительность тех же людей, не перешедших порог кристаллизации. И, что столь же важно, удовольствие от работы также выше, чем можно было бы ожидать, учитывая природу работы. В некоторых случаях кристаллизованная команда может замечательно себя чувствовать, работая над задачей, которую другие посчитали бы откровенно скучной.

Как только начинается кристаллизация команды, вероятность успеха очень резко возрастает. Команда может стать неумолимой силой, стремящейся к успеху. Управлять этой стихией — одно удовольствие. Управление в традиционном смысле этого слова им не нужно, и уж точно не нужны дополнительные стимулы. Они уже обладают собственным импульсом.

Причины такого явления не столь сложны: команды по природе своей склонны формироваться при наличии целей. До кристаллизации команды ее участники, возможно, имели различные цели. Однако в процессе кристаллизации каждый поверил в общую цель. Эта корпоративная цель обретает особую важность по причине ее значимости для группы. И хотя сама по себе цель может казаться участникам команды выбранной наудачу, они стремятся к ее достижению с невероятным напором».

Обеспечить такую кристаллизацию — главная задача руководителя проекта.

Подбор членов команды

Прежде всего, руководитель должен организовать правильный подбор членов команды: они могут дополнять друг друга по навыкам и опыту и должны быть совместимы друг с другом психологически.

При работе с кандидатом менеджер обычно учитывает следующие аспекты:

1) опыт работы во многих аппаратно-программных средах;
2) знание языков программирования;
3) образование и опыт работы по специальности;
4) коммуникабельность;
5) способность адаптироваться;
6) жизненная позиция;
7) личностные качества.

Поясним некоторые из перечисленных аспектов.

Образование является комплексным показателем начальных знаний и навыков кандидата, а также его способности к обучению. Опыт же характеризует конечные знания и навыки специалиста.

Коммуникабельность характеризует возможности общения с коллегами, руководителями и другими заинтересованными в проекте лицами. Способность к адаптации может пояснять «послужной список» — имеющийся рабочий стаж.

О жизненной позиции судить трудно, но важно. Ведь она говорит о любви к профессии и стремлении развивать знания и навыки.

Личностные качества оценить, пожалуй, труднее всего. Здесь и психологический портрет, и темперамент, и инициативность, и целеустремленность, и многое другое. Именно эти качества определяют совместимость кандидата с коллективом.

Конечно, руководители должны не на словах, а на деле учитывать типичные *человеческие факторы* сотрудников. Люди хотят иметь интересную работу, хотят иметь возможность проявить себя, хотят, чтобы их заметили и наградили, и хотят теплых дружеских отношений в коллективе. Здоровое самоуважение является предпосылкой этих желаний. Один из источников самоуважения — качественная работа. Следовательно, сотрудники должны точно знать, что означает *качество*. Например, сотрудники должны знать, как оценить трудозатраты на создание хорошего продукта, как доказать себе и другим, что работа сделана корректно, и как измерить качество сделанной работы.

Важно также правильно выбрать лидера команды. Он отвечает за техническое руководство или за административное управление (возможно и совмещение этих обязанностей). Лидеры должны быть в курсе повседневной деятельности команды, обеспечивая ее эффективную работу и сотрудничество с руководством проекта. Они должны ладить со всеми членами коллектива, сглаживая напряженности и демпфируя неприятности.

Лидеры во многом обеспечивают сплоченность команды, чувствуя самые тонкие нюансы профессиональных и личностных отношений и помогая отдельным сотрудникам преодолевать трудности. Они «несут знамя командного духа», воспитывают чувство единения, ответственности за работу всей команды.

Как отмечает Б. Шнейдерман [18], ссылаясь на Г. Вейнберга, в сплоченной команде возможно доминирование стиля «программирования без персонализации».

При программировании без персонализации все рабочие продукты (модели, код, документация) считаются собственностью всей команды, а не отдельного сотрудника, который занимался их созданием.

Преимущества *разработки без персонализации*:

- упрощение процедур проверки, критики недостатков, повышение их объективности;
- поощрение стиля непринужденного обсуждения рабочих заданий, достоинств и недостатков отдельных решений;
- активизация дружеских отношений, повышение уровня искренности;
- быстрый рост мастерства (благодаря работе бок о бок);
- улучшение качества, совершенствование результатов работы.

Несомненно, опытные руководители способствуют сплоченности команды, регулярно проводя в этих целях различные совещания, собрания, дискуссии, диспуты, устраивая совместные празднования и другие неофициальные мероприятия.

Взаимодействия в команде

Для команды программного проекта просто необходима развитая система взаимодействия, иными словами, общение и хорошие средства связи между сотрудниками. Сотрудники должны информировать друг друга о ходе работы, принимаемых решениях, а также об изменениях, которые внесены в предыдущие решения. Постоянное взаимодействие тоже способствует сплоченности и повышению качества работы, поскольку сотрудники совместно обсуждают решения, начинают лучше понимать мотивацию своих коллег.

На эффективность взаимодействия влияют следующие параметры.

- *Размер/структура команды.* С ростом числа участников количество связей по взаимодействию растет квадратично. Например, между тремя участниками есть три связи, четыре участника имеют шесть связей, пять человек — десять связей, то есть n человек имеют $(n - 1) + (n - 2) + ... + 1 = n(n - 1)/2$ связей (каждый с каждым). Следовательно, 50 человек должны участвовать в 1225 взаимодействиях. Но ведь это невозможно! Для больших команд альтернативой является их разделение на группы. Каждая группа отвечает за определенную часть проекта и работает над ней. Обычно численность группы не превышает восьми человек. В таких группах проблемы взаимодействия исчезают. Для взаимодействия с другими группами в каждой группе выделяется один сотрудник. Такая структура сохраняет преимущества небольших команд, но позволяет большому количеству людей создавать большие программные продукты.
- *Иерархия команды.* Сотрудники в команде с горизонтальной организацией (один уровень, все сотрудники равны) легче общаются между собой, чем в командах с многоуровневой организацией и иерархией отношений (начальники/подчиненные). В последних командах взаимодействие происходит между уровнями, в иерархической последовательности. Сотрудники одного уровня могут вовсе не общаться между собой. Если в такой команде сотрудники разных групп общаются только через своих менеджеров, возможны запаздывания в разработке и проблемы недопонимания.
- *Рабочее окружение.* Организация рабочего места оказывает существенное влияние на возможности общения. Психологи доказали, что человек всегда предпочитает работать в отдельном помещении, которое он может оформить по своему

вкусу и в котором может сосредоточиться. Исследования подтвердили — в открытом помещении сотруднику сконцентрироваться труднее, следствием чего становится снижение рабочей активности (производительность труда может упасть в два раза).

Состав группы

Увы, не существует универсального рецепта для определения оптимального состава группы разработчиков. Этот состав зависит от большого числа разнообразных факторов: стиля менеджмента, принятого в организации, предметной области и размера проекта, профессиональных возможностей сотрудников организации и т. д.

Перечислим типовые роли:

1. Аналитик — отвечает за развитие и интерпретацию требований заказчика; должен быть экспертом в предметной области, но работать в тесном контакте с остальными сотрудниками.
2. Архитектор — впередсмотрящий; отвечает за проектирование и развитие архитектуры продукта, является одним из наиболее квалифицированных специалистов, имеющих опыт принятия стратегических решений; кроме опыта проектирования, архитектор должен уметь программировать, поскольку его решения воплощаются в программном коде.
3. Конструктор компонентов — главный создатель компонентов (строительных кирпичиков, из которых компонуется продукт).
4. Специалист по применению — программирует компоненты и отвечает за их сборку.
5. Специалист по повторному использованию — внедряет в продукт готовые компоненты из сторонних коммерческих библиотек.
6. Специалист по интеграции — отвечает за сборку совместимых версий компонентов и проверку правильности их совместной работы, поддерживает выпуск версий продукта.
7. Специалист по документации — документирует все реализованные решения, готовит документацию для пользователя.
8. Системный программист — отвечает за создание и адаптацию программных утилит, облегчающих разработку в проекте.
9. Системный администратор — управляет физическими компьютерными ресурсами в проекте.

Разумеется, не каждый проект требует исполнения всех этих ролей. В небольших проектах сотрудники могут играть сразу несколько ролей.

Группа разработчиков обычно пользуется услугами вспомогательного технического персонала и программной библиотеки. Библиотека обслуживает многие группы и выполняет следующие функции: сопровождение и контроль всех элементов программной конфигурации (документации, листингов, данных, моделей, результатов измерений, проверки, оценки), причем хранит весь архив данных по прошлым, уже выполненным проектам.

Управление документацией

Управление документацией программного проекта требует значительных организационных навыков, поскольку документацию можно уподобить сложному, живому организму, который подвержен многочисленным изменениям. Эти изменения очень часто вносятся одновременно и самыми разными людьми. Создание хорошей документации во многом похоже на написание хорошего программного кода.

Управление документацией ориентировано на поддержание ее *полноты* и *согласованности*. *Полнота* подразумевает, что комплект документации должен охватывать весь жизненный цикл ПО. *Согласованность* означает, что комплект документов не содержит внутренних противоречий. Проблема в том, что когда этот комплект достаточно велик, трудно избежать появления в нем взаимоисключающих утверждений.

Стандарты и полнота документации

Разработка программного обеспечения поддерживается документацией. Когда мы смотрим на документированный программный код, его значение становится намного понятнее. Чтобы документация была понятна всем, она должна соответствовать принятым стандартам.

Организации должны делать стандарты простыми и понятными. Перечисленные ниже организации публикуют стандарты, важные для всего человеческого общества.

- Государственный комитет Российской Федерации по стандартизации и метрологии (Госстандарт России) является национальным органом по стандартизации в России. Это федеральный орган исполнительной власти, осуществляющий межотраслевую координацию, а также функциональное регулирование в области стандартизации, метрологии и сертификации.
- Международная организация по стандартизации (ISO, по-русски ИСО) и Международная электротехническая комиссия (IEC, по-русски МЭК)[1] имеют огромное влияние во всем мире, особенно среди организаций производителей, имеющих дело с Евросоюзом (ЕС). ЕС предписывает следование стандартам ISO любой компании, имеющей дело со странами-членами Евросоюза, что является мощным стимулом для поддержания этих стандартов странами всего мира.
- Институт инженеров по электротехнике и радиоэлектронике (IEEE) в течение многих лет весьма активен в создании стандартов документации программного обеспечения. Большинство стандартов разработано различными комитетами, состоящими из опытных и ответственных профессионалов. Некоторые из стандартов IEEE стали также стандартами ANSI (Института американских национальных стандартов).
- Институт программной инженерии (SEI) был учрежден Министерством обороны США в университете Карнеги- Меллон для поднятия уровня программной инженерии у подрядчиков Министерства обороны. Работа SEI также была

[1] Сфера деятельности ИСО касается стандартизации во всех областях, кроме электротехники и электроники, относящихся к компетенции Международной электротехнической комиссии.

принята многими коммерческими компаниями, которые считают улучшение процесса разработки ПО своей стратегической целью.

- Консорциум по технологии манипулирования объектами (OMG) является некоммерческой организацией, в которую входит более 800 авторитетных компаний. OMG устанавливает стандарты для распределенных объектно-ориентированных вычислений. В частности, она использует унифицированный язык моделирования UML в качестве своего стандарта для описания проектов.

Приведем описание полного комплекта документов из набора IEEE. Мы выбрали его как хорошо проработанный комплект, оказавший решающее влияние на все остальные стандарты.

- План верификации и валидации ПО (Software verification and validation plan). Этот план определяет содержание и последовательность проверки шагов проекта и самого продукта на его соответствие поставленным требованиям. Верификация — это процесс проверки правильности создания и сборки продукта; валидация проверяет тот факт, что собран требуемый заказчику продукт. Зачастую валидацию и верификацию осуществляют сторонние организации, в этом случае экспертиза называется независимой (Independent V&V, IV&V).
- План обеспечения качества ПО (Software quality assurance plan). Этот план определяет действия для достижения проектом установленного уровня качества.
- План управления конфигурацией ПО (Software configuration management plan). Он указывает порядок хранения документов, программного кода и их версий, а также устанавливает между ними взаимное соответствие. Крайне неразумно начинать работу без такого плана, так как любой документ обречен на изменения, а мы должны знать, как управлять этими изменениями для поддержания равновесия.
- План управления программным проектом (Software project management plan). Этот план определяет, каким образом руководить проектом. План приводился в разделе *Планирование программного проекта* текущей главы.
- Спецификация требований к ПО (Software requirements specification). Этот документ задает требования к продукту и является частью контракта, заключаемого между заказчиком и фирмой-разработчиком.
- Описание проектирования ПО (Software design description). Описывает архитектуру и детали проектирования продукта, обычно с использованием графических диаграмм.
- Документация тестирования ПО (Software test documentation). Этот документ описывает содержание процесса тестирования программных модулей и всего ПО.

Иногда в проектах привлекается дополнительная документация, разработанная внутри организации разработчика.

Согласованность документации

Для обеспечения непротиворечивости следует сделать так, чтобы описание некоторого факта располагалось только в одном документе.

Положим, для банковской системы, работающей в веб-среде, задано требование «количество попыток авторизации в системе равно трем». Это требование документируется в *Спецификации требований к ПО*. Хотелось бы отразить это требование во всех документах, где оно используется. К чему это приведет? К необходимости отслеживать изменение требования во всех этих местах. Сделать это в условиях дефицита времени крайне сложно. Поэтому разумнее применить механизм гиперссылок (из документа-спецификации требований на все зависимые документы).

Большинство документов зависят друг от друга. Так формируется многоуровневая структура документации. Если появилась новая версия какого-то документа, нужно оперативно отразить это в зависимых документах, автоматически построив новые гиперссылки. Иначе говоря, большинство документов являются живыми объектами, о которых надо заботиться.

Для небольших организаций довольно трудно синхронизировать документооборот. Здесь важно обучить сотрудников применению техники гиперссылок и *убедить* их в необходимости и важности регулярного обновления документации.

Управление конфигурацией

Управление конфигурацией — это координация различных версий и частей документации и программного кода. Управление конфигурацией ПО — защитная деятельность, применяемая на всех этапах жизненного цикла ПО. Она обеспечивает управление изменениями в ПО, которое включает в себя следующие действия:

- Идентификация изменения.
- Контроль изменения.
- Гарантия правильной реализации изменения.
- Формирование сообщения об изменениях.

Управление конфигурацией стартует с началом программного проекта и заканчивается с прекращением использования ПО.

За время жизни программный код продукта претерпевает изменения двух категорий: добавление новых частей, подключение новых версий существующих частей. Конечно, следует учитывать обе категории изменений.

Информацию на выходе процесса разработки ПО можно разделить на три категории:

1. Компьютерные программы (в виде исполняемых кодов).
2. Документы, описывающие программы (как для технического персонала, так и для пользователей).
3. Структуры данных (как внешние, так и внутренние).

Совокупность всех элементов информации, вырабатываемых как часть процесса разработки ПО, называют конфигурацией ПО.

С развитием процесса разработки ПО количество элементов конфигурации стремительно растет.

Минимальная конфигурация ПО включает следующие базовые элементы:

1. Системная спецификация.
2. План программного проекта.

3. Спецификация требований к ПО. Работающий или бумажный макет.
4. Предварительное руководство пользователя.
5. Спецификация проектирования.
6. Листинги исходных текстов программ.
7. План и методика тестирования. Тестовые варианты и полученные результаты.
8. Руководства по работе и инсталляции.
9. Исполняемый код программ.
10. Описание базы данных.
11. Руководство пользователя по настройке.
12. Документы сопровождения. Отчеты о проблемах ПО. Запросы сопровождения. Отчеты об изменениях.
13. Стандарты и методики разработки ПО.

Дополнительно многие организации включают сюда программные утилиты для управления конфигурацией (редакторы, компиляторы, CASE-системы).

Конфигурация ПО — это объект, к которому применяется техника управления конфигурацией.

Управление конфигурацией состоит в применении действий для управления изменениями в течение всего жизненного цикла ПО.

Изменение — непременный факт жизненного цикла ПО. Заказчики хотят изменить требования. Разработчики хотят модифицировать технический подход. Руководство хочет улучшить подход к проектированию. В чем причина? С течением времени все участники узнают что-то новое (о том, в чем они нуждаются; какой подход лучше; как увеличить прибыль).

Эти дополнительные знания и приводят к утверждению, которое плохо воспринимается практиками разработки: *большинство изменений обоснованно*!

Задачи управления конфигурацией: идентификация объектов, контроль версий, контроль изменений, проверка конфигурации и отчетность.

Отчетность о состоянии конфигурации позволяет зафиксировать ответы на следующие вопросы: Что случилось? Кто это сделал? Когда это произошло? Что еще будет затронуто?

Отчетность гарантирует, что разработчики не попадут в ситуацию, когда «левая рука не знает, что делает правая».

Обсудим решение остальных задач управления конфигурацией.

Идентификация объектов в конфигурации ПО

Для контроля и управления должны быть определены объекты конфигурации. Идентифицируются два типа объектов: базисные объекты и составные объекты.

Базисный объект — элемент информации, создаваемый в ходе анализа, проектирования, кодирования или тестирования. Например, базисный объект может быть секцией спецификации требований, частью проектной модели, исходным кодом для компонента или набором тестов для проверки программного кода.

Составной объект является коллекцией базисных объектов и других составных объектов. Например, ПроектнаяСпецификация является составным объектом. Он

может рассматриваться как поименованный (идентифицированный) список указателей, которые определяют такие составные объекты, как АрхитектурнаяМодель и МодельДанных, а также базисные объекты Компонент_N и ДиаграммаКлассов_N.

Каждый объект имеет набор характеристик, которые определяют его уникальность: имя, описание, список ресурсов и реализацию. Имя объекта – строка символов. Описание объекта – перечень элементов данных, определяющий:

- тип элемента (элемент модели, программа, данные);
- идентификатор проекта;
- информацию изменения и/или версии.

Ресурсы – это элементы, которые предоставляются, обрабатываются, указываются или как-то иначе запрашиваются объектом. Например, типы данных, конкретные функции и даже имена переменных могут рассматриваться как ресурсы объектов.

Реализация – указатель на элемент текста для базисного объекта и null для составного объекта.

При идентификации объектов конфигурации могут также рассматриваться отношения, которые существуют между именованными объектами. Например, используя простую нотацию

```
ДиаграммаUseCase <part-of> МодельТребований;
МодельТребований <part-of> СпецификацияТребований,
```

можно создать иерархию объектов конфигурации. Отношение <part-of> определяет иерархию объектов.

Во многих случаях существуют внутренние отношения, пересекающие ветви в иерархии объектов. Такие пересекающие отношения представляются в следующем виде:

```
МодельДанных <interrelated> МодельПотоковДанных;
МодельДанных <interrelated> ТестовыйВариантКласса_M.
```

В первом случае рассматривается внутреннее отношение между составными объектами, во втором – между составным объектом МодельДанных и базисным объектом ТестовыйВариантКласса_M.

Схема идентификации должна выявлять программные объекты, эволюционирующие в процессе разработки.

Контроль версий

Контроль версий объединяет процедуры и средства для управления различными версиями объектов конфигурации, которые создаются в ходе разработки.

Система контроля версий обычно состоит из следующих элементов:

1) база данных проекта, сохраняющая все значимые объекты конфигурации;
2) средство управления версиями, сохраняющее все версии объектов конфигурации (или создающее любую версию на основе различий предыдущих версий);
3) устройство генерации, позволяющее собирать все значимые объекты конфигурации и создавать определенную версию ПО.

Кроме того, системы контроля версий и контроля изменений часто предлагают возможности отслеживания результатов (часто называемые отслеживанием

ошибок), которые позволяют записать и отследить состояние всех нереализованных результатов, связанных с каждым объектом конфигурации.

Некоторые системы контроля версий определяют набор изменений — коллекцию всех изменений (по отношению к базовой конфигурации), которые требуются для создания определенной версии ПО.

Контроль изменений

При разработке больших систем неконтролируемые изменения быстро приводят к хаосу. Контроль изменений обеспечивается механизмом, включающим человеческие действия и автоматические средства. Представим шаги процесса контроля изменений (листинг 2.1). Будем считать, что сформирован запрос на изменение.

Листинг 2.1. Шаги процесса контроля и проведения изменения ПО

```
1.Запрос оценивается по следующим параметрам: техническое качество, эффект, воз-
действие на другие объекты конфигурации и функции системы, стоимость. Формируется
донесение об изменении.
2.Специалист по контролю принимает решение: отвергнуть или утвердить. В случае от-
каза оповещается пользователь и прекращается процесс. В случае утверждения создает-
ся заказ на изменение (описывает само изменение, ограничения, критерии проверки).
3.Для обработки заказа объект конфигурации выбирается из базы данных.
4.Выполняется «выходная» проверка объекта.
5.Выполняется изменение объекта.
6.Проверяется правильность и качество проведенного изменения.
7.Выполняется «входная» проверка объекта, после чего он заносится в базу данных.
8.После утверждения изменения обновленный объект включается в следующую реализацию.
9.Перестраивается соответствующая версия ПО.
10.Проверяются изменения всех элементов конфигурации.
11.Изменения включаются в новую версию.
12.Выпускается новая версия ПО.
```

Видим, что шаги процесса обеспечивают корректность оценки и реализации изменений.

План управления конфигурацией

Приведем пример типового плана управления конфигурацией:

1. **Введение.**
2. **Управление конфигурациями.**
   1. Организация.
   2. Ответственность за управление конфигурациями.
   3. Применяемые методики, директивы, процедуры.
3. **Виды деятельности.**
   1. Определение элементов конфигурации.
      - Именование элементов конфигурации.
      - Получение элементов конфигурации.
   2. Контроль конфигурации.
      - Запрос на изменения.

- Оценка изменений.
- Одобрение или неодобрение изменений.
- Реализация изменений.

3. Определение статуса конфигурации.
4. Аудиты и проверки конфигурации.
5. Контроль интерфейса.
6. Контроль поставщиков и субподрядчиков.

4. **Расписание.**
5. **Ресурсы.**
6. **Сопровождение.**

В этом плане приводится описание всех узловых моментов процесса управления конфигурацией, фиксируются временные ограничения, периодичность выполнения действий.

Контрольные вопросы и упражнения

1. Что такое мера? Приведите примеры мер.
2. Что такое метрика?
3. Что такое выполнение оценки программного проекта?
4. Что такое трассировка и контроль?
5. Поясните последовательность действий при планировании проекта.
6. Какие разделы входят в план программного проекта? Какие разделы следует считать стабильными? Содержание каких разделов меняется быстро?
7. Охарактеризуйте содержание графика работ программного проекта. Какие элементы графика могут быть распараллелены?
8. Как следует расставлять вехи в графике? Обоснуйте ответ.
9. Охарактеризуйте рекомендуемое правило распределения затрат проекта.
10. В чем суть управления риском?
11. Какие действия определяют управление риском?
12. Какие источники проектного риска вы знаете?
13. Какие источники технического риска вы знаете?
14. Какие источники коммерческого риска вы знаете?
15. В чем суть анализа риска?
16. В чем состоит ранжирование риска?
17. В чем состоит планирование управления риском?
18. Что означает разрешение и наблюдение риска? Поясните методику «Отслеживание 10 верхних элементов риска».
19. Какие аспекты следует учитывать при подборе членов команды для программного проекта? Какие из этих аспектов являются главными? Дайте обоснование ответа.

20. За что отвечает лидер команды?
21. Какие преимущества имеет «программирование без персонализации»?
22. Как соотносятся размер и структура команды?
23. Каким образом иерархия команды влияет на ограничения проекта? Какие ограничения вы знаете?
24. Дайте характеристику влияния, которое оказывает на сотрудника рабочее окружение.
25. Какие цели имеет управление документацией?
26. Как добиваются полноты документации?
27. Как поддерживают согласованность документации?
28. В чем суть управления конфигурацией? Дайте развернутый ответ.
29. Когда начинается управление конфигурацией? Когда заканчивается?
30. Что такое конфигурация? Из каких элементов она состоит?
31. Дайте развернутую характеристику задач управления конфигурацией.
32. Поясните понятие объекта конфигурации. Какие существуют типы объектов конфигурации? Чем они схожи? В чем отличаются друг от друга?
33. Поясните возможные отношения между объектами конфигурации. Приведите примеры.
34. Из каких подсистем образуется система контроля версий? В чем их назначение?
35. Выделите наиболее важные, с вашей точки зрения, шаги в процессе проведения изменения. Ответ обоснуйте.
36. Дайте развернутую характеристику и обоснование необходимости основных разделов плана управления конфигурацией.
37. Представьте, что вы назначены менеджером открывающегося проекта. Цель проекта — создать систему для отслеживания успеваемости студентов. Команда разработчиков набирается из студентов вашей группы. Определите структуру и численность команды и выделите возможные риски. Дайте развернутое пояснение принятых решений.

Глава 3

Оценка при планировании программного проекта

В этой главе обсуждаются измерения, производимые при планировании программного проекта. Здесь рассматриваются размерно-ориентированные и функционально-ориентированные метрики затрат, методика их применения. Достаточно подробно описывается наиболее популярная модель для оценивания затрат — COCOMO II. В качестве иллюстраций приводятся примеры предварительной оценки проекта, анализа влияния на проект конкретных условий разработки.

Размерно-ориентированные метрики

Размерно-ориентированные метрики прямо измеряют программный продукт и процесс его разработки. Основываются размерно-ориентированные метрики на LOC-оценках (Lines Of Code). LOC-оценка — это количество строк в программном продукте.

Исходные данные для расчета этих метрик сводятся в таблицу (табл. 3.1).

Таблица 3.1. Исходные данные для расчета LOC-метрик

Проект	Затраты, чел.-мес.	Стоимость, тыс. $	KLOC, тыс. LOC	Прогр. документы, страниц	Ошибки	Люди
aaa01	24	168	12,1	365	29	3
bbb02	62	440	27,2	1224	86	5
ccc03	43	314	20,2	1050	64	6

Таблица содержит данные о проектах за последние несколько лет. Например, запись о проекте aaa01 показывает: 12 100 строк программы были разработаны за 24 человеко-месяца и стоили $168 000. Кроме того, по проекту aaa01 было разработано 365 страниц документации, а в течение первого года эксплуатации было зарегистрировано 29 ошибок. Разрабатывали проект aaa01 три человека.

На основе таблицы вычисляются размерно-ориентированные метрики производительности и качества (для каждого проекта):

$$Производительность = \frac{Длина}{Затраты}\left[\frac{KLOC}{чел.-мес.}\right];$$

$$Качество = \frac{Ошибки}{Длина}\left[\frac{Единиц}{KLOC}\right];$$

$$УдельнаяСтоимость = \frac{Стоимость}{Длина}\left[\frac{Тыс.\$}{LOC}\right];$$

$$Документированность = \frac{СтраницДокумента}{Длина}\left[\frac{Страниц}{KLOC}\right].$$

Достоинства размерно-ориентированных метрик:

1) широко распространены;
2) просты и легко вычисляются.

Недостатки размерно-ориентированных метрик:

1) зависимы от языка программирования;
2) требуют исходных данных, которые трудно получить на начальной стадии проекта;
3) не приспособлены к непроцедурным языкам программирования.

Функционально-ориентированные метрики

Функционально-ориентированные метрики косвенно измеряют программный продукт и процесс его разработки. Вместо подсчета LOC-оценки при этом рассматривается не размер, а функциональность или полезность продукта.

Используется 5 информационных характеристик.

1. *Количество внешних вводов.* Подсчитываются все вводы пользователя, по которым поступают разные прикладные данные. Вводы должны быть отделены от запросов, которые подсчитываются отдельно.
2. *Количество внешних выводов.* Подсчитываются все выводы, по которым к пользователю поступают результаты, вычисленные программным приложением. В этом контексте выводы означают отчеты, экраны, распечатки, сообщения об ошибках. Индивидуальные единицы данных внутри отчета отдельно не подсчитываются.
3. *Количество внешних запросов.* Под запросом понимается диалоговый ввод, который приводит к немедленному программному ответу в форме диалогового вывода. При этом диалоговый ввод в приложении не сохраняется, а диалоговый вывод не требует выполнения вычислений. Подсчитываются все запросы — каждый учитывается отдельно.

4. *Количество внутренних логических файлов.* Подсчитываются все логические файлы (то есть логические группы данных, которые могут быть частью базы данных или отдельным файлом).
5. *Количество внешних интерфейсных файлов.* Подсчитываются все логические файлы из других приложений, на которые ссылается данное приложение.

Вводы, выводы и запросы относят к категории *транзакция*. Транзакция — это элементарный процесс, различаемый пользователем и перемещающий данные между внешней средой и программным приложением. В своей работе транзакции используют внутренние и внешние файлы. Приняты следующие определения.

Внешний ввод — элементарный процесс, перемещающий данные из внешней среды в приложение. Данные могут поступать с экрана ввода или из другого приложения. Данные могут использоваться для обновления внутренних логических файлов. Данные могут содержать как управляющую, так и деловую информацию. Управляющие данные не должны модифицировать внутренний логический файл.

Внешний вывод — элементарный процесс, перемещающий данные, вычисленные в приложении, во внешнюю среду. Кроме того, в этом процессе могут обновляться внутренние логические файлы. Данные создают отчеты или выходные файлы, посылаемые другим приложениям. Отчеты и файлы создаются на основе внутренних логических файлов и внешних интерфейсных файлов. Дополнительно этот процесс может использовать вводимые данные, их образуют критерии поиска и параметры, не поддерживаемые внутренними логическими файлами. Вводимые данные поступают извне, но носят временный характер и не сохраняются во внутреннем логическом файле.

Внешний запрос — элементарный процесс, работающий как с вводимыми, так и с выводимыми данными. Его результат — данные, возвращаемые из внутренних логических файлов и внешних интерфейсных файлов. Входная часть процесса не модифицирует внутренние логические файлы, а выходная часть не несет данных, вычисляемых приложением (в этом и состоит отличие запроса от вывода).

Внутренний логический файл — распознаваемая пользователем группа логически связанных данных, которая размещена внутри приложения и обслуживается через внешние вводы.

Внешний интерфейсный файл — распознаваемая пользователем группа логически связанных данных, которая размещена внутри другого приложения и поддерживается им. Внешний файл данного приложения является внутренним логическим файлом в другом приложении.

Каждой из выявленных характеристик ставится в соответствие сложность. Для этого характеристике назначается низкий, средний или высокий ранг, а затем формируется числовая оценка ранга.

Для транзакций ранжирование основано на количестве ссылок на файлы и количестве типов элементов данных. Для файлов ранжирование основано на количестве типов элементов-записей и типов элементов данных, входящих в файл.

Тип элемента-записи — подгруппа элементов данных, распознаваемая пользователем в пределах файла.

Тип элемента данных — уникальное не рекурсивное (неповторяемое) поле, распознаваемое пользователем. В качестве примера рассмотрим табл. 3.2.

Таблица 3.2. Пример для расчета элементов данных

Уровень активности дня недели			
День	**Хиты**	**% от Сумма хитов**	**Сеансы пользователя**
Понедельник	1887	16,41	201
Вторник	1547	13,45	177
Среда	1975	17,17	195
Четверг	1591	13,83	191
Пятница	2209	19,21	200
Суббота	1286	11,18	121
Воскресенье	1004	8,73	111
Сумма по рабочим дням	9209	80,08	964
Сумма по выходным дням	2290	19,91	232

В этой таблице 10 элементов данных: День, Хиты, % от Сумма хитов, Сеансы пользователя, Сумма хитов (по рабочим дням), % от Сумма хитов (по рабочим дням), Сумма сеансов пользователя (по рабочим дням), Сумма хитов (по выходным дням), % от Сумма хитов (по выходным дням), Сумма сеансов пользователя (по выходным дням). Отметим, что поля День, Хиты, % от Сумма хитов, Сеансы пользователя имеют рекурсивные данные, которые в расчете не учитываются.

Примеры элементов данных для различных характеристик приведены в табл. 3.3, а табл. 3.4 содержит правила учета элементов данных из графического интерфейса пользователя (GUI).

Таблица 3.3. Примеры элементов данных

Информационная характеристика	**Элементы данных**
Внешние вводы	Поля ввода данных, сообщения об ошибках, вычисляемые значения, кнопки
Внешние выводы	Поля данных в отчетах, вычисляемые значения, сообщения об ошибках, заголовки столбцов, которые читаются из внутреннего файла
Внешние запросы	Вводимые элементы: поле, используемое для поиска, щелчок мыши. Выводимые элементы – отображаемые на экране поля

Таблица 3.4. Правила учета элементов данных из графического интерфейса пользователя

Элемент данных	**Правило учета**
Группа радиокнопок	Так как в группе пользователь выбирает только одну радиокнопку, все радиокнопки группы считаются одним элементом данных
Группа флажков (переключателей)	Так как в группе пользователь может выбрать несколько флажков, каждый флажок считают элементом данных
Командные кнопки	Командная кнопка может определять действие добавления, изменения, запроса. Кнопка OK может вызывать транзакции (различных типов). Кнопка Next может быть входным элементом запроса или вызывать другую транзакцию. Каждая кнопка считается отдельным элементом данных

Элемент данных	Правило учета
Списки	Список может быть внешним запросом, но результат запроса может быть элементом данных внешнего ввода

Например, GUI для обслуживания клиентов может иметь поля Имя, Адрес, Город, Страна, Почтовый Индекс, Телефон, Email. Таким образом, имеется 7 полей или семь элементов данных. Восьмым элементом данных может быть командная кнопка (добавить, изменить, удалить). В этом случае каждый из внешних вводов Добавить, Изменить, Удалить будет состоять из 8 элементов данных (7 полей плюс командная кнопка).

Обычно одному экрану GUI соответствует несколько транзакций. Типичный экран включает несколько внешних запросов, сопровождающих внешний ввод.

Обсудим порядок учета сообщений. В приложении с GUI генерируются 3 типа сообщений: сообщения об ошибке, сообщения подтверждения и сообщения уведомления. Сообщения об ошибке (например, Требуется пароль) и сообщения подтверждения (например, Вы действительно хотите удалить клиента?) указывают, что произошла ошибка или что процесс может быть завершен. Эти сообщения не образуют самостоятельного процесса, они являются частью другого процесса, то есть считаются элементом данных соответствующей транзакции.

С другой стороны, уведомление является независимым элементарным процессом. Например, при попытке получить из банкомата сумму денег, превышающую их количество на счете, генерируется сообщение Не хватает средств для завершения транзакции. Оно является результатом чтения информации из файла счета и формирования заключения. Сообщение уведомления рассматривается как внешний вывод.

Данные для определения ранга и оценки сложности транзакций и файлов приведены в табл. 3.5–3.9 (числовая оценка указана в круглых скобках). Использовать их очень просто. Например, внешнему вводу, который ссылается на два файла и имеет 7 элементов данных, по табл. 3.5 назначается средний ранг и оценка сложности 4.

Таблица 3.5. Ранг и оценка сложности внешних вводов

Ссылки на файлы	Элементы данных		
	1–4	5–15	> 15
0–1	Низкий (3)	Низкий (3)	Средний (4)
2	Низкий (3)	Средний (4)	Высокий (6)
> 2	Средний (4)	Высокий (6)	Высокий (6)

Таблица 3.6. Ранг и оценка сложности внешних выводов

Ссылки на файлы	Элементы данных		
	1–4	6–19	> 19
0–1	Низкий (4)	Низкий (4)	Средний (5)
2–3	Низкий (4)	Средний (5)	Высокий (7)
> 3	Средний (5)	Высокий (7)	Высокий (7)

Таблица 3.7. Ранг и оценка сложности внешних запросов

Ссылки на файлы	Элементы данных		
	1–4	6–19	> 19
0–1	Низкий (3)	Низкий (3)	Средний (4)
2–3	Низкий (3)	Средний (4)	Высокий (6)
> 3	Средний (4)	Высокий (6)	Высокий (6)

Таблица 3.8. Ранг и оценка сложности внутренних логических файлов

Типы элементов-записей	Элементы данных		
	1–19	20–50	> 50
1	Низкий (7)	Низкий (7)	Средний (10)
2–5	Низкий (7)	Средний (10)	Высокий (15)
> 5	Средний (10)	Высокий (15)	Высокий (15)

Таблица 3.9. Ранг и оценка сложности внешних интерфейсных файлов

Типы элементов-записей	Элементы данных		
	1–19	20–50	> 50
1	Низкий (5)	Низкий (5)	Средний (7)
2–5	Низкий (5)	Средний (7)	Высокий (10)
> 5	Средний (7)	Высокий (10)	Высокий (10)

Отметим, что если во внешнем запросе ссылка на файл используется как на этапе ввода, так и на этапе вывода, она учитывается только один раз. Такое же правило распространяется и на элемент данных (однократный учет).

После сбора всей необходимой информации приступают к расчету метрики — *количества функциональных указателей FP* (Function Points). Автором этой метрики является А. Албрехт (1979) [20].

Исходные данные для расчета сводятся в табл. 3.10.

Таблица 3.10. Исходные данные для расчета FP-метрик

Имя характеристики	Ранг, сложность, количество			
	Низкий	Средний	Высокий	Итого
Внешние вводы	☐ × 3 = ___	☐ × 4 = ___	☐ × 6 = ___	= ☐
Внешние выводы	☐ × 4 = ___	☐ × 5 = ___	☐ × 7 = ___	= ☐
Внешние запросы	☐ × 3 = ___	☐ × 4 = ___	☐ × 6 = ___	= ☐
Внутренние логические файлы	☐ × 7 = ___	☐ ×10 = ___	☐ ×15 = ___	= ☐
Внешние интерфейсные файлы	☐ × 5 = ___	☐ × 7 = ___	☐ ×10 = ___	= ☐
Общее количество				= ☐

В таблицу заносится количественное значение характеристики каждого вида (по всем уровням сложности). Места подстановки значений отмечены прямо-

угольниками (прямоугольник играет роль метки-заполнителя). Количественные значения характеристик умножаются на числовые оценки сложности. Полученные в каждой строке значения суммируются, давая полное значение для данной характеристики. Эти полные значения затем суммируются по вертикали, формируя общее количество.

Количество функциональных указателей вычисляется по формуле:

$$FP = Общее\ количество \times \left(0,65 + 0,01 \times \sum_{i=1}^{14} F_i\right), \qquad (3.1)$$

где F_i — коэффициенты регулировки сложности.

Каждый коэффициент может принимать следующие значения: 0 — нет влияния, 1 — случайное, 2 — небольшое, 3 — среднее, 4 — важное, 5 — основное.

Значения выбираются эмпирически в результате ответа на 14 вопросов, которые характеризуют системные параметры приложения (табл. 3.11).

Таблица 3.11. Определение системных параметров приложения

Системные параметры		Описание
1	Передачи данных	Сколько средств связи требуется для передачи или обмена информацией с приложением или системой?
2	Распределенная обработка данных	Как обрабатываются распределенные данные и функции обработки?
3	Производительность	Нуждается ли пользователь в фиксации времени ответа или производительности?
4	Распространенность используемой конфигурации	Насколько распространена текущая аппаратная платформа, на которой будет выполняться приложение?
5	Скорость транзакций	Как часто выполняются транзакции? (каждый день, каждую неделю, каждый месяц)
6	Оперативный ввод данных	Какой процент информации надо вводить в режиме онлайн?
7	Эффективность работы конечного пользователя	Приложение проектировалось для обеспечения эффективной работы конечного пользователя?
8	Оперативное обновление	Как много внутренних файлов обновляется в онлайновой транзакции?
9	Сложность обработки	Выполняет ли приложение интенсивную логическую или математическую обработку?
10	Повторная используемость	Приложение разрабатывалось для удовлетворения требований одного или многих пользователей?
11	Легкость инсталляции	Насколько трудны преобразование и инсталляция приложения?
12	Легкость эксплуатации	Насколько эффективны и/или автоматизированы процедуры запуска, резервирования и восстановления?
13	Разнообразные условия размещения	Была ли спроектирована, разработана и поддержана возможность инсталляции приложения в разных местах для различных организаций?
14	Простота изменений	Была ли спроектирована, разработана и поддержана в приложении простота изменений?

После вычисления *FP* на его основе формируются метрики производительности, качества и т. д.:

$$Производительность = \frac{ФункцУказатель}{Затраты}\left[\frac{FP}{чел.\text{-}мес.}\right];$$

$$Качество = \frac{Ошибки}{ФункцУказатель}\left[\frac{Единиц}{FP}\right];$$

$$УдельнаяСтоимость = \frac{Стоимость}{ФункцУказатель}\left[\frac{Тыс.\$}{FP}\right];$$

$$Документированность = \frac{СтраницДокумента}{ФункцУказатель}\left[\frac{Страниц}{FP}\right].$$

Область применения метода функциональных указателей — коммерческие информационные системы. Для продуктов с высокой алгоритмической сложностью используются метрики *указателей свойств* (Features Points). Они применимы к системному и инженерному ПО, ПО реального времени и встроенному ПО.

Для вычисления указателя свойств добавляется одна характеристика — *количество алгоритмов*. Алгоритм здесь определяется как ограниченная подпрограмма вычислений, которая включается в общую компьютерную программу. Примеры алгоритмов: обработка прерываний, инвертирование матрицы, расшифровка битовой строки.

Для формирования указателя свойств составляется табл. 3.12.

Таблица 3.12. Исходные данные для расчета указателя свойств

№	Характеристики	Кол-во	Сложность	Итого
1	Вводы	□	×4	= □
2	Выводы	□	×5	= □
3	Запросы	□	×4	= □
4	Логические файлы	□	×7	= □
5	Интерфейсные файлы	□	×7	= □
6	Количество алгоритмов	□	×3	= □
Общее количество				= □

После заполнения таблицы по формуле (3.1) вычисляется значение указателя свойств. Для сложных систем реального времени это значение на 25–30% больше значения, вычисляемого по таблице для количества функциональных указателей.

Достоинства функционально-ориентированных метрик:

1. Не зависят от языка программирования.
2. Легко вычисляются на любой стадии проекта.

Недостаток функционально-ориентированных метрик: результаты основаны на субъективных данных, используются не прямые, а косвенные измерения.

FP-оценки легко пересчитать в LOC-оценки. Как показано в табл. 3.13, результаты пересчета зависят от языка программирования, используемого для реализации ПО.

Таблица 3.13. Пересчет FP-оценок в LOC-оценки

Язык программирования	Количество операторов на один FP
Ассемблер	320
C	128
Кобол	106
Фортран	106
Паскаль	90
C++	64
Java	53
Ada 95	49
Visual Basic	32
Visual C++	34
Delphi Pascal	29
Smalltalk	22
Perl	21
Html 3	15
LISP	64
Prolog	64
Miranda	40
Haskell	38

Выполнение оценки в ходе планирования проекта

Процесс руководства программным проектом начинается с множества действий, объединяемых общим названием: *планирование проекта*. Первое из этих действий — выполнение оценки. Оно закладывает фундамент для других действий по планированию проекта. При оценке проекта чрезвычайно высока цена ошибок. Очень важно провести оценку с минимальным риском.

Выполнение оценки проекта на основе LOC- и FP-метрик

Цель этой деятельности — сформировать предварительные оценки, которые позволят:

- предъявить заказчику корректные требования по стоимости и затратам на разработку программного продукта;
- составить план программного проекта.

При выполнении оценки возможны два варианта использования LOC- и FP-данных:

- в качестве оценочных переменных, определяющих размер каждого элемента продукта;
- в качестве метрик, собранных за прошлые проекты и входящих в метрический базис фирмы.

Обсудим шаги процесса оценки.

Шаг 1. Область назначения проектируемого продукта разбивается на ряд функций, каждую из которых можно оценить индивидуально:

$$f_1, f_2, ..., f_n.$$

Шаг 2. Для каждой функции f_i планировщик формирует лучшую $LOC_{\text{лучш}_i}$ ($FP_{\text{лучш}_i}$), худшую $LOC_{\text{худш}_i}$ ($FP_{\text{худш}_i}$) и вероятную оценку $LOC_{\text{вероятн}_i}$ ($FP_{\text{вероятн}_i}$). Используются опытные данные (из метрического базиса) или интуиция. Диапазон значения оценок соответствует степени предусмотренной неопределенности.

Шаг 3. Для каждой функции f_i в соответствии с β-распределением вычисляется ожидаемое значение *LOC*- (или *FP*-) оценки:

$$LOC_{\text{ож}i} = (LOC_{\text{лучш}_i} + LOC_{\text{худш}_i} + 4 \times LOC_{\text{вероятн}_i})/6.$$

Шаг 4. Определяется значение *LOC*- или *FP*-производительности разработки функции.

Используется один из трех подходов:

1) для всех функций принимается одна и та же метрика средней производительности $ПРОИЗВ_{\text{ср}}$, взятая из метрического базиса;
2) для *i*-й функции на основе метрики средней производительности вычисляется настраиваемая величина производительности:

$$ПРОИЗВ_i = ПРОИЗВ_{\text{ср}} \times (LOC_{\text{ср}}/LOC_{\text{ож}i}),$$

где $LOC_{\text{ср}}$ – средняя LOC-оценка, взятая из метрического базиса (соответствует средней производительности);

3) для *i*-й функции настраиваемая величина производительности вычисляется по аналогу, взятому из метрического базиса:

$$ПРОИЗВ_i = ПРОИЗВ_{\text{ан}i} \times (LOC_{\text{ан}i}/LOC_{\text{ож}i}).$$

Первый подход обеспечивает минимальную точность (при максимальной простоте вычислений), а третий подход – максимальную точность (при максимальной сложности вычислений).

Шаг 5. Вычисляется общая оценка затрат на проект:
для первого подхода

$$ЗАТРАТЫ = \left(\sum_{i=1}^{n} LOC_{\text{ож}i}\right) / ПРОИЗВ_{\text{ср}} \text{ [чел.-мес.]};$$

для второго и третьего подходов

$$ЗАТРАТЫ = \sum_{i=1}^{n} (LOC_{\text{ожi}} / ПРОИЗВ_i)[\text{чел.-мес.}].$$

Шаг 6. Вычисляется общая оценка стоимости проекта:
для первого и второго подходов

$$СТОИМОСТЬ = \left(\sum_{i=1}^{n} LOC_{\text{ожi}} \right) \times УД\_СТОИМОСТЬ_{\text{ср}},$$

где $УД\_СТОИМОСТЬ_{\text{ср}}$ — метрика средней стоимости одной строки, взятая из метрического базиса.

Для третьего подхода

$$СТОИМОСТЬ = \sum_{i=1}^{n} (LOC_{\text{ожi}} \times УД\_СТОИМОСТЬ_{\text{ан}i}),$$

где $УД\_СТОИМОСТЬ_{\text{ан}i}$ — метрика стоимости одной строки аналога, взятая из метрического базиса.

Пример применения данного процесса оценки приведем ниже.

Конструктивная модель стоимости

В данной модели для вывода формул использовался статистический подход — учитывались реальные результаты огромного количества проектов. Автор оригинальной модели — Барри Боэм (1981) — дал ей название COCOMO 81 (Constructive Cost Model) и ввел в ее состав три разные по сложности статистические подмодели [1].

Иерархию подмоделей Боэма (версии 1981 года) образуют:

- базисная COCOMO — статическая модель, вычисляет затраты разработки и ее стоимость как функцию размера программы;
- промежуточная COCOMO — дополнительно учитывает атрибуты стоимости, включающие основные оценки продукта, аппаратуры, персонала и проектной среды;
- усовершенствованная COCOMO — объединяет все характеристики промежуточной модели, дополнительно учитывает влияние всех атрибутов стоимости на каждый этап процесса разработки ПО (анализ, проектирование, кодирование, тестирование и т. д.).

Подмодели COCOMO 81 могут применяться к трем типам программных проектов. По терминологии Боэма их образуют:

- *распространенный тип* — небольшие программные проекты, в которых работает небольшая группа разработчиков с хорошим стажем работы, устанавливаются мягкие требования к проекту;
- *полунезависимый тип* — средний по размеру проект, выполняется группой разработчиков с разным опытом, устанавливаются как мягкие, так и жесткие требования к проекту;

- *встроенный тип* — программный проект разрабатывается в условиях жестких аппаратных, программных и вычислительных ограничений.

Уравнения базовой подмодели имеют вид:

$$E = a_b \times (KLOC)^{b_b} \text{ [чел.-мес.]};$$

$$D = c_b \times (E)^{d_b} \text{ [мес.]},$$

где E - затраты в человеко-месяцах, D — время разработки, $KLOC$ — количество строк в программном продукте.

Коэффициенты a_b, b_b, c_b, d_b берутся из табл. 3.14.

Таблица 3.14. Коэффициенты для базовой подмодели COCOMO 81

Тип проекта	a_b	b_b	c_b	d_b
Распространенный	2,4	1,05	2,5	0,38
Полунезависимый	3.0	1,12	2,5	0,35
Встроенный	3.6	1,20	2.5	0,32

В 1995 году Боэм ввел более совершенную модель COCOMO II, ориентированную на применение в программной инженерии 21 века [37].

В состав COCOMO II входят:

- модель композиции приложения;
- модель раннего этапа проектирования;
- модель этапа пост-архитектуры.

Для описания моделей COCOMO II требуется информация о размере программного продукта. Возможно использование LOC-оценок, объектных указателей, функциональных указателей.

Модель композиции приложения

Модель композиции используется на ранней стадии разработки ПО, когда:

- рассматривается макетирование пользовательских интерфейсов;
- обсуждается взаимодействие ПО и компьютерной системы;
- оценивается производительность;
- определяется степень зрелости технологии.

Модель композиции приложения ориентирована на применение объектных указателей.

Объектный указатель — средство косвенного измерения ПО, для его расчета определяется количество экранов (как элементов пользовательского интерфейса), отчетов и компонентов, требуемых для построения приложения. Как показано в табл. 3.15, каждый объектный экземпляр (экран, отчет) относят к одному из трех уровней сложности. Здесь места подстановки измеренных и вычисленных значе-

ний отмечены прямоугольниками (прямоугольник играет роль метки-заполнителя). В свою очередь, сложность является функцией от параметров клиентских и серверных таблиц данных (табл. 3.16 и 3.17), которые требуются для генерации экрана и отчета, а также от количества представлений и секций, входящих в экран или отчет.

Таблица 3.15. Оценка количества объектных указателей

Тип объекта		Количество	ВЕС			Итого
			Простой	Средний	Сложный	
1	Экран	□	×1	×2	×3	= □
2	Отчет	□	×2	×5	×8	= □
3	3GL-компонент	□			×10	= □
Объектные указатели						= □

Таблица 3.16. Оценка сложности экрана

ЭКРАНЫ	Количество серверных (срв) и клиентских (клт) таблиц данных		
Количество представлений	Всего < 4 (< 2 срв, < 3 клт)	Всего < 8 (2–3 срв, 3–5 клт)	Всего ≥ 8 (> 3 срв, > 5 клт)
< 3	Простой	Простой	Средний
3–7	Простой	Средний	Сложный
> 8	Средний	Сложный	Сложный

Таблица 3.17. Оценка сложности отчета

ОТЧЕТЫ	Количество серверных (срв) и клиентских (клт) таблиц данных		
Количество представлений	Всего < 4 (< 2 срв, < 3 клт)	Всего < 8 (2–3 срв, 3–5 клт)	Всего ≥ 8 (> 3 срв, > 5 клт)
0 или 1	Простой	Простой	Средний
2 или 3	Простой	Средний	Сложный
≥4	Средний	Сложный	Сложный

После определения сложности количество экранов, отчетов и компонентов взвешивается в соответствии с табл. 3.15. Количество объектных указателей определяется перемножением исходного числа объектных экземпляров на весовые коэффициенты и последующим суммированием промежуточных результатов.

Для учета реальных условий разработки вычисляется процент повторного использования программных компонентов *%REUSE* и определяется количество новых объектных указателей *NOP*:

$$NOP = (\textit{Объектные указатели}) \times [(100 - \%REUSE)/100].$$

Для оценки затрат, основанной на величине *NOP*, надо знать скорость разработки продукта *PROD*. Эту скорость определяют по табл. 3.18, учитывающей уровень опытности разработчиков и зрелость среды разработки.

Таблица 3.18. Оценка скорости разработки

Опытность/возможности разработчика	Зрелость/возможности среды разработки	PROD
Очень низкая	Очень низкая	4
Низкая	Низкая	7
Номинальная	Номинальная	13
Высокая	Высокая	25
Очень высокая	Очень высокая	50

Проектные затраты оцениваются по формуле:

$$ЗАТРАТЫ = NOP / PROD [чел.-мес.]$$

где *PROD* — производительность разработки, выраженная в терминах объектных указателей.

В более развитых моделях дополнительно учитывается множество масштабных факторов, формирователей затрат, процедур поправок.

Модель раннего этапа проектирования

Модель раннего этапа проектирования используется в период, когда стабилизируются требования и определяется базисная программная архитектура.

Основное уравнение этой модели имеет следующий вид:

$$ЗАТРАТЫ = A \times РАЗМЕР^{B} \times M_e + ЗАТРАТЫ_{auto} [чел.-мес.],$$

где

- масштабный коэффициент $A = 2,5$;
- показатель B отражает нелинейную зависимость затрат от размера проекта (размер системы *РАЗМЕР* выражается в тысячах LOC);
- множитель поправки M_e зависит от 7 формирователей затрат, характеризующих продукт, процесс и персонал;
- слагаемое $ЗАТРАТЫ_{auto}$ отражает затраты на автоматически генерируемый программный код.

Значение показателя степени B изменяется в диапазоне 1,01...1,26, зависит от пяти масштабных факторов W_i и вычисляется по формуле:

$$B = 1,01 + 0,01 \sum_{i=1}^{5} W_i.$$

Общая характеристика масштабных факторов приведена в табл. 3.19, а табл. 3.20 позволяет определить оценки этих факторов. Оценки принимают 6 значений: от очень низкой (5) до сверхвысокой (0).

Таблица 3.19. Характеристика масштабных факторов

Масштабный фактор (W_i)	Пояснение
Предсказуемость PREC	Отражает предыдущий опыт организации в реализации проектов этого типа. Очень низкий означает отсутствие опыта. Сверхвысокий означает, что организация полностью знакома с этой прикладной областью

Масштабный фактор (W_i)	Пояснение
Гибкость разработки FLEX	Отражает степень гибкости процесса разработки. Очень низкий означает, что используется заданный процесс. Сверхвысокий означает, что клиент установил только общие цели
Разрешение архитектуры/риска RESL	Отражает степень выполняемого анализа риска. Очень низкий означает малый анализ. Сверхвысокий означает полный и сквозной анализ риска
Связность группы TEAM	Отражает, насколько хорошо разработчики группы знают друг друга и насколько удачно они совместно работают. Очень низкий означает очень трудные взаимодействия. Сверхвысокий означает интегрированную группу, без проблем взаимодействия
Зрелость процесса PMAT	Означает зрелость процесса в организации. Вычисление этого фактора может выполняться по вопроснику CMM

Таблица 3.20. Оценка масштабных факторов

Масштабный фактор (W_i)	Очень низкий 5	Низкий 4	Номинальный 3	Высокий 2	Очень высокий 1	Сверхвысокий 0
PREC	Полностью непредсказуемый проект	Главным образом, в значительной степени непредсказуемый	Отчасти непредсказуемый	Большей частью знакомый	В значительной степени знакомый	Полностью знакомый
FLEX	Точный, строгий процесс разработки	Редкое расслабление в работе	Некоторое расслабление в работе	Большей частью согласованный процесс	Некоторое согласование процесса	Заказчик определил только общие цели
RESL	Малое разрешение риска (20%)	Некоторое (40%)	Частое (60%)	Большей частью (75%)	Почти всегда (90%)	Полное (100%)
TEAM	Очень трудное взаимодействие	Достаточно трудное взаимодействие	Среднее взаимодействие	Главным образом кооперативность	Высокая кооперативность	Безукоризненное взаимодействие
PREC	Полностью непредсказуемый проект	В значительной степени непредсказуемый	Отчасти непредсказуемый	Большей частью знакомый	В значительной степени знакомый	Полностью знакомый
PMAT	Взвешенное среднее значение от количества ответов «Yes» на вопросник CMM Maturity					

В качестве иллюстрации рассмотрим компанию, которая берет проект в малознакомой предметной области. Положим, что заказчик не определил используемый процесс разработки и не допускает выделения времени на всесторонний анализ риска. Для реализации этой программной системы нужно создать новую группу разработчиков. Компания имеет возможности, соответствующие 2 уровню зрелости согласно модели CMM. Возможны следующие значения масштабных факторов:

- предсказуемость. Это новый проект для компании — значение Низкий (4);
- гибкость разработки. Заказчик требует некоторого согласования — значение Очень высокий (1);

- разрешение архитектуры/риска. Не выполняется анализ риска, как следствие, малое разрешение риска — значение Очень низкий (5);
- связность группы. Новая группа, нет информации, значение — Номинальный (3);
- зрелость процесса. Имеет место некоторое управление процессом — значение Номинальный (3).

Сумма этих значений равна 16, поэтому конечное значение степени $B = 1{,}17$.

Вернемся к обсуждению основного уравнения модели раннего этапа проектирования. Множитель поправки M_e зависит от набора формирователей затрат, перечисленных в табл. 3.21.

Таблица 3.21. Формирователи затрат для раннего этапа проектирования

Обозначение	Название
PERS	Возможности персонала (Personnel Capability)
RCPX	Надежность и сложность продукта (Product Reliability and Complexity)
RUSE	Требуемое повторное использование (Required Reuse)
PDIF	Трудность платформы (Platform Difficulty)
PREX	Опытность персонала (Personnel Experience)
FCIL	Средства поддержки (Facilities)
SCED	График (Schedule)

Для каждого формирователя затрат определяется оценка (по 6-бальной шкале), где 1 соответствует очень низкому значению, а 6 — сверхвысокому значению. На основе оценки для каждого формирователя по таблице Боэма определяется множитель затрат EM_i. Перемножение всех множителей затрат формирует множитель поправки:

$$M_e = \prod_{i=1}^{7} EM_i.$$

Слагаемое $ЗАТРАТЫ_{auto}$ используется, если некоторый процент программного кода генерируется автоматически. Поскольку производительность такой работы значительно выше, чем при ручной разработке кода, требуемые затраты вычисляются отдельно, по следующей формуле:

$$ЗАТРАТЫ_{auto} = (KALOC \times (AT/100))/ATPROD,$$

где

- $KALOC$ — количество строк автоматически генерируемого кода (в тысячах строк);
- AT — процент автоматически генерируемого кода (от всего кода системы);
- $ATPROD$ — производительность автоматической генерации кода.

Сомножитель AT в этой формуле позволяет учесть затраты на организацию взаимодействия автоматически генерируемого кода с оставшейся частью системы.

Далее затраты на автоматическую генерацию добавляются к затратам, вычисленным для кода, разработанного вручную.

Модель этапа пост-архитектуры

Модель этапа пост-архитектуры используется в период, когда уже сформирована архитектура и выполняется дальнейшая разработка программного продукта.

Основное уравнение пост-архитектурной модели является развитием уравнения предыдущей модели и имеет следующий вид:

$$ЗАТРАТЫ = A \times K_{\_req} \times РАЗМЕР^{B} \times M_{p} + ЗАТРАТЫ_{auto}\,[\text{чел.-мес.}],$$

где

- коэффициент $K_{\_req}$ учитывает возможные изменения в требованиях;
- показатель B отражает нелинейную зависимость затрат от размера проекта (размер выражается в KLOC), вычисляется так же, как и в предыдущей модели;
- в размере проекта различают две составляющие — новый код и повторно используемый код;
- множитель поправки M_p зависит от 17 факторов затрат, характеризующих продукт, аппаратуру, персонал и проект.

Изменчивость требований приводит к повторной работе, требуемой для учета предлагаемых изменений, оценка их влияния выполняется по формуле:

$$K_{\_req} = 1 + (BRAK/100),$$

где $BRAK$ — процент кода, отброшенного (модифицированного) из-за изменения требований.

Размер проекта и продукта определяют по выражению:

$$РАЗМЕР = РАЗМЕР_{new} + РАЗМЕР_{reuse}\,[KLOC],$$

где

- $РАЗМЕР_{new}$ — размер нового (создаваемого) программного кода;
- $РАЗМЕР_{reuse}$ — размер повторно используемого программного кода.

Формула для расчета размера повторно используемого кода записывается следующим образом:

$$РАЗМЕР_{reuse} = KASLOC \times ((100 - AT)/100) \times (AA + SU + 0.4\,DM + 0.3\,CM + 0.3\,IM)/100,$$

где

- $KASLOC$ — количество строк повторно используемого кода, который должен быть модифицирован (в тысячах строк);
- AT — процент автоматически генерируемого кода;
- DM — процент модифицируемых проектных моделей;
- CM — процент модифицируемого программного кода;

- *IM* — процент затрат на интеграцию, требуемых для подключения повторно используемого ПО;
- *SU* — фактор, основанный на стоимости понимания добавляемого ПО; изменяется от 50 (для сложного неструктурированного кода) до 10 (для хорошо написанного объектно-ориентированного кода);
- *AA* — фактор, который отражает стоимость решения о том, может ли ПО быть повторно используемым; зависит от размера требуемого тестирования и оценивания (величина изменяется от 0 до 8).

Правила выбора этих параметров приведены в руководстве по COCOMO II.

Для определения множителя поправки M_p основного уравнения используют 17 факторов затрат, которые могут быть разбиты на 4 категории. Перечислим факторы затрат, сгруппировав их по категориям.

Факторы продукта:

1) требуемая надежность ПО — *RELY*;
2) размер базы данных — *DATA*;
3) сложность продукта — *CPLX*;
4) требуемая повторная используемость — *RUSE*;
5) документирование требований жизненного цикла — *DOCU*.

Факторы платформы (виртуальной машины):

6) ограничения времени выполнения — *TIME*;
7) ограничения оперативной памяти — *STOR*;
8) изменчивость платформы — *PVOL*.

Факторы персонала:

9) возможности аналитика — *ACAP*;
10) возможности программиста — *PCAP*;
11) опыт работы с приложением — *AEXP*;
12) опыт работы с платформой — *PEXP*;
13) опыт работы с языком и утилитами — *LTEX*;
14) непрерывность персонала — *PCON*.

Факторы проекта:

15) использование программных утилит — *TOOL*;
16) мультисетевая разработка — *SITE*;
17) требуемый график разработки — *SCED*.

Для каждого фактора определяется оценка (по 6-бальной шкале). На основе оценки для каждого фактора по таблице Боэма определяется множитель затрат EM_i. Перемножение всех множителей затрат дает множитель поправки постархитектурной модели:

$$M_p = \prod_{i=1}^{17} EM_i .$$

Значение M_p отражает реальные условия выполнения программного проекта и позволяет троекратно увеличить (уменьшить) начальную оценку затрат.

ПРИМЕЧАНИЕ

Трудоемкость работы с факторами затрат минимизируется за счет использования специальных таблиц. Справочный материал для оценки факторов затрат приведен в приложении А.

От оценки затрат легко перейти к стоимости проекта. Переход выполняют по формуле:

$$СТОИМОСТЬ = ЗАТРАТЫ \times РАБ\_КОЭФ,$$

где среднее значение рабочего коэффициента составляет $15 000 за человеко-месяц.

После определения затрат и стоимости можно оценить длительность разработки. Модель COCOMO II содержит уравнение для оценки календарного времени *TDEV*, требуемого для выполнения проекта. Для моделей всех уровней справедливо:

$$Длительность(TDEV) = [3,0 \times (ЗАТРАТЫ)^{(0,33+0,2(B-1,01))}] \times SCEDPercentage / 100 [мес.],$$

где

- B — ранее рассчитанный показатель степени;
- *SCEDPercentage* — процент увеличения (уменьшения) номинального графика.

Если нужно определить номинальный график, то принимается *SCEDPercentage* = = *100* и правый сомножитель в уравнении обращается в единицу. Следует отметить, что COCOMO II ограничивает диапазон уплотнения/растягивания графика (от 75 до 160%). Причина проста — если планируемый график существенно отличается от номинального, это означает внесение в проект высокого риска.

Рассмотрим пример. Положим, что затраты на проект составляют 20 человеко-месяцев. Примем, что все масштабные факторы номинальны (имеют значения 3), поэтому, в соответствии с табл. 3.20, показатель степени B = 1,16. Отсюда следует, что номинальная длительность проекта равна:

$$TDEV = 3,0 \times (20)^{0,36} = 8,8 \text{ мес.}$$

Отметим, что зависимость между затратами и количеством разработчиков носит характер, существенно отличающийся от линейного. Очень часто увеличение количества разработчиков приводит к возрастанию затрат. В чем причина? Ответ прост:

- увеличивается время на взаимодействие и обучение сотрудников, согласование совместных решений;
- возрастает время на определение интерфейсов между частями программной системы.

Удвоение разработчиков не приводит к двукратному сокращению длительности проекта. Модель COCOMO II явно утверждает, что длительность проекта является функцией требуемых затрат, прямой зависимости от количества сотрудников нет.

Другими словами, она устраняет миф нерадивых менеджеров, что добавление людей поможет ликвидировать отставание в проекте.

COCOMO II предостерегает от определения потребного количества сотрудников путем деления затрат на длительность проекта. Такой упрощенный подход часто приводит к срыву работ. Реальная картина имеет другой характер. Количество людей, требуемых на этапе планирования и формирования требований, достаточно мало. На этапах проектирования и кодирования потребность в увеличении команды возрастает, после окончания кодирования и тестирования численность необходимых сотрудников достигает минимума.

Предварительная оценка программного проекта

В качестве иллюстрации применения методики оценки, изложенной в разделе «Выполнение оценки проекта на основе LOC- и FP-метрик», рассмотрим конкретный пример. Предположим, что поступил заказ от концерна «СУПЕРАВТО». Необходимо создать ПО для рабочей станции дизайнера автомобиля (РДА). Заказчик определил предметную область проекта в своей спецификации:

- ПО РДА должно формировать 2- и 3-мерные изображения для дизайнера;
- дизайнер должен вести диалог с РДА и управлять им с помощью стандартизованного графического пользовательского интерфейса;
- геометрические данные и прикладные данные должны содержаться в базе данных РДА;
- модули проектного анализа рабочей станции должны формировать данные для широкого класса дисплеев SVGA;
- ПО РДА должно управлять и вести диалог со следующими периферийными устройствами: мышь, дигитайзер (графический планшет для ручного ввода), плоттер (графопостроитель), сканер, струйный и лазерный принтеры.

Прежде всего надо детализировать предметную область. Следует выделить базовые функции ПО и очертить количественные границы. Очевидно, нужно определить, что такое «стандартизованный графический пользовательский интерфейс», каким должен быть размер и другие характеристики базы данных РДА и т. д.

Будем считать, что эта работа проделана и что идентифицированы следующие основные функции ПО:

1. Средства управления пользовательским интерфейсом СУПИ.
2. Анализ 2-мерной графики А2Г.
3. Анализ 3-мерной графики А3Г.
4. Управление базой данных УБД.
5. Средства компьютерной дисплейной графики КДГ.
6. Управление периферией УП.
7. Модули проектного анализа МПА.

Теперь нужно оценить каждую из функций количественно, с помощью LOC-оценки. По каждой функции эксперты предоставляют лучшее, худшее и веро-

ятное значения. Ожидаемую LOC-оценку реализации функции определяем по формуле

$$LOC_{ожi} = (LOC_{лучшi} + LOC_{худшi} + 4 \times LOC_{вероятнi}) / 6,$$

результаты расчетов заносим в табл. 3.22.

Таблица 3.22. Начальная таблица оценки проекта

Функция	Лучш. [LOC]	Вероят. [LOC]	Худш. [LOC]	Ожид. [LOC]	Уд. стоимость [$/LOC]	Стоимость [$]	Произв. [LOC/чел.-мес.]	Затраты [чел.-мес.]
СУПИ	1800	2400	2650	2340				
А2Г	4100	5200	7400	5380				
А3Г	4600	6900	8600	6800				
УБД	2950	3400	3600	3350				
КДГ	4050	4900	6200	4950				
УП	2000	2100	2450	2140				
МПА	6600	8500	9800	8400				
Итого				33 360				

Для определения удельной стоимости и производительности обратимся в архив фирмы, где хранятся данные метрического базиса, собранные по уже выполненным проектам. Предположим, что из метрического базиса извлечены данные по функциям-аналогам, представленные в табл. 3.23.

Таблица 3.23. Данные из метрического базиса фирмы

Функция	$LOC_{анi}$	$УД\_СТОИМОСТЬ_{анi}$ [$ / LOC]	$ПРОИЗВ_{анi}$ [LOC / чел.-мес.]
СУПИ	585	14	1260
А_Г	3000	20	440
УБД	1117	18	720
КДГ	2475	22	400
УП	214	28	1400
МПА	1400	18	1800

Видно, что наибольшую удельную стоимость имеет строка функции управления периферией (требуются специфические и конкретные знания по разнообразным периферийным устройствам), наименьшую удельную стоимость — строка функции управления пользовательским интерфейсом (применяются широко известные решения).

Считается, что удельная стоимость строки является константой и не изменяется от реализации к реализации. Следовательно, стоимость разработки каждой функции рассчитываем по формуле:

$$СТОИМОСТЬ_i = LOC_{ожi} \times УД\_СТОИМОСТЬ_{анi}.$$

Для вычисления производительности разработки каждой функции выберем самый точный подход — подход настраиваемой производительности:

$$ПРОИЗВ_i = ПРОИЗВ_{анi} \times (LOC_{анi}/LOC_{ожi}).$$

Соответственно, затраты на разработку каждой функции будем определять по выражению

$$ЗАТРАТЫ_i = (LOC_{ожi}/ПРОИЗВ_i)[\text{чел.-мес.}].$$

Теперь мы имеем все необходимые данные для завершения расчетов. Заполним до конца таблицу оценки нашего проекта (табл. 3.24).

Таблица 3.24. Конечная таблица оценки проекта

Функция	Лучш.	Вероят.	Худш.	Ожид. [LOC]	Уд. стоимость [$/LOC]	Стоимость [$]	Произв. [LOC/чел.-мес.]	Затраты [чел.-мес.]
СУПИ	1800	2400	2650	2340	14	32 760	315	7,4
А2Г	4100	5200	7400	5380	20	107 600	245	21,9
А3Г	4600	6900	8600	6800	20	136 000	194	35,0
УБД	2950	3400	3600	3350	18	60 300	240	13,9
КДГ	4050	4900	6200	4950	22	108 900	200	24,7
УП	2000	2100	2450	2140	28	59 920	140	15,2
МПА	6600	8500	9800	8400	18	151 200	300	28,0
Итого				33 360		$656 680		146

Учитывая важность полученных результатов, проверим расчеты с помощью FP-указателей. На данном этапе оценивания разумно допустить, что все информационные характеристики имеют средний уровень сложности. В этом случае результаты экспертной оценки принимают вид, представленный в табл. 3.25–3.26.

Таблица 3.25. Оценка информационных характеристик проекта

Характеристика		Лучш.	Вероят.	Худш.	Ожид.	Сложность	Количество
1	Вводы	20	24	30	24	×4	96
2	Выводы	12	15	22	16	×5	80
3	Запросы	16	22	28	22	×4	88
4	Логические файлы	4	4	5	4	×10	40
5	Интерфейсные файлы	2	2	3	2	×7	14
Общее количество							318

Таблица 3.26. Оценка системных параметров проекта

Коэффициенты регулировки сложности		Оценка
F_1	Передачи данных	2
F_2	Распределенная обработка данных	0
F_3	Производительность	4
F_4	Распространенность используемой конфигурации	3
F_5	Скорость транзакций	4
F_6	Оперативный ввод данных	5
F_7	Эффективность работы конечного пользователя	5
F_8	Оперативное обновление	3
F_9	Сложность обработки	5
F_{10}	Повторная используемость	4
F_{11}	Легкость инсталляции	3
F_{12}	Легкость эксплуатации	4
F_{13}	Разнообразные условия размещения	5
F_{14}	Простота изменений	5

Таким образом, получаем:

$$FP = Общее\ количество \times \left(0,65 + 0,01 \times \sum_{i=1}^{14} F_i\right) = 318 \times 1,17 = 372.$$

Используя значение производительности, взятое в метрическом базисе фирмы,

$$Производительность = 2,55\,[FP/\text{чел.-мес.}],$$

вычисляем значения затрат и стоимости:

$$Затраты = FP/Производительность = 145,9\,[\text{чел.-мес.}],$$

$$Стоимость = Затраты \times \$4500 = \$656\,500.$$

Итак, результаты проверки показали хорошую достоверность результатов. Но мы не будем останавливаться на достигнутом и организуем еще одну проверку, с помощью модели COCOMO II.

Примем, что все масштабные факторы и факторы затрат имеют номинальные значения. В силу этого показатель степени $B = 1,16$, а множитель поправки $M_p = 1$. Кроме того, будем считать, что автоматическая генерация кода и повторное использование компонентов не предусматриваются. Следовательно, мы вправе применить формулу

$$ЗАТРАТЫ = A \times РАЗМЕР^B\,[\text{чел.-мес.}]$$

и получаем:

$$ЗАТРАТЫ = 2,5(33,3)^{1,16} = 145,87 [\text{чел.-мес.}].$$

Соответственно, номинальная длительность проекта равна

$$Длительность = [3,0 \times (ЗАТРАТЫ)^{(0,33+0,2(B-1,01))}] = 3(145,87)^{0,36} = 18 [\text{мес.}].$$

Подведем итоги. Выполнена предварительная оценка программного проекта. Для минимизации риска оценивания использованы три методики, доказавшие корректность полученных результатов.

Анализ чувствительности программного проекта

COCOMO II – авторитетная и многоплановая модель, позволяющая решать самые разнообразные задачи управления программным проектом.

Рассмотрим возможности этой модели в задачах анализа чувствительности – чувствительности программного проекта к изменению условий разработки.

Будем считать, что корпорация «СверхМобильныеСвязи» заказала разработку ПО для встроенной космической системы обработки сообщений. Ожидаемый размер ПО – 10 KLOC, используется серийный микропроцессор. Примем, что масштабные факторы имеют номинальные значения (показатель степени B = 1,16) и что автоматическая генерация кода не предусматривается. К проведению разработки привлекаются главный аналитик и главный программист высокой квалификации, поэтому средняя оплата в команде составит $6000 в месяц. Команда имеет годовой опыт работы с этой предметной областью и полгода работает с нужной аппаратной платформой.

В терминах COCOMO II предметную область (область применения продукта) классифицируют как «операции с приборами» со следующим описанием: встроенная система для высокоскоростного мультиприоритетного обслуживания удаленных линий связи, обеспечивающая возможности диагностики.

Оценку пост-архитектурных факторов затрат для проекта сведем в табл. 3.27.

Таблица 3.27. Оценка пост-архитектурных факторов затрат

Фактор	Описание	Оценка	Множитель
RELY	Требуемая надежность ПО	Номинал.	1
DATA	Размер базы данных – 20 Кбайт	Низкая	0,93
CPLX	Сложность продукта	Очень высок.	1,3
RUSE	Требуемая повторная используемость	Номинал.	1
DOCU	Документирование жизненного цикла	Номинал.	1
TIME	Ограничения времени выполнения (70%)	Высокая	1,11
STOR	Ограничения оперативной памяти (45 Кбайт из 64 Кбайт, 70%)	Высокая	1,06
PVOL	Изменчивость платформы (каждые 6 мес.)	Номинал.	1
ACAP	Возможности аналитика (75%)	Высокая	0,83
PCAP	Возможности программиста (75%)	Высокая	0,87

Фактор	Описание	Оценка	Множитель
AEXP	Опыт работы с приложением (1 год)	Номинал.	1
PEXP	Опыт работы с платформой (6 мес.)	Низкая	1,12
LTEX	Опыт работы с языком и утилитами (1 год)	Номинал.	1
PCON	Непрерывность персонала (12% в год)	Номинал.	1
TOOL	Активное использование программных утилит	Высокая	0,86
SITE	Мультисетевая разработка (телефоны)	Низкая	1,1
SCED	Требуемый график разработки	Номинал.	1
Множитель поправки M_L			1,088

Из таблицы следует, что увеличение затрат в 1,3 раза из-за очень высокой сложности продукта уравновешивается их уменьшением вследствие высокой квалификации аналитика и программиста, а также активного использования программных утилит.

Рассчитаем затраты и стоимость проекта:

$$ЗАТРАТЫ = A \times РАЗМЕР^{B} \times M_p = 2,5(10)^{1,16} \times 1,088 = 36 \times 1,088 = 39[\text{чел.-мес.}],$$

$$СТОИМОСТЬ = ЗАТРАТЫ \times \$6000 = \$234\ 000.$$

Таковы стартовые условия программного проекта. А теперь обсудим несколько сценариев возможного развития событий.

Сценарий понижения зарплаты

Положим, что заказчик решил сэкономить на зарплате разработчиков. Рычаг — понижение квалификации аналитика и программиста. Соответственно зарплата сотрудников снижается до $5000. Оценки их возможностей становятся номинальными, а соответствующие множители затрат принимают единичные значения:

$$EM_{ACAP} = EM_{PCAP} = 1.$$

Следствием такого решения является возрастание множителя поправки $M_p = 1,507$, а также затрат и стоимости:

$$ЗАТРАТЫ = 36 \times 1,507 = 54[\text{чел.-мес.}],$$

$$СТОИМОСТЬ = ЗАТРАТЫ \times \$5000 = \$270\ 000,$$

$$Проигрыш\_в\_Стоимости = \$36\ 000.$$

Сценарий наращивания памяти

Положим, что разработчик предложил нарастить память — купить за $1000 чип ОЗУ емкостью 96 Кбайт (вместо 64 Кбайт). Это меняет ограничение памяти (используется не 70% а 47%), после чего фактор *STOR* снижается до номинального:

$$EM_{STOR} = 1 \rightarrow M_p = 1,026,$$

$$ЗАТРАТЫ = 36 \times 1,026 = 37 [\text{чел.-мес.}],$$

$$СТОИМОСТЬ = ЗАТРАТЫ \times \$6000 = \$222\,000,$$

$$Выигрыш\_в\_Стоимости = \$12\,000.$$

Сценарий использования нового микропроцессора

Положим, что заказчик предложил использовать новый, более дешевый МП (дешевле на $1000). К чему это приведет? Опыт работы с его языком и утилитами понижается от номинального до очень низкого и EM_{LTEX} = 1,22, а разработанные для него утилиты (компиляторы, ассемблеры и отладчики) примитивны и ненадежны (в результате фактор *TOOL* понижается от высокого до очень низкого и EM_{TOOL} = 1,24):

$$M_p = (1,088 / 0,86) \times 1,22 \times 1,24 = 1,914,$$

$$ЗАТРАТЫ = 36 \times 1,914 = 69 [\text{чел.-мес.}],$$

$$СТОИМОСТЬ = ЗАТРАТЫ \times \$6000 = \$414\,000,$$

$$Проигрыш\_в\_Стоимости = \$180\,000.$$

Сценарий уменьшения средств на завершение проекта

Положим, что к разработке принят сценарий с наращиванием памяти:

$$ЗАТРАТЫ = 36 \times 1,026 = 37 [\text{чел.-мес.}],$$

$$СТОИМОСТЬ = ЗАТРАТЫ \times \$6000 = \$222\,000.$$

Кроме того, предположим, что завершился этап анализа требований, на который было израсходовано $22 000 (10% от бюджета). После этого на завершение проекта осталось $200 000.

Допустим, что в этот момент «коварный» заказчик сообщает об отсутствии у него достаточных денежных средств и о предоставлении на завершение разработки только $170 000 (15%-ное уменьшение оплаты).

Для решения этой проблемы надо установить возможные изменения факторов затрат, позволяющие уменьшить оценку затрат на 15%.

Первое решение: уменьшение размера продукта (за счет исключения некоторых функций). Нам надо определить размер минимизированного продукта. Будем исходить из того, что затраты должны уменьшиться с 37 до 31,45 чел.-мес. Решим уравнение:

$$2,5 (НовыйРазмер)^{1,16} = 31,45 [\text{чел.-мес.}].$$

Очевидно, что

$$(НовыйРазмер)^{1,16} = 12,58,$$

$$(НовыйРазмер)^{1,16} = 12,58^{1/1,16} = 8,872[KLOC].$$

Другие решения:

- Уменьшить требуемую надежность с номинальной до низкой. Это сокращает стоимость проекта на 12% (EM_{RELY} изменяется с 1 до 0,88). Такое решение приведет к увеличению затрат и трудностей при применении и сопровождении.
- Повысить требования к квалификации аналитиков и программистов (с высоких до очень высоких). При этом стоимость проекта уменьшается на 15–19%. Благодаря программисту стоимость может уменьшиться на $(1 - 0,74/0,87) \times 100\% = 15\%$. Благодаря аналитику стоимость может понизиться на $(1 - 0,67/0,83) \times 100\% = 19\%$. Основная трудность — поиск специалистов такого класса (готовых работать за те же деньги).
- Повысить требования к опыту работы с приложением (с номинальных до очень высоких) или требования к опыту работы с платформой (с низких до высоких). Повышение опыта работы с приложением сокращает стоимость проекта на $(1 - 0,81) \times 100\% = 19\%$; повышение опыта работы с платформой сокращает стоимость проекта на $(1 - 0,88/1,12) \times 100\% = 21,4\%$. Основная трудность — поиск экспертов (специалистов такого класса).
- Повысить уровень мультисетевой разработки с низкого до высокого. При этом стоимость проекта уменьшается на $(1 - 0,92/1,1) \times 100\% = 16,4\%$.
- Ослабить требования к режиму работы в реальном времени. Предположим, что 70%-ное ограничение по времени выполнения связано с желанием заказчика обеспечить обработку одного сообщения за 2 мс. Если же заказчик согласится на увеличение среднего времени обработки с 2 до 3 мс, то ограничение по времени станет равно $(2 \text{ мс}/3 \text{ мс}) \times 70\% = 47\%$, в результате чего фактор *TIME* уменьшится с высокого до номинального, что приведет к экономии затрат на $(1 - 1/1,11) \times 100\% = 10\%$.
- Учет других факторов затрат не имеет смысла. Некоторые факторы (размер базы данных, ограничения оперативной памяти, требуемый график разработки) уже имеют минимальные значения, для других трудно ожидать быстрого улучшения (использование программных утилит, опыт работы с языком и утилитами), третьи имеют оптимальные значения (требуемая повторная используемость, документирование требований жизненного цикла). На некоторые разработчик почти не может повлиять (сложность продукта, изменчивость платформы). Наконец, житейские неожиданности едва ли позволят улучшить принятое значение фактора «непрерывность персонала».

Какое же решение следует выбрать? Наиболее целесообразное решение — исключение отдельных функций продукта. Вторым (по предпочтительности) решением является повышение уровня мультисетевой разработки (все равно это придется сделать в ближайшее время). В качестве третьего решения можно рассматривать ослабление требований к режиму работы в реальном времени. Принятие же других решений зависит от наличия необходимых специалистов или средств разработки.

Впрочем, окончательное решение должно выбираться в процессе переговоров с заказчиком, когда учитываются все соображения.

Выводы

1. Факторы затрат оказывают существенное влияние на выходные параметры программного проекта.
2. Модель COCOMO II предлагает широкий спектр факторов затрат, учитывающих большинство реальных ситуаций в «жизни» программного проекта.
3. Модель COCOMO II обеспечивает перевод качественного обоснования решения менеджера на количественные рельсы, тем самым повышая объективность принимаемого решения.

Контрольные вопросы и упражнения

1. Что такое метрика?
2. Что такое выполнение оценки программного проекта?
3. Какие размерно-ориентированные метрики вы знаете?
4. Для чего используют размерно-ориентированные метрики?
5. Определите достоинства и недостатки размерно-ориентированных метрик.
6. Что такое функциональный указатель?
7. От каких информационных характеристик зависит функциональный указатель?
8. Как вычисляется количество функциональных указателей?
9. Что такое коэффициенты регулировки сложности в метрике количества функциональных указателей?
10. Определите достоинства и недостатки функционально-ориентированных метрик.
11. Можно ли перейти от FP-оценок к LOC-оценкам?
12. Охарактеризуйте шаги оценки проекта на основе LOC- и FP-метрик. Чем отличается наиболее точный подход от наименее точного?
13. Что такое конструктивная модель стоимости? Для чего она применяется?
14. Чем отличается версия COCOMO 81 от версии COCOMO II?
15. В чем состоит назначение модели композиции? На каких оценках она базируется?
16. В чем состоит назначение модели раннего этапа проектирования?
17. Охарактеризуйте основное уравнение модели раннего этапа проектирования.
18. Охарактеризуйте масштабные факторы модели COCOMO II.
19. Как оцениваются масштабные факторы?
20. В чем состоит назначение модели этапа пост-архитектуры COCOMO II?
21. Чем отличается основное уравнение модели этапа пост-архитектуры от аналогичного уравнения модели раннего этапа проектирования?

22. Что такое факторы затрат модели этапа пост-архитектуры и как они вычисляются?
23. Как определяется длительность разработки в модели COCOMO II?
24. Что такое анализ чувствительности программного проекта?
25. Как применить модель COCOMO II к анализу чувствительности?
26. Рассчитайте значение FP-оценки для проекта «система для отслеживания успеваемости студентов».
27. С помощью модели COCOMO II проведите предварительную оценку проекта «система для отслеживания успеваемости студентов». Особое внимание обратите на оценку возможностей своей команды, сформированной в упражнении 37 второй главы.

Глава 4

Формирование и анализ требований

В этой главе рассматриваются разновидности требований, предъявляемых к программным системам, обсуждаются их характеристики и поясняются основные процессы для работы с требованиями: формирование требований, анализ требований, управление изменениями требований.

Виды требований к программному обеспечению

Требованиями (requirements) называют описание функциональных возможностей и ограничений, накладываемых на создаваемую программную систему. Обычно требования выражают, *что* система должна делать. Здесь не пытаются сформулировать, *как* добиться выполнения этих функций.

Например, возможно такое требование к банковской системе:

Система должна предоставить клиенту возможности выполнения следующих операций над его счетом: просмотр, снятие денег, добавление денег.

А вот такая запись требованием не является:

Информация банковского счета должна храниться в виде трех таблиц СУБД MySQL.

Здесь указывается *как* должна быть построена система, а не то, *что* она должна делать. Впрочем, могут быть и исключения из этого правила. Например, у заказчика могут быть особые причины для такой реализации.

Различают две категории представления требований: требования заказчика (первичные требования) и требования разработчика (детальные требования). Отличаются они друг от друга степенью проработки описаний. *Первичные требования* документируют желания и потребности заказчика и пишутся на языке, понятном заказчику. *Детальные требования* документируют требования в специальной,

структурированной форме, они детализированы по отношению к первичным требованиям.

Работу по созданию первичных требований будем называть сбором, или формированием требований. Проводится она на этапе подготовки жизненного цикла разработки.

Работу по созданию детальных требований будем называть анализом требований. Проводится она на этапе моделирования жизненного цикла разработки.

Некоторые проблемы могут быть порождены отсутствием четкого понимания различия между этими категориями требований. Поясним различие между ними.

Например, требование заказчика имеет вид:

ТРЕБОВАНИЯ ЗАКАЗЧИКА

1. ПО должно обеспечить средства для ввода и сохранения разнообразных данных абонента-пользователя,

а соответствующее ему требование разработчика может записываться в структурированной форме:

ТРЕБОВАНИЯ РАЗРАБОТЧИКА

1.1. Пользователь должен иметь возможность определять тип вводимых данных.

1.2. Для каждого типа данных должно иметься соответствующее средство, обеспечивающее ввод и сохранение элемента данных этого типа.

1.3. Каждый тип данных должен представляться соответствующей пиктограммой на дисплее пользователя.

1.4. Пользователю должна предлагаться пиктограмма для каждого типа данных. Кроме того, должна предлагаться возможность самостоятельного выбора пиктограммы для каждого типа данных.

1.5. При выборе пользователем пиктограммы типа данных к элементу данных должно быть применено средство, ассоциированное с указанным типом.

Таким образом, требования заказчика являются первичным описанием на естественном языке функций, выполняемых системой, и ограничений, накладываемых на нее. Дополнительно к ним могут прикладываться поясняющие диаграммы. Требования заказчика помещаются в *системную спецификацию.*

Требования разработчика содержат детализированное описание функций и ограничений системы, оформляемое в виде *спецификации анализа.* Она служит основой для заключения контракта между покупателем и разработчиками.

Весьма часто стандарты программной инженерии интегрируют обе спецификации в единый документ (рис. 4.1).

Различают два вида требований:

- *Функциональные требования.* Они описывают поведение системы и сервисы (функции), которые она должна выполнять. При этом исходят из всестороннего анализа проблемной (предметной) области. Рассматриваются разнообразные варианты поведения, определяемые различными данными и состояниями внешней среды.

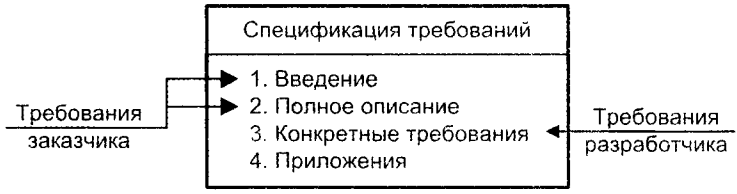


Рис. 4.1. Интеграция требований заказчика и разработчика в единую спецификацию требований

- *Нефункциональные требования.* Эти требования относятся к характеристикам системы и ее внешнего окружения. Дополнительно могут перечисляться ограничения, накладываемые на действия и функции системы, а также на условия разработки (ограничения по времени, ограничения на организацию проекта, стандарты и т. д.).

Функциональные требования задают работу, которую должна выполнять программная система, например:

> ПО будет вычислять стоимость товаров, находящихся в корзине пользователя.

С другой стороны, такое требование, как

> вычисление стоимости товаров должно производиться менее чем за одну секунду.

не является функциональным, поскольку оно не задает конкретную работу. Вместо этого оно оценивает работу, то есть определяет некоторое утверждение о работе.

Пример 4.1. Функциональные требования для ПО системы управления летательного аппарата имеют вид:

1. Система должна измерять проекции скорости (по трем каналам), вычислять отклонения от программных значений вектора скорости и радиус-вектора, а затем формировать управляющие воздействия, посылая их на исполнительные органы летательного аппарата (ЛА).
2. Система должна обеспечивать угловую стабилизацию движения ЛА: измерение углов (по трем каналам), вычисление отклонений углов от программных значений, формирование управляющих воздействий, распределение управляющих воздействий между исполнительными органами.
3. Каждая программа автопилота должна снабжаться уникальным идентификатором (Program_ID), который записывается в «черный ящик» вместе с параметрами полета.

Первое требование описывает цикл поведения системы. Второе требование определяет набор функций системы, а третье требование вводит комбинацию поведения и конкретного параметра системы.

Очевидно, что формы записи требований могут быть самыми разными. Однако требования должны быть:

- *ясными* (не допускать двоякого толкования, приводящего к искажению смысла пожеланий заказчика);
- *согласованными* (не содержать противоречивых и взаимоисключающих утверждений);
- *полными* (определять всю требуемую функциональность системы).

Нефункциональные требования не связаны непосредственно с функциями системы. Многие нефункциональные требования относятся к системе в целом, а не к отдельным ее элементам. Это означает, что они могут быть более критичными, чем единичные функциональные требования. Ошибка в функциональном требовании может понизить функциональные возможности системы, а ошибка в нефункциональном требовании может привести к отказу всей системы.

И. Соммервилл предложил выделять три группы нефункциональных требований [15]:

- *Требования к программной системе.* Описывают свойства и характеристики системы. Сюда относятся требования к скорости работы, производительности, емкости необходимой памяти, надежности, переносимости системы на разные компьютерные платформы и удобству эксплуатации.
- *Организационные требования.* Отображают вопросы работы и организации взаимодействия заказчика и разработчика. Они включают стандарты разработки программной системы, требования к методам и средствам разработки, указывают сроки создания и набор документации.
- *Внешние требования.* Учитывают факторы внешней среды. Они определяют требования по взаимодействию данной системы с внешним окружением, юридические обязательства, а также этические требования, гарантирующие приемлемость системы для пользователей.

Пример 4.2. Нефункциональные требования к производительности.

Требования к производительности определяют временные ограничения, которые должны быть выполнены в системе. Обсуждают ограничения по времени вычислений, периодичности вычислений, использованию оперативной памяти, использованию внешних устройств и т. д.:

Цикл регулирования скорости летательного аппарата должен укладываться в 64 мс.

Требования к производительности являются важной частью систем, работающих в реальном времени, в которых действия должны уложиться в определенные временные рамки. Примерами систем реального времени могут быть системы предотвращения столкновений, управления полетом.

Пример 4.3. Нефункциональные требования к надежности и доступности.

Требования надежности задают уровень надежности, определяя границы (или величину) неидеальной работы системы:

Система управления микроклиматом оранжереи должна давать не более двух ошибок в месяц.

Доступность близка по смыслу к надежности, оценивает степень доступности системы пользователям. Например:

Система продажи авиабилетов должна быть доступна пользователям 24 часа в сутки. Она может быть недоступна (находиться в состоянии профилактики) в течение 10 минут за 30-дневный период.

Пример 4.4. Нефункциональные требования ограничений.

Ограничения описывают границы характеристик или условий работы системы, впрочем, могут ограничиваться и условия разработки:

Система управления крылатой ракеты (СУ КР) должна рассчитывать координаты цели с точностью до трех метров.

Часто накладываются ограничения по инструментам и языкам. Они обусловлены сложившимися традициями организации, опытом программистов:

СУ КР должна быть разработана на языке Ada 2005.

Управление конфигурацией разработки должно проводиться в среде утилиты IBM Rational ClearCase.

Ограничение, требующее следовать определенному стандарту, часто определяется политикой фирмы или заказчиком:

Документация на СУ КР должна удовлетворять требованиям стандарта Mil.2011-35.

Проекты часто ограничены платформами, на которых они будут использоваться. Например:

Система управления крылатой ракеты должна работать на компьютерах Агат 1415 с расширением емкости оперативной памяти до 4 Гбайт.

Пример 4.5. Нефункциональные интерфейсные требования.

Интерфейсные требования описывают формат, в котором система общается с внешней средой:

Для передачи сообщений в систему телеметрии используется строковый формат out_tm <code>, где <code> -- двухбайтовый код из таблицы посылок Tajna_321b.

Формирование требований

Цель этой работы — сформировать требования заказчика.

Требования заказчика представляются в такой форме, что они понятны любому пользователю, не владеющему специальными техническими знаниями. Первичные требования должны определять только внешнее поведение ПО, без детализации структурной организации системы. Они записываются на естественном языке с использованием простых таблиц, а также наглядных диаграмм, рисунков.

Увы, естественный язык, по существу, является неоднозначным, поэтому простота формы приводит к проблемам содержания, например:

- *витиеватость стиля изложения.* Иногда нелегко выразить какую-то мысль на человеческом языке ясно и недвусмысленно, не сделав при этом текст многословным и трудночитаемым.
- *смешение и объединение требований.* В требованиях могут быть размыты границы между функциональными и нефункциональными требованиями. Несколько различных требований могут описываться как единое требование заказчика, и разработчик может сосредоточиться только на одном из них, «потеряв» другое.

Опишем шаги процесса формирования требований.

Шаг 1. Определение представителей заказчика. Важно выявить такой круг лиц, который позволит составить комплексное представление о портрете будущей системы: ее функциональности и полном перечне характеристик. Иногда это сделать довольно сложно. В любом случае надо разобраться в предметной области системы, выявить круг пользователей и других заинтересованных лиц, в состав которых может войти руководство заказчика (и даже руководство разработчиков), обслуживающий персонал и т. д.

Шаг 2. Проведение опроса представителей заказчика. Поскольку обычно имеется несколько заинтересованных лиц, первый вопрос — решить, в каком порядке их опрашивать. Очевидно, нужно расставить приоритеты в списке опрашиваемых (согласно вашим предположениям о важности получаемых сведений). Далее планируется время и длительность опросов, на которых должны присутствовать как минимум два члена команды разработчиков и имеются средства записи разговоров (диктофоны, камеры и т. д.).

Во время интервью следует сконцентрироваться на слушании, участвовать в диалоге (надо задавать вопросы и мотивировать собеседника, уточнять пожелания и потребности, предлагать варианты поведения и использования системы), делать подробные заметки. В конце опроса следует запланировать следующую встречу.

Пара разработчиков, участвующих в опросе и поддерживающих друг друга, обеспечивают возможность оперативной формулировки «трудных» требований со слов заказчика. Часто заказчик сам формулирует требования по ходу разговора, но иногда нуждается в помощи. В этих случаях разработчик и заказчик совместно «шлифуют» концепцию требования. Иными словами, заказчику бывают необходимы подсказки для формирования концепции.

Эффективным способом получения и формулировки требований являются примеры. Эти примеры предлагаются разработчиками, как правило, в графической форме.

Шаг 3. Документирование результатов опроса. После каждого опроса создается черновик — письменная форма набора требований. Она отсылается заказчикам для комментариев и коррекции. Как правило, затем проводится серия повторных опросов. Завершается серия опросов собранием. По итогам собрания готовится документ, содержащий все требования и представляемый в формате стандарта, выбранного для спецификации требований. Этот документ утверждается заказчиком.

Шаг 4. Проверка требований. Цель проверки спецификации требований состоит в оценке правильности определений, которые в ней содержатся. Проверка гарантирует, что все положения требований корректны, отражают желаемые

характеристики и удовлетворяют потребностям заказчика. Может оказаться, что требования, которые в спецификации выглядели превосходно, при реализации чреваты проблемами. Разработчики испытывают серьезный дискомфорт при реализации неясных или неполных требований. Поскольку у них нет необходимой информации, они вынуждены ориентироваться на собственные предположения, которые не всегда верны. Доказано, что исправление ошибок в требованиях, работа над которыми уже завершена, требует очень много усилий. Исследования показали: исправлять ошибки требований в конце разработки системы в 100 раз дороже, чем в ходе формирования этих требований. Многократно подтвержден следующий постулат: «любые усилия, затраченные на выявление ошибок в спецификации требований, сэкономят реальные время и деньги».

Проверка требований выполняется заказчиком и разработчиком совместно, она удостоверяет:

1) предметная область проекта описана корректно;
2) разработчик и заказчик имеют одинаковые представления о целях системы;
3) анализ внешней среды и риска разработки подтверждает возможность создания системы;
4) спецификация требований верно описывает желаемую функциональность и характеристики системы, которые соответствуют потребностям заказчика и других заинтересованных лиц;
5) требования полные и качественные;
6) все требования согласованы друг с другом, не содержат противоречий;
7) требования обеспечивают реальную возможность разработки системы.

Проверка должна подтвердить, что в спецификации присутствуют только качественные требования как по содержанию (корректные, полные, согласованные, осуществимые и поддающиеся проверке), так и по форме записи (ясные, легко модифицируемые и поддающиеся отслеживанию).

Разумеется, проверить можно только задокументированные, а не воображаемые требования. Дело в том, что многие организации не справляются с формированием требований. Это не значит, что они не пользуются требованиями — просто требования существуют лишь в головах конкретных разработчиков. В итоге крах программного проекта становится почти неизбежным. Еще одна проблема создается организациями, которые формируют лишь часть требований — требования для начальной итерации разработки. С этой порцией требований начинается проект, а вот изменения в спецификации требований для последующих итераций уже не поддерживаются. Конечно, сопровождение спецификации требований организовать нелегко, но это ведь необходимо!

Анализ требований

Формирование требований является лишь начальной фазой работы с требованиями.

Анализ требований рассматривает требования заказчика как исходные данные, на выходе анализа — требования разработчика, которые справедливо называют *детальными требованиями*. Анализ требований служит мостом между подготовкой-

планированием и проектированием ПО. Здесь происходит переход из мира заказчика в мир разработчика. Меняется язык записи требований. Теперь это не естественный язык человека, а язык формализованных моделей. Он непонятен заказчику, но мил и близок разработчику. С помощью этих моделей можно добиться более точного отображения множества деталей, присущих программной системе. Причем «малой кровью», то есть минимальным набором выразительных средств.

Конечно, эти модели тоже отвечают принципам построения требований: показывать *«что»* надо делать, а не *«как»* это делается. Но, с другой стороны, для полного описания системы требуется такая детализация, которая должна включать информацию об организации системы на уровне архитектуры. Почему?

Ответим на этот вопрос. Уровень архитектуры несет в себе лишь эскиз структурной организации системы. В этом эскизе прописаны далеко не все детали структуры, но крупные части подсистемы уже обозначены. Эти подсистемы просто необходимы для конкретизации максимально полного набора требований, к ним привязываются, их адресуют отдельные детальные требования. Практически элементы архитектуры позволяют провести дальнейшее структурирование спецификации требований. Образно говоря, детальные требования привязываются к «скелету» будущей системы. Кстати, во многих случаях требования *должны* задавать конкретный образец архитектуры.

Разработчикам программных систем нужен базис для проектирования и конструирования. Этот базис и образуется набором детальных требований, которые подробно, полно и согласованно описывают свойства и функциональность системы. Каждое из этих требований нумеруется и отслеживается по ходу разработки. Реализация каждого требования тестируется. Ясно, что детальные требования должны «проистекать» из первичных требований.

Детальные требования ориентированы на чтение разработчиками, написаны они на их «родном» языке. Правда, заказчики тоже могут их оценить (с помощью разработчиков). Несправедливо? Вспомним, что основная аудитория для первичных требований состоит из заказчиков.

Здесь нет никакой дискриминации, так нужно для дела. Язык представления должен обеспечивать компактную, точную и выразительную запись огромного количества подробных требований. Иначе говоря, язык представления должен быть адекватен решаемой задаче. Ведь формулирование *всех* требований разработчика со всеми деталями является, по своей сути, очень сложной задачей, задающей уровень качества дальнейшей разработки ПО.

Рассмотрим типичные шаги анализа требований.

Шаг 1. Организация первичных требований. Необходимость этой работы обусловлена большим количеством требований. По мере их разрастания неорганизованный список быстро превращается в неуправляемый. Стандарты рекомендуют несколько способов организации:

- по *режиму*. Некоторые системы меняют поведение в зависимости от режима работы. Например, система управления может иметь различные наборы функций в зависимости от режима: *обучение*, *нормальный режим* или *аварийный режим*;
- по *категориям пользователей*. Некоторые системы обеспечивают различные наборы функций для разных категорий пользователей. Например, система

управления лифтом предоставляет различные возможности для пассажиров, обслуживающего персонала и пожарных;

- по *объектам*. Объекты — это программные сущности системы, которые могут иметь физические аналоги во внешней среде. Например, в системе контроля за пациентом объекты включают пациентов, датчики, медсестер, помещения, врачей, лекарства. Каждый объект несет в себе набор данных и функций. Функции объектов также называют услугами, методами, операциями;
- по *свойствам*. Свойство — сервис, предоставляемый внешней среде, определяется с помощью пар «входное воздействие — реакция». Например, в телефонной системе свойства включают локальный вызов, переадресацию вызовов и циркулярный вызов. Заметьте, что «реакция» может быть рассредоточена по различным частям программной системы;
- *по стимулам*. Некоторые системы легко организуются при описании их функций на языке стимулов. Например, функции автоматической системы посадки самолета могут быть организованы в разделы по энергетическим потерям, сдвигу ветра, изменению направления качения, избыточной вертикальной скорости и т. д.;
- по *откликам*. Некоторые системы организуются посредством описания всех функций, поддерживающих генерацию различных откликов. Например, функции системы учета персонала могут быть организованы в разделы, соответствующие всем функциям для составления чека по оплате, всем функциям для составления списка служащих и т. д.;
- по *иерархии функций*. То есть путем разделения ПО на множество высокоуровневых функций и последующего разбиения их на подфункции. Например, требования для ПО домашнего бюджета можно разбить на функции проверки, функции сбережений и функции инвестирования. Функции проверки могут затем быть разложены на функции чековой книжки, баланса счета, функцию составления отчетов и т. д. Это классический способ упорядочения требований. Иерархия функций может быть образована по общим вводам, общим выводам, или доступу к общим данным. Связи между функциями и данными можно показывать с помощью диаграмм потоков данных и словарей данных.

При рассмотрении системы могут оказаться применимыми несколько классификационных признаков организации. В таких случаях можно организовать конкретные требования в виде нескольких иерархий, построенных по 2–3-м признакам. Например, возможна организация одновременно по категориям пользователей и свойствам.

Выбор организации требований автоматически влияет на выбор языка записи требований — формализованного метода и моделей анализа. Например, при организации требований по режимам полезны конечные автоматы или диаграммы состояний; при организации по объектам — методы объектно-ориентированного анализа; при организации по свойствам — последовательности *входное воздействие — реакция*; а при организации по иерархии функций — диаграммы потоков данных и словари данных. Эти методы будут рассмотрены в последующих главах.

Шаг 2. Преобразование первичных требований в детальные требования. Обычно одно требование заказчика преобразуется в несколько детальных требований, хотя

возможно и отображение «один в один». Рекомендации по работе с детальным требованием:

1. Первичная оценка. Возможны варианты: 1) это функциональное требование — соответствует реализующему методу; 2) слишком большое — трудно управлять, следует разделить на части; 3) слишком маленькое — нет смысла рассматривать отдельно, надо присоединить к другому требованию.
2. Обеспечение прослеживаемости требования. Анализ возможности прослеживания при проектировании и конструировании.
3. Обеспечение тестируемости требования. Написание тестов для проверки реализации требования. Продумываются варианты как положительного, так и отрицательного исхода тестов.
4. Анализ однозначности толкования требования.
5. Назначение приоритета требования. Выбираются варианты: существенное, желательное или необязательное.
6. Проверка полноты требования. Следует убедиться в наличии всех «обеспечивающих» требований.
7. Проверка согласованности требования с другими требованиями. Анализируются и устраняются возможные противоречия.
8. Требование заносится в спецификацию анализа (спецификацию требований).
9. Описание и обоснование желаемых характеристик детального требования приведено в следующем разделе.

Шаг 3. Аттестация детальных требований. Аттестация (валидация) должна подтвердить, что требования действительно определяют ту систему, которая нужна заказчику. Аттестация очень важна, так как ошибки требований могут привести к существенной переделке системы и большим затратам, если будут обнаружены при разработке или эксплуатации системы. Содержательно в состав аттестации входят:

- *Проверка правильности требований.* Заказчик считает, что система необходима для выполнения заданных им функций. Однако обсуждение с заинтересованными лицами и последующий анализ могут выявить дополнительные функции, и их тоже надо учесть.
- *Проверка на непротиворечивость.* Требования не должны определять противоречивые факты. Иначе говоря, в требованиях не должно быть противоречащих друг другу ограничений или различных определений одной и той же функции.
- *Проверка на полноту.* Требования должны описывать все необходимые функции и ограничения системы.
- *Проверка на выполнимость.* Здесь анализируется реализуемость требований в рамках временных и бюджетных ограничений проекта.

В ходе аттестации применяют следующие методы:

1. *Совместные проверки требований.* Требования анализируются рецензентами, избираемыми из числа заинтересованных лиц и внешних экспертов. Цель — найти неточности и ошибки. Обнаруженные противоречия, ошибки и упуще-

ния в требованиях документируются и передаются заказчику и разработчикам системы.

2. *Макетирование.* Макет создается экспертами и обсуждается заказчиком и разработчиками. Содержание макетирования описано в главе 1.
3. *Генерация тестов.* Создание тестов для требований очень часто выявляет проблемы в описании требований. Проблемы при создании тестов означают, что требования трудно выполнить и поэтому нужно их пересмотреть.
4. *Автоматизированная проверка непротиворечивости.* Если требования представлены как формализованные модели, для проверки непротиворечивости моделей можно применить программные CASE-утилиты.

Аттестация требований – дорогое и сложное занятие, требующее высокой квалификации, хорошего кругозора и развитого воображения. Редко удается выявить все проблемы требований, поэтому возвращаться к аттестации приходится многократно.

Желаемые характеристики детального требования

Как отмечалось в предыдущем разделе, в ходе создания детальных требований ориентируются на формирование целого набора желаемых характеристик. Обсудим эти характеристики более подробно.

Прослеживание. Прослеживанием называют отображение каждого требования на соответствующие части проекта и системы. Например, как показано на рис. 4.2, каждое функциональное детальное требование отображают на артефакты разработки. Здесь показаны артефакты проекта, совместно обеспечивающие возможность контроля.

По мере продвижения проекта должна сохраняться согласованность спецификации требований с архитектурой системы и программным кодом. Если требования сложно отследить в архитектуре и коде, разработчики частенько избегают обновления спецификации (при внесении изменений в исходный код). Мотив: для этого потребуется много усилий. В конце концов, такая «забывчивость» приводит к увеличению времени и затрат на разработку.

Менеджеры должны исключить такие ситуации, добиваясь ясной и простой ссылки друг на друга и подчеркивая обязательность процедуры документирования. Когда код, реализующий требование, находится в нескольких местах, прослеживание достигается с помощью *матрицы прослеживания требований* (табл. 4.1).

Таблица 4.1. Матрица прослеживания требований

Требование	Модуль 1	Модуль 2	Модуль 3
M_1415	showData()	AverageMark()	getInfo()
M_1416	showGroup()	showSemester()	showData()

Как показано в табл. 4.1, требование M_1415 реализовано посредством функций *showData()* в модуле 1, *AverageMark()* в модуле 2 и *getInfo()* в модуле 3. Изменение в этом требовании повлечет изменение в одном или нескольких из этих функций.

Рис. 4.2. Прослеживание детального требования (стрелки показывают связи прослеживания)

Это следует четко контролировать, поскольку эти функции могут участвовать также и в выполнении других требований (например, *showData()* используется еще и для реализации требования M_1416). В результате изменения, сделанные для удовлетворения одного требования, могут нарушить другое. Поскольку с отношениями «многие-ко-многим» трудно работать, стараются привести отображение между требованием и функцией к категории «один-к-одному».

Мы хотим, чтобы каждое детальное требование можно было проследить *прямо и обратно*. Предыдущие обсуждения касались прямого отслеживания от детальных требований до реализации. Обратное отслеживание детальных требований означает, что требование представляет собой четкую последовательность одного или нескольких первичных требований. Например, детальное требование

> Управляющее воздействие по каналу тангажа должно формироваться каждые 32 мс.

может быть прослежено назад к следующему первичному требованию из примера 4.1 (полагаем, что оно входило в ту часть спецификации требований, где записаны требования заказчика):

> Система должна обеспечивать угловую стабилизацию движения ЛА: измерение углов (по трем каналам), вычисление отклонений углов от программных значений, формирование управляющих воздействий, распределение управляющих воздействий между исполнительными органами.

Такой обратный контроль является основой проверки детальных требований.

Возможность полного прослеживания означает, что каждое детальное требование связано с конкретным элементом программной системы, а также с тестом элемента (рис. 4.3). Рисунок демонстрирует преимущество жесткого соответствия между каждым отдельным функциональным требованием, описывающей требование частью проектного решения и реализующей его частью программного кода. Они связаны с тестом реализации требования.

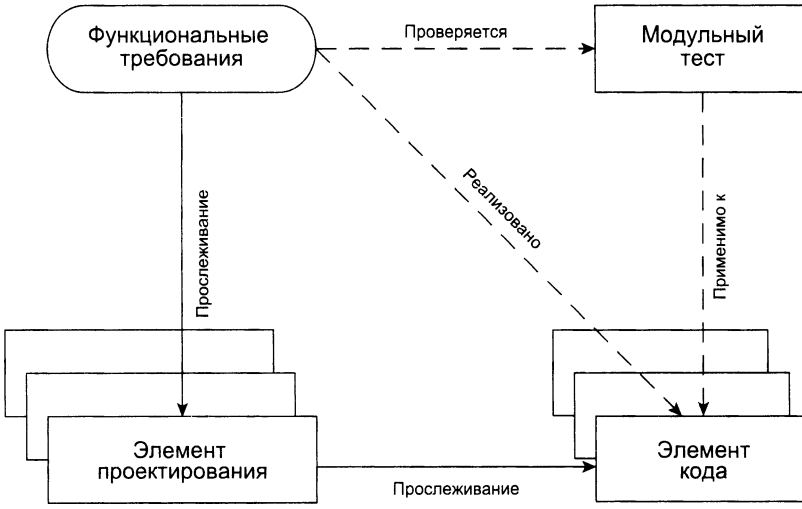


Рис. 4.3. Прослеживание и тестирование функциональных детальных требований

Отметим, что прослеживать *нефункциональное* требование обычно труднее, поскольку ему соответствуют несколько частей проектного решения и реализации.

Тестируемость. Должна существовать возможность протестировать реализацию требования. Рассмотрим требование:

Коэффициент интеллекта сотрудника должен превышать средний уровень.

Это нетестируемое требование, поскольку средний уровень не задан. Его нужно модифицировать, например, так:

Коэффициент интеллекта сотрудника должен превышать средний уровень выпускника американского вуза (115 единиц).

Готовность к тестированию будет полной, если для требования определено «свидетельство об ошибке». Появление такого свидетельства говорит о некорректности реализации требования (или о некорректных условиях работы). Многие требования зависят от конкретных данных, и надо описывать, как реализация требования должна работать в случае неправильных данных.

Однозначность. Если требование записано нечетко или двусмысленно, нельзя определить правильность его реализации. Следующее требование неоднозначно:

Центральный процессор космического самолета должен быть идеален.

Что означает здесь «идеал»? Самый производительный, или самый надежный, или совместимый по системе команд с ЦП корабля Space Shuttle? А может, нужна какая-то комбинация этих показателей?

Однозначное требование может быть записано в следующей форме:

Центральный процессор космического самолета должен иметь следующие параметры: среднее быстродействие на задачах управления полетом не менее 2 млн оп./ с, вероятность безотказной работы не менее 0,9999, рабочий диапазон температур –80 °С ... +140 °С, общая доза радиации 1000 крад.

Приоритетность. Часто бывает трудно реализовать все запланированные требования в срок и не выходя за рамки бюджета. В этом случае уменьшают количество реализованных требований. Одна из процедур отсеивания базируется на расстановке приоритетов между детальными требованиями. Обычно выделяют три категории требований: *существенные*, *желательные* и *необязательные*. Порядок сокращения прост: вначале сокращают необязательные требования, при необходимости переходят к категории желательных, стараясь сохранить категорию существенных требований. Оптимизм сокращения питается правилом Парето — 20% требований реализуют 80% желаемой функциональности.

Полнота. Полнота набора требований является гарантией охвата всей функциональности и установки всех характеристик системы. Оценивают полноту эксперты предметной области. Их работа может приводить к добавлению недостающих требований.

Согласованность. Набор требований согласован, если между требованиями нет противоречий. По мере роста числа требований вероятность противоречий возрастает:

Тр_221. Для повышения надежности следует вводить многоканальную избыточность вычислений основных управляющих воздействий и их голосование перед подачей на исполнительные органы ЛА.

...

Тр_14247. Для повышения скорости отработки возмущений не следует использовать схемы голосования управляющих воздействий в каналах угловой стабилизации и стабилизации центра масс ЛА.

Организация детальных требований помогает минимизировать противоречия благодаря группировкам родственных требований. Однако и при такой организации противоречия возможны, поэтому следует проверять согласованность наряду с другими характеристиками.

Спецификация требований

Спецификация требований — это документ, являющийся официальным предписанием для разработчиков ПО. Он содержит описание требований заказчика (первичных требований) и разработчика (детальных требований). Первичные требования

документируются при формировании требований, а детальные требования — при выполнении анализа требований.

Многие организации разрабатывали стандарты документирования требований. Наиболее полным и авторитетным считают стандарт Института инженеров по электротехнике и радиоэлектронике IEEE Std 830-1998, который должен помочь:

- заказчикам программного обеспечения точно описать, что они хотят получить;
- разработчикам программного обеспечения точно понять, что хочет заказчик.

Этот стандарт утверждает, что качественно составленная спецификация (SRS) должна принести заказчикам, разработчикам и другим лицам определенные выгоды:

- *Создать основу для соглашения между заказчиками и разработчиками по поводу функций, которые должен выполнять программный продукт.* Полное описание функций ПО, приведенное в SRS, поможет потенциальным пользователям определить, отвечает ли ПО их потребностям или как необходимо изменить ПО, чтобы удовлетворить эти потребности.
- *Уменьшить объем работ по разработке.* Подготовка SRS вынуждает заказчика рассмотреть все требования до начала проекта и сокращает последующее повторное проектирование, кодирование и тестирование. Тщательный анализ требований, указанных в SRS, может вскрыть упущения, неправильное понимание и противоречия на ранних стадиях цикла разработки, когда эти проблемы проще исправить.
- *Обеспечить основу для оценки расходов и графиков работ.* Описание продукта, разрабатываемого в соответствии с SRS, является практической основой для оценки затрат на проект и может использоваться для формирования контракта и бюджетных ограничений.
- *Обеспечить основу для аттестации (валидации) и верификации.* При использовании качественной SRS организации могут повысить эффективность планов аттестации и верификации. Как часть контракта на разработку, SRS обеспечивает основу для проведения проверок.
- *Облегчить передачу пользователям.* SRS делает более простой передачу программного изделия новым пользователям или его установку на новых компьютерах. Таким образом, для заказчиков упрощается передача ПО другим подразделениям их организации, а для разработчиков упрощается передача ПО новым заказчикам.
- *Служить в качестве основы для расширения.* Поскольку в SRS обсуждается продукт, а не проект, в котором он разработан, то SRS служит основой для последующего расширения готового продукта. SRS может несколько измениться, но останется базисом для дальнейшей эволюции продукта.

Данный стандарт предлагает следующую структуру спецификации требований к программному обеспечению (SRS).

1. **Введение.**

   1.1. Назначение (назначение спецификации, аудитория).

1.2. Область действия (название ПО, его задачи, применение).

1.3. Определения и сокращения (терминология).

1.4. Публикации (список литературы).

1.5. Краткий обзор (характеристика всех разделов).

2. **Полное описание.**

2.1. Перспектива изделия (оценка продукта, связи с другими продуктами).

2.2. Функции изделия (основные функции продукта).

2.3. Характеристики пользователя (общие характеристики пользователей продукта).

2.4. Ограничения (ограничения возможностей разработчика).

2.5. Допущения и зависимости (факторы, влияющие на требования).

2.6. Распределение требований (требования, которые откладываются до появления будущих версий продукта).

3. **Конкретные требования** (охватывают функциональные, нефункциональные и интерфейсные требования. Это наиболее значимая часть документа, описывает детальные требования, организованные выбранным способом).

4. **Приложения** (оценка себестоимости, форматы ввода-вывода, проблемы).

5. **Алфавитный указатель.**

В стандарте указано, что он может использоваться частными организациями и для создания собственных стандартов по требованиям.

Управление требованиями

Зададим вопрос: меняются ли требования в жизненном цикле разработки программной системы? Да, да, да. Меняются. И процесс этот имеет объективную основу. После создания начальной версии требований приходит новое, более глубокое понимание предметной области ПО, прорисовываются новые детали, которые раньше были незаметны. Отсюда новый виток работы над требованиями. Постоянство требований — это, скорее, исключение из общего правила. Чтобы изменения требований не похоронили проект раньше времени, процессом изменения требований надо управлять.

В ходе управления требованиями нужно решить ряд вопросов:

- *Распознавание и учет требований.* Каждое требование должно быть индивидуально учтено, поскольку оно может пересекаться с другими требованиями и использоваться в оценках трассировки.
- *Управление внесением изменений.* Должна предусматриваться последовательность защитных действий для оценки воздействия изменения и стоимости изменения.
- *Стратегия трассировки.* Существуют зависимости между требованиями, а также между требованиями и проектными решениями системы. Трассировка должна обнаруживать зависимые требования, запоминать эти зависимости и отслеживать влияние требований друг на друга и на проектные решения.

Управление требованиями нуждается в автоматизированной поддержке, которую обеспечивают программные утилиты. Утилиты поддерживают следующие действия:

- *Хранение требований.* Информация о требованиях должна сохраняться в защищенной управляемой памяти, доступной для участников процесса разработки требований.
- *Реализация цикла изменения требования.* Изменение требования происходит в интерактивном режиме: организуется диалог сотрудника с программной утилитой.
- *Управление трассировкой.* Утилита обнаруживает зависимые требования, выполняя действия трассировки.

Опишем шаги процесса управления изменениями.

Шаг 1. Распознавание проблемы. Фиксируется проблема в требованиях или прямой запрос на внесение изменения. Проверяется обоснованность проблемы или запроса. Если обоснованность подтверждена, переходят к следующему шагу. В противном случае процесс прекращается.

Шаг 2. Анализ изменения. При определении возможности изменения исходят из информации трассировки и общих представлений о требованиях к системе. Стоимость изменения определяется двумя параметрами: стоимостью изменения спецификации и (если это необходимо) стоимостью изменения проектного решения системы и программного кода. По окончании анализа принимается решение об изменении требования.

Шаг 3. Выполнение изменения. Вносится изменение в спецификацию требований и, если необходимо, в проектное решение и программный код. Спецификация требований должна быть организована так, чтобы внесение изменения носило локальный характер и не потребовало реорганизации всего документа. Как и в случае программ, изменяемость документов достигается минимизацией внешних ссылок и обеспечением модульности разделов. Это означает, что разделы могут изменяться без влияния на остальные части документа.

ВНИМАНИЕ

Всегда существует соблазн внести сначала изменение в программную систему, а лишь затем изменить документ — спецификацию требований. Не поддавайтесь на провокацию! В этом случае «рассинхронизация» ПО и требований почти неизбежна!

В гибких процессах разработки, например в экстремальном программировании, требования меняются в течение всего процесса разработки. В этих условиях, когда заказчик предлагает изменение требований, эти изменения не проходят через официальный процесс управления изменениями. Напротив, заказчик присваивает изменению приоритет, и если он высокий, то заказчик решает, что свойства системы, запланированные на следующую итерацию, должны быть отброшены.

Контрольные вопросы и упражнения

1. Определите и опишите четыре вида требований к ПО.
2. Определите двусмысленности и пропуски в следующем описании требований к системе продажи билетов: Автоматизированная система продает авиабилеты.

Клиент указывает аэропорт назначения, вставляет кредитную карточку и вводит PIN-код. Система выдает билет и снимает с кредитной карточки сумму, равную стоимости полета в указанный пункт. Когда клиент нажимает кнопку выбора, отображается меню возможных аэропортов назначения и предлагается выбрать пункт назначения. После выбора аэропорта назначения клиенту предлагается вставить кредитную карточку. Затем проверяется кредитная карточка, после чего пользователю предлагается ввести PIN-код. По окончании операций с кредитной карточкой выдается билет.

3. Запишите нефункциональные требования для описанной в пункте 2 системы, характеризующие надежность системы и время отклика.
4. Запишите на естественном языке требования заказчика для следующих функций: функция выдачи денег в банкомате; функция контроля грамматики и исправления ошибок в текстовом процессоре.
5. Предложите способ прослеживания в спецификации требований зависимостей между функциональными и нефункциональными требованиями.
6. Какие разновидности нефункциональных требований вы знаете?
7. Используя ваши представления о банкомате, создайте для него требования заказчика.
8. Сравните процессы формирования требований и анализа требований. Чем они схожи? В чем отличаются? Поясните причины отличий.
9. В чем различия требований заказчика и требований разработчика?
10. В чем различия детальных требований и требований разработчика?
11. Прокомментируйте желаемые характеристики детальных требований.
12. Что надо сделать для обеспечения тестируемости требования?
13. Зачем вводится приоритетность требований? Какие уровни приоритетов вы знаете?
14. Какие способы организации детальных требований вы знаете? Опишите главные идеи этих способов.
15. Поясните задачи, решаемые в ходе аттестации требований.
16. Поясните методы, применяемые при аттестации требований.
17. Предложите, кто бы мог участвовать в формировании требований для автоматизированной системы подготовки семейного праздника. Объясните, почему требования разных лиц-заказчиков будут противоречивы.
18. В гибких процессах разработки возможны ситуации, когда изменения в систему вносятся прежде, чем изменения в требованиях будут утверждены. Предложите методику срочного внесения изменений в систему, которая гарантирует согласованность системы и спецификации требований.

Глава 5

Классические методы анализа

В этой главе рассматриваются классические методы анализа требований, *ориентированные на процедурную* реализацию программных систем. Анализ требований служит мостом между неформальным описанием требований, выполняемым заказчиком, и проектированием системы. Методы анализа призваны формализовать обязанности системы, фактически их применение дает ответ на вопрос «Что должна делать будущая система?»

Структурный анализ

Структурный анализ — один из формализованных методов анализа требований к ПО. Автор этого метода — Том Де Марко (1979) [47]. В этом методе программное изделие рассматривается как преобразователь информационного потока данных. Основной элемент структурного анализа — диаграмма потоков данных.

Диаграммы потоков данных

Диаграмма потоков данных ПДД — графическое средство для изображения информационного потока и преобразований, которым подвергаются данные при движении от входа к выходу системы. Элементы диаграммы имеют вид, показанный на рис. 5.1.

Диаграмма может использоваться для представления программного изделия на любом уровне абстракции.

Пример системы взаимосвязанных диаграмм показан на рис. 5.2.

Диаграмма высшего (нулевого) уровня представляет систему как единый овал со стрелкой, ее называют основной или контекстной моделью. Контекстная модель используется для указания внешних связей программного изделия.

Для детализации (уточнения системы) вводится диаграмма 1-го уровня. Каждый из преобразователей этой диаграммы — подфункция общей системы. Таким образом, речь идет о замене преобразователя F на целую систему преобразователей.

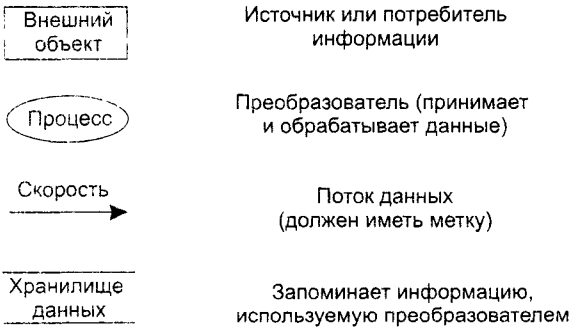


Рис. 5.1. Элементы диаграммы потоков данных

Дальнейшее уточнение (например, преобразователя *F3*) приводит к диаграмме 2-го уровня. Говорят, что ПДД1 разбивается на диаграммы 2-го уровня.

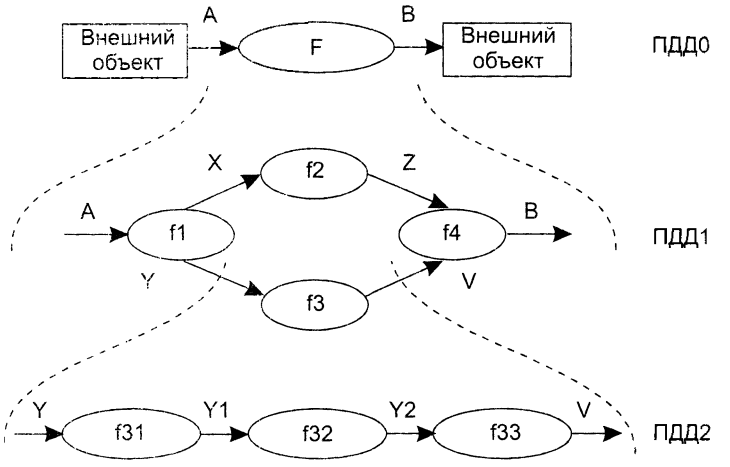


Рис. 5.2. Система взаимосвязанных диаграмм потоков данных

ПРИМЕЧАНИЕ

Важно сохранить непрерывность информационного потока и его согласованность. Это значит, что входы и выходы у каждого преобразователя на любом уровне должны оставаться прежними. В диаграмме отсутствуют точные указания на последовательность обработки. Точные указания откладываются до этапа проектирования.

Диаграмма потоков данных — это абстракция, граф. Для связи графа с проблемной областью (превращения в граф-модель) надо задать интерпретацию ее компонентов — дуг и вершин.

Описание потоков данных и процессов

Базовые средства диаграммы не обеспечивают полного описания требований к программному изделию. Очевидно, что должны быть описаны стрелки — потоки данных,

и преобразователи — процессы. Для этих целей используются словарь требований (данных) и спецификации процессов.

Словарь требований (данных) содержит описания потоков данных и хранилищ данных. Словарь требований является неотъемлемым элементом любой CASE-утилиты автоматизации анализа. Структура словаря зависит от особенностей конкретной CASE-утилиты. Тем не менее можно выделить базисную информацию типового словаря требований.

Большинство *словарей* содержит следующую информацию.

1. *Имя* (основное имя элемента данных, хранилища или внешнего объекта).
2. *Прозвище* (Alias) — другие имена того же объекта.
3. *Где и как используется объект* — список процессов, которые используют данный элемент, с указанием способа использования (ввод в процесс, вывод из процесса, как внешний объект или как память).
4. *Описание содержания* — запись для представления содержания.
5. *Дополнительная информация* — дополнительные сведения о типах данных, допустимых значениях, ограничениях и т. д.

Спецификация процесса — это описание преобразователя. Спецификация поясняет: ввод данных в преобразователь, алгоритм обработки, характеристики производительности преобразователя, формируемые результаты.

Количество спецификаций равно количеству преобразователей диаграммы.

ПРИМЕЧАНИЕ

Исходными данными для создания диаграмм потоков данных и словарей требований являются словесные требования заказчика, которые обсуждались в предыдущей главе. Это положение распространяется на все методы анализа, рассматриваемые в данной главе.

Расширения для систем реального времени

Как известно, программное изделие (ПИ) является дискретной моделью проблемной области, взаимодействующей с непрерывными процессами физического мира (рис. 5.3).

Рис. 5.3. Программное изделие как дискретная модель проблемной области

П. Вард и С. Меллор приспособили диаграммы потоков данных к следующим требованиям систем реального времени [99]:

1) информационный поток накапливается или формируется в непрерывном времени;

2) фиксируется управляющая информация. Считается, что она проходит через систему и связывается с управляющей обработкой;
3) допускается множественный запрос на одну и ту же обработку (из внешней среды).

Новые элементы имеют обозначения, показанные на рис. 5.4.

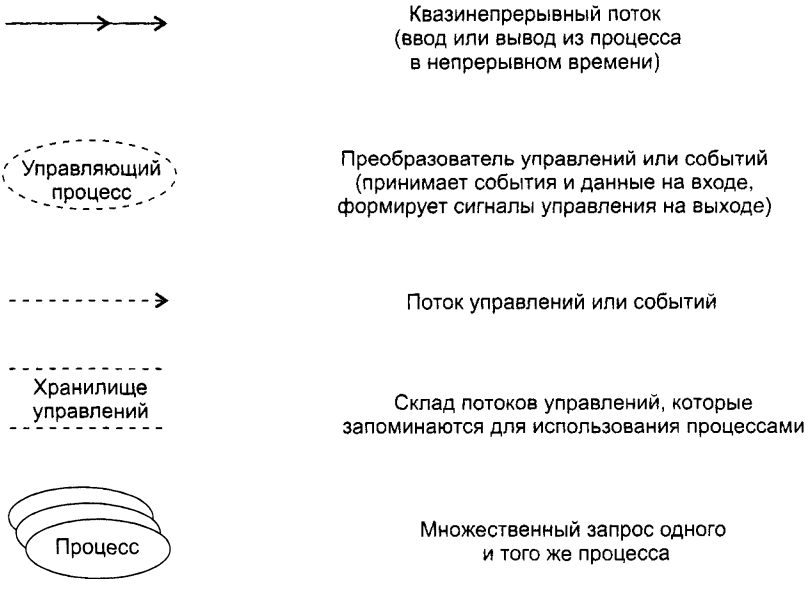


Рис. 5.4. Расширения диаграмм для систем реального времени

Приведем два примера использования новых элементов.

Пример 5.1. Использование потоков, непрерывных во времени.

На рис. 5.5 представлена модель анализа программного изделия для системы слежения за газовой турбиной.

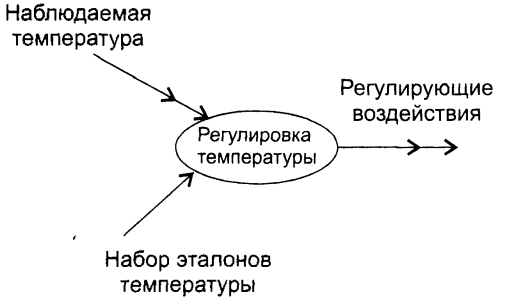


Рис. 5.5. Модель ПО для системы слежения за газовой турбиной

Видим, что здесь наблюдаемая температура измеряется непрерывно до тех пор, пока не будет найдено дискретное значение в наборе эталонов температуры.

Преобразователь формирует регулирующие воздействия как непрерывный во времени вывод. Чем полезна эта модель?

Во-первых, инженер делает вывод, что для приема-передачи квазинепрерывных значений нужно использовать аналого-цифровую и цифро-аналоговую аппаратуру.

Во-вторых, необходимость организации высокоскоростного управления этой аппаратурой делает критичным требование к производительности системы.

Пример 5.2. Использование потоков управления.

Рассмотрим компьютерную систему, которая управляет роботом (рис. 5.6).

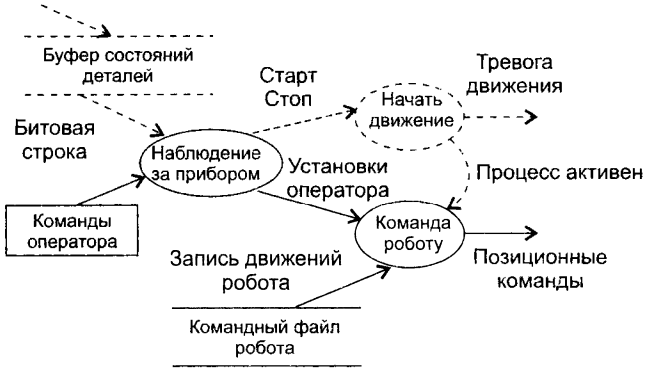


Рис. 5.6. Модель ПО для управления роботом

Установка в прибор деталей, собранных роботом, фиксируется установкой бита в буфере состояния деталей (он показывает присутствие или отсутствие каждой детали). Информация о событиях, запоминаемых в буфере, посылается в виде строки битов в преобразователь «Наблюдение за прибором». Преобразователь читает команды оператора только тогда, когда управляющая информация (битовая строка) показывает наличие всех деталей. Флаг события (Старт-Стоп) посылается в управляющий преобразователь «Начать движение», который руководит дальнейшей командной обработкой. Потоки данных посылаются в преобразователь команд роботу при наличии события «Процесс активен».

Расширение возможностей управления

Д. Хетли и И. Пирбхаи сосредоточили внимание на аспектах управления программным продуктом [54]. Они выделили системные состояния и механизм перехода из одного состояния в другое. Д. Хетли и И. Пирбхаи предложили не вносить в ПДД элементы управления, такие как потоки управления и управляющие процессы. Вместо этого они ввели диаграммы управляющих потоков (УПД).

Диаграмма управляющих потоков содержит:

- обычные преобразователи (управляющие преобразователи исключены вообще);
- потоки управления и потоки событий (без потоков данных).

Вместо управляющих преобразователей в УПД используются указатели — ссылки на управляющую спецификацию УСПЕЦ. Как показано на рис. 5.7, ссылка изображается как косая пунктирная стрелка, указывающая на окно УСПЕЦ (вертикальную черту).

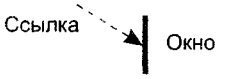


Рис. 5.7. Изображение ссылки на управляющую спецификацию

УСПЕЦ управляет преобразователями в ПДД на основе события, которое проходит в ее окно (по ссылке). Она предписывает включение конкретных преобразователей как результат конкретного события.

Иллюстрация модели программной системы, использующей описанные средства, приведена на рис. 5.8.

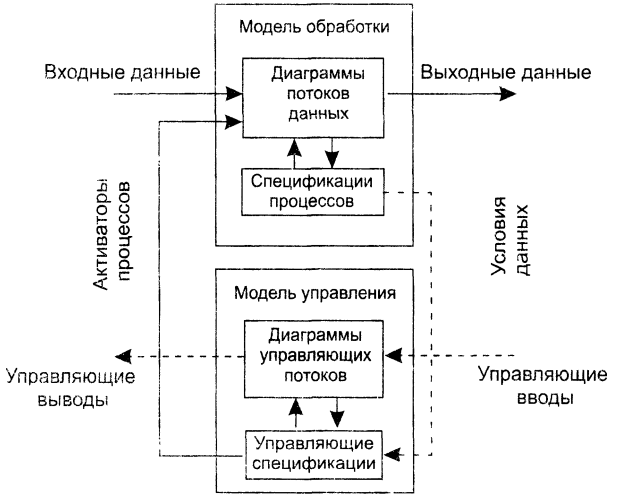


Рис. 5.8. Композиция модели обработки и управления

В модель обработки входит набор диаграмм потоков данных и набор спецификаций процессов. Модель управления образуют набор диаграмм управляющих потоков и набор управляющих спецификаций. Модель обработки подключается к модели управления с помощью активаторов процессов. Активаторы включают в конкретной ПДД конкретные преобразователи. Обратная связь модели обработки с моделью управления осуществляется с помощью условий данных. Условия данных формируются в ПДД (когда входные данные преобразуются в события).

Модель системы регулирования давления космического корабля

Обсудим модель системы регулирования давления космического корабля, представленную на рис. 5.9.

Начнем с диаграммы потоков данных. Основной процесс в ПДД — Слежение и регулирование давления. На его входы поступают: измеренное Давление в кабине и Max давление. На выходе процесса — поток данных Изменение давления. Содержание процесса описывается в его спецификации ПСПЕЦ.

Спецификация процесса ПСПЕЦ может включать:

1) поясняющий текст (обязательно);
2) описание алгоритма обработки;
3) математические уравнения;
4) таблицы;
5) диаграммы.

Элементы со второго по пятый не обязательны.

Рис. 5.9. Модель системы регулирования давления космического корабля

С помощью ПСПЕЦ разработчик создает описание для каждого преобразователя, которое рассматривается как:

- первый шаг создания спецификации требований к программному изделию;
- руководство для проектирования программ, которые будут реализовывать процессы.

В нашем примере спецификация процесса имеет вид:

```
если Давление в кабине > max
        то Избыточное давление:=1;
        иначе Избыточное давление:=0;
алгоритм регулирования;
выч.Изменение давления;
конец если;
```

Таким образом, когда давление в кабине превышает максимум, генерируется управляющее событие Избыточное давление. Оно должно быть показано на диаграмме управляющих потоков УПД. Это событие входит в окно управляющей спецификации УСПЕЦ.

Управляющая спецификация моделирует поведение системы. Она содержит:

- таблицу активации процессов (ТАП);
- диаграмму переходов-состояний (ДПС).

Таблица активации процессов показывает, какие процессы будут вызываться (активироваться) в потоковой модели в результате конкретных событий.

ТАП включает три раздела — Входные события, Выходные события, Активация процессов. Логика работы ТАП такова: входное событие вызывает выходное событие, которое активирует конкретный процесс. Для нашей модели ТАП имеет вид, представленный в табл. 5.1.

Таблица 5.1. Таблица активации процессов

Входные события:			
Включение системы	1	0	0
Избыточное давление	0	1	0
Норма	0	0	1
Выходные события:			
Тревога	0	1	0
Работа	1	0	1
Активация процессов:			
Слежение и регулирование давления	1	0	1
Уменьшение давления	0	1	0

Видим, что в нашем примере входных событий три: два внешних события (Включение системы, Норма) и одно — условие данных (Избыточное давление). Работа ТАП инициируется входным событием, «втекающим» в окно УСПЕЦ. В результате ТАП вырабатывает выходное событие — активатор. В нашем примере активаторами являются события Работа и Тревога. Активатор «вытекает» из окна УСПЕЦ, запуская в УПД конкретный процесс.

Другой элемент УСПЕЦ — Диаграмма переходов-состояний. ДПС отражает состояния системы и показывает, как она переходит из одного состояния в другое.

ДПС для нашей модели показан на рис. 5.10.

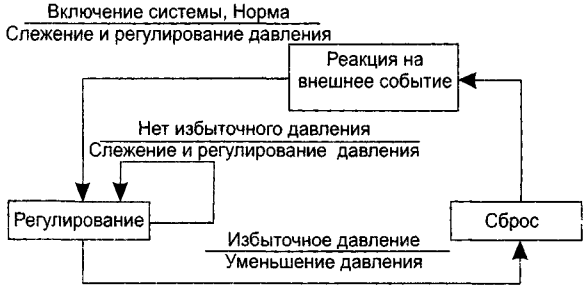


Рис. 5.10. Диаграмма переходов-состояний

Системные состояния показаны прямоугольниками. Стрелки показывают переходы между состояниями. Стрелки переходов подписывают следующим образом: в числителе – событие, которое вызывает переход, в знаменателе — процесс, запускаемый как результат события. Изучая ДПС, разработчик может анализировать поведение модели и установить, нет ли «дыр» в определении поведения.

Методы анализа, ориентированные на структуры данных

Элементами предметной области для любой системы являются потоки, процессы и структуры данных. При структурном анализе активно работают только с потоками данных и процессами.

Методы, ориентированные на структуры данных, обеспечивают:

1. Определение ключевых информационных объектов и операций.
2. Определение иерархической структуры данных.
3. Компоновку структур данных из типовых конструкций — последовательности, выбора, повторения.
4. Последовательность шагов для превращения иерархической структуры данных в структуру программы.

Наиболее известны два метода: метод Варнье–Орра и метод Джексона.

В методе Варнье–Орра для представления структур применяют диаграммы Варнье [78]. Для построения диаграмм Варнье используют три базовых элемента: последовательность, выбор, повторение (рис. 5.11) [100].

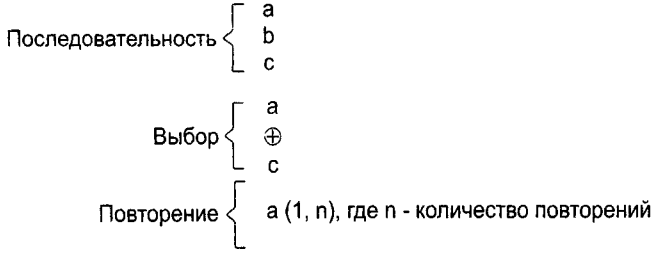


Рис. 5.11. Базовые элементы в диаграммах Варнье

Как показано на рис. 5.12, с помощью этих элементов можно строить информационные структуры с любым количеством уровней иерархии.

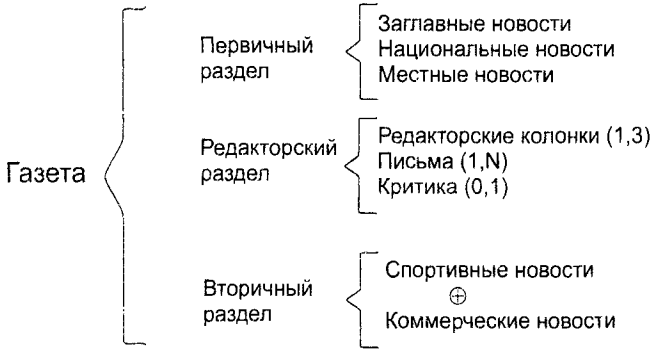


Рис. 5.12. Структура газеты в виде диаграммы Варнье

Как видим, для представления структуры газеты здесь используются три уровня иерархии.

Метод анализа Джексона

Как и метод Варнье–Орра, метод Джексона появился в период революции структурного программирования. Фактически оба метода решали одинаковую задачу: распространить базовые структуры программирования (последовательность, выбор, повторение) на всю область разработки сложных программных систем. Именно поэтому основные выразительные средства этих методов оказались так похожи друг на друга.

Методика Джексона

Метод Джексона (1975) включает 6 шагов [61]. Три шага выполняются на этапе анализа, а остальные — на этапе проектирования.

1. *Объект-действие.* Определяются объекты — источники или приемники информации и действия — события реального мира, воздействующие на объекты.
2. *Объект-структура.* Действия над объектами представляются диаграммами Джексона.
3. *Начальное моделирование.* Объекты и действия представляются как обрабатывающая модель. Определяются связи между моделью и реальным миром.
4. *Доопределение функций.* Выделяются и описываются сервисные функции.
5. *Учет системного времени.* Определяются и оцениваются характеристики планирования будущих процессов.
6. *Реализация.* Согласование с системной средой, разработка аппаратной платформы.

Шаг объект-действие

Начинается с определения проблемы на естественном языке.

Пример 5.3. Разработать компьютерную систему для обслуживания университетских перевозок. Университет размещается на двух территориях. Для перемещения студентов используется один транспорт. Он перемещается между двумя фиксированными остановками. На каждой остановке имеется кнопка вызова.

При нажатии кнопки:

- если транспорт на остановке, то студенты заходят в него и перемещаются на другую остановку;
- если транспорт в пути, то студенты ждут прибытия на другую остановку, приема студентов и возврата на текущую остановку;
- если транспорт на другой остановке, то он ее покидает, прибывает на текущую остановку и принимает студентов, нажавших кнопку.

Транспорт должен стоять на остановке до появления запроса на обслуживание.

Описание исследуется для выделения объектов. Производится грамматический разбор. Возможны следующие кандидаты в объекты: территория, студенты, транспорт, остановка, кнопка. У нас нет нужды прямо использовать территорию, студентов, остановку — все они лежат вне области модели и отвергаются как возможные объекты. Таким образом, мы выбираем объекты транспорт и кнопка.

Для выделения действий исследуются все глаголы описания.

Кандидатами действий являются: перемещаться, прибывает, нажимать, принимать, покидать. Мы отвергаем перемещаться, принимать потому, что они относятся к студентам, а студенты не выделены как объект. Мы выбираем действия: прибывает, нажимать, покидать.

Заметим, что при выделении объектов и действий возможны ошибки. Например, отвергнув студентов, мы лишились возможности исследования загрузки транспорта. Впрочем, список объектов и действий может модифицироваться в ходе дальнейшего анализа.

Шаг объект-структура

Структура объектов описывает последовательность действий над объектами (в условном времени).

Для представления структуры объектов Джексон предложил три типа структурных диаграмм. Они показаны на рис. 5.13. В первой диаграмме к объектам применяется такое действие, как последовательность, во второй — выбор, в третьей — повторение.

Рассмотрим объектную структуру для транспорта (рис. 5.14). Условимся, что начало и конец истории транспорта — у первой остановки. Действиями, влияющими на объект, являются Покинуть и Прибыть.

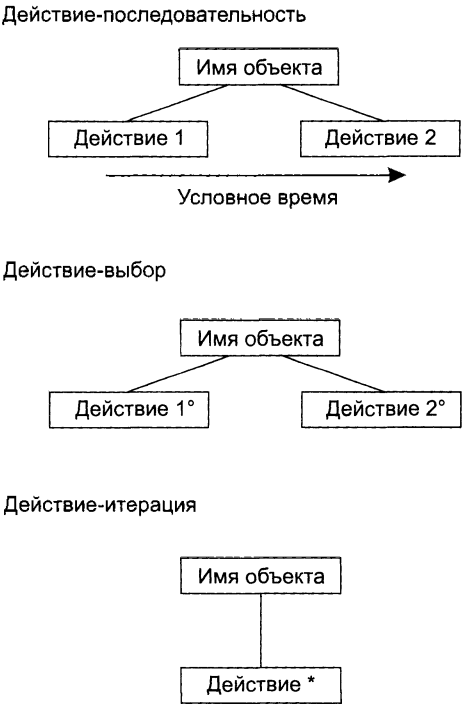


Рис. 5.13. Три типа структурных диаграмм Джексона

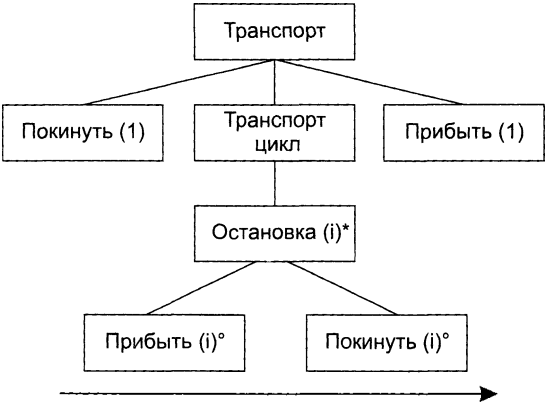


Рис. 5.14. Объектная структура для транспорта

Диаграмма показывает, что транспорт начинает работу у остановки 1, тратит основное время на перемещение между остановками 1 и 2 и окончательно возвращается на остановку 1. Прибытие на остановку, следующее за отъездом с другой остановки, представляется как пара действий Прибыть(i) и Покинуть(i). Заметим, что диаграмму можно сопровождать комментариями, которые не могут прямо пред-

ставляться средствами метода. Например, «значение *i* в двух последовательных остановках должно быть разным».

Структурная диаграмма для объекта Кнопка показывает (рис. 5.15), что к нему многократно применяется действие Нажать.

Рис. 5.15. Структурная диаграмма для объекта Кнопка

В заключение заметим, что структурная диаграмма — время-ориентированное описание действий, выполняемых над объектом. Она создается для каждого объекта модели.

Шаг начального моделирования

Начальное моделирование — это шаг к созданию описания системы как модели реального мира. Описание создается с помощью диаграммы системной спецификации.

Элементами диаграммы системной спецификации являются физические процессы (имеют суффикс 0) и их модели (имеют суффикс 1). Как показано на рис. 5.16, предусматриваются два вида соединений между физическими процессами и моделями.

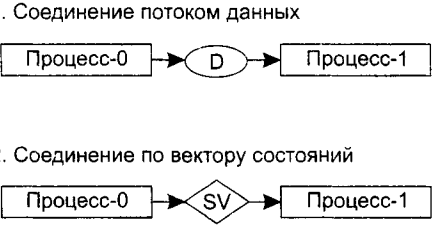


Рис. 5.16. Соединения между физическими процессами и их моделями

Соединение потоком данных производится, когда физический процесс передает, а модель принимает информационный поток. Полагают, что поток передается через буфер неограниченной емкости типа FIFO (обозначается овалом).

Соединение по вектору состояний происходит, когда модель наблюдает вектор состояния физического процесса. Вектор состояния обозначается ромбиком.

Диаграмма системной спецификации для системы обслуживания перевозок приведена на рис. 5.17.

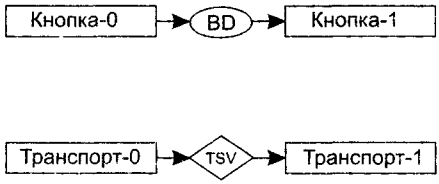


Рис. 5.17. Диаграмма системной спецификации для системы обслуживания перевозок

ПРИМЕЧАНИЕ

При нажатии кнопки формируется импульс, который может быть передан в модель как элемент данных, поэтому для кнопки выбрано соединение потоком данных.

Датчики, регистрирующие прибытие и убытие транспорта, не формируют импульса, они воздействуют на электронный переключатель. Состояние переключателя может быть оценено. Поэтому для транспорта выбрано соединение по вектору состояний.

Для фиксации особенностей процессов-моделей Джексон предлагает специальное описание — структурный текст. Например, структурный текст для модели КНОПКА-1 имеет вид:

```
КНОПКА-1
    читать BD;
    НАЖАТЬ цикл ПОКА BD
        нажать;
        читать BD;
    конец НАЖАТЬ;
конец КНОПКА-1;
```

Структура модели КНОПКА-1 отличается от структуры физического процесса КНОПКА-0 добавлением оператора для чтения буфера BD, который соединяет физический мир с моделью.

Прежде чем написать структурный текст для модели ТРАНСПОРТ-1, мы должны сделать ряд замечаний.

Во-первых, состояние транспорта будем отслеживать по переменным ПРИБЫЛ, УБЫЛ. Они отражают состояние электронного переключателя физического транспорта.

Во-вторых, для учета инерционности процессов в физическом транспорте в модель придется ввести дополнительные операции:

- ЖДАТЬ (ожидание в изменении состояния физического транспорта);
- ТРАНЗИТ (операция задержки в модели на перемещение транспорта между остановками).

С учетом замечаний структурная диаграмма модели примет вид, изображенный на рис. 5.18.

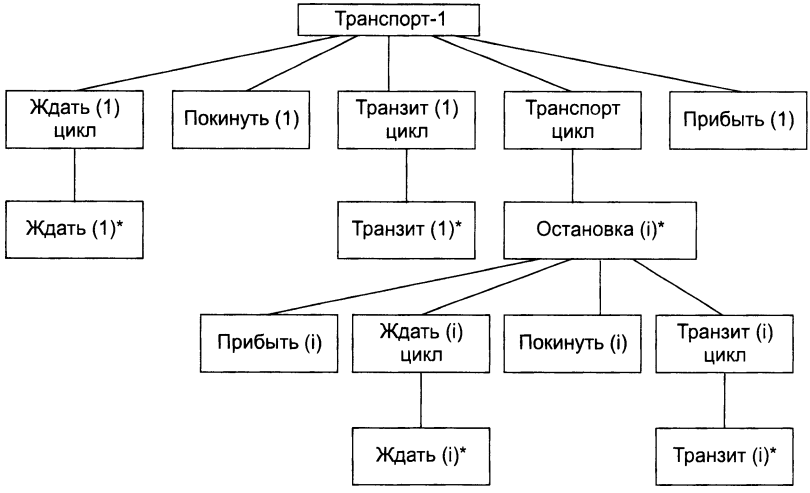


Рис. 5.18. Структурная диаграмма модели транспорта

Соответственно, структурный текст модели записывается в форме:

```
ТРАНСПОРТ-1
    опрос TSV;
    ЖДАТЬ цикл ПОКА ПРИБЫЛ(1)
        опрос TSV;
    конец ЖДАТЬ;
    покинуть(1);
    ТРАНЗИТ цикл ПОКА УБЫЛ(1)
        опрос TSV;
    конец ТРАНЗИТ;
    ТРАНСПОРТ цикл
        ОСТАНОВКА
            прибыть(i);
            ЖДАТЬ цикл ПОКА ПРИБЫЛ(i)
                опрос TSV;
            конец ЖДАТЬ;
            покинуть(i);
            ТРАНЗИТ цикл ПОКА УБЫЛ(i)
                опрос TSV;
            конец ТРАНЗИТ;
        конец ОСТАНОВКА;
    конец ТРАНСПОРТ;
    прибыть(1);
конец ТРАНСПОРТ-1;
```

Он описывает полный цикл работы транспорта.

Контрольные вопросы и упражнения

1. Какие задачи решает аппарат анализа?
2. Что такое диаграмма потоков данных?
3. Чем отличается диаграмма потоков данных от блок-схемы алгоритма?
4. Какие элементы диаграммы потоков данных вы знаете?

5. Как формируется иерархия диаграмм потоков данных?
6. Какую задачу решает диаграмма потоков данных высшего (нулевого) уровня? Почему ее называют контекстной моделью?
7. Чем нагружены вершины диаграммы потоков данных?
8. Чем нагружены дуги диаграммы потоков данных?
9. Как организован словарь требований?
10. С помощью аппарата структурного анализа создайте модель системы управления летательного аппарата. В качестве исходных данных используйте требования заказчика из примера 4.1 предыдущей главы.
11. С чем связана необходимость расширения диаграмм потоков данных для систем реального времени? Какие средства расширения вы знаете?
12. С помощью расширений Варда и Меллора модифицируйте модель, построенную в пункте 10.
13. Как решается проблема расширения возможностей управления на базе диаграмм потоков данных?
14. Каковы особенности диаграммы управляющих потоков?
15. Поясните понятие активатора процесса.
16. Поясните понятие условия данных.
17. Поясните понятие управляющей спецификации.
18. Поясните понятие окна управляющей спецификации.
19. Как организована спецификация процесса?
20. Поясните назначение таблицы активации процессов.
21. Поясните организацию диаграммы переходов-состояний.
22. Примените расширения Хетли и Пирбхаи к усовершенствованию модели из пункта 12. Какие новые проблемы приходится теперь решать?
23. Какие задачи решают методы анализа, ориентированные на структуры данных?
24. Какие методы анализа, ориентированные на структуры данных, вы знаете?
25. Из каких базовых элементов состоят диаграммы Варнье?
26. Создайте с помощью диаграмм Варнье модель газеты вашего института.
27. Какие шаги выполняет метод Джексона на этапе анализа?
28. Какие типы структурных диаграмм Джексона вы знаете?
29. Как организовано в методе Джексона обнаружение объектов?
30. Что такое структура объектов Джексона?
31. Как создается структура объектов Джексона?
32. Поясните диаграмму системной спецификации Джексона.
33. Чем отличается соединение потоком данных от соединения по вектору состояний?
34. Какова задача структурного текста Джексона?
35. Усовершенствуйте модель из примера 5.3 данной главы для случая, когда университет размещается на трех территориях.

Глава 6

Основы проектирования программных систем

В этой главе обсуждается содержание этапа проектирования и его место в жизненном цикле разработки программных систем. Подробно рассматривается архитектурное проектирование, особенности применения архитектурных паттернов. Дается обзор принципов, средств и характеристик проектирования: разделения понятий, модульности, информационной закрытости, пошаговой детализации, аспектов, рефакторинга, сложности, связности, сцепления и метрик для их оценки.

Особенности процесса синтеза программных систем

Известно, что технологический цикл разработки любого инженерного изделия включает анализ и синтез. Аналогичные действия можно выделить и в разработке программной системы (ПС).

В ходе анализа ищется ответ на вопрос: «Что должна делать будущая система?» Именно на этой стадии закладывается фундамент успеха всего проекта. Известно множество неудачных реализаций из-за неполноты и неточностей в определении требований к системе.

В процессе синтеза формируется ответ на вопрос: «Каким образом система будет реализовывать предъявляемые к ней требования?» Выделяют три этапа синтеза: проектирование ПС, кодирование ПС, тестирование ПС (рис. 6.1).

Рассмотрим информационные потоки процесса синтеза.

Этап проектирования питают требования к ПС, представленные информационной, функциональной и поведенческой моделями анализа. Иными словами, модели анализа поставляют этапу проектирования исходные сведения для работы. Информационная модель описывает информацию, которую, по мнению заказчика, должна обрабатывать ПС. Функциональная модель определяет перечень функций обработки. Поведенческая модель фиксирует желаемую динамику системы (режимы ее работы). На выходе этапа проектирования — модель данных, модель архитектуры и модели подсистем архитектуры ПС.

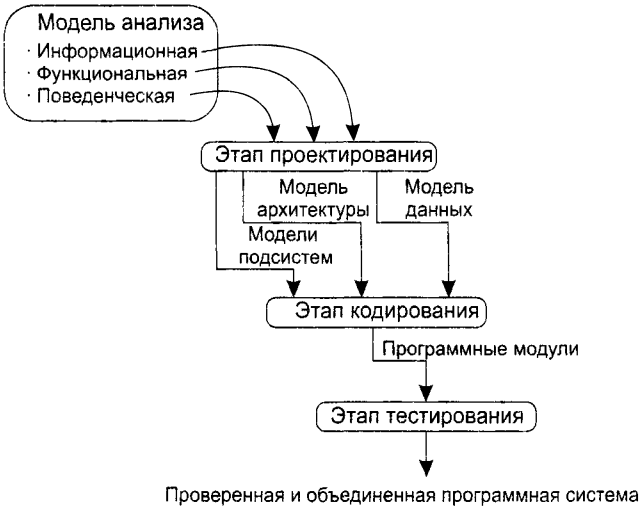


Рис. 6.1. Информационные потоки процесса синтеза ПС

Модель данных — это результат преобразования информационной модели анализа в структуры данных, которые потребуются для реализации программной системы.

Модель архитектуры выделяет основные подсистемы структуры, фиксирует связи между ними и задает принципы взаимодействия подсистем.

Модели подсистем описывают организацию подсистем, то есть определяют их содержание.

Далее создаются тексты программных модулей, проводится тестирование для объединения и проверки ПС. На проектирование, кодирование и тестирование приходится более 75% стоимости разработки ПС. Принятые здесь решения оказывают решающее воздействие на успех реализации ПС и легкость, с которой ПС будет сопровождаться.

Следует отметить, что решения, принимаемые в ходе проектирования, делают его стержневым этапом процесса синтеза. Важность проектирования можно определить одним словом — качество. Проектирование — этап, на котором «выращивается» качество разработки ПС.

ПРИМЕЧАНИЕ

Международный стандарт ISO 8402:1994 определяет качество так: «это совокупность характеристик объекта, относящихся к его способности удовлетворить установленные и предполагаемые потребности». В программной инженерии под качеством понимают степень соответствия системы, компонента или процесса определенным требованиям.

Справедлива следующая аксиома разработки: может быть плохая ПС при хорошем проектировании, но не может быть хорошей ПС при плохом проектировании. Проектирование обеспечивает нас такими представлениями ПС, качество которых

можно оценить. Проектирование — единственный путь, обеспечивающий правильную трансляцию требований заказчика в конечный программный продукт.

Особенности архитектурного этапа проектирования

Проектирование — итерационный процесс, при помощи которого требования к ПС транслируются в инженерные представления ПС. Вначале эти представления дают только концептуальную информацию (на высоком уровне абстракции), последующие уточнения приводят к формам, которые близки к текстам на языках программирования.

Обычно в проектировании выделяют две ступени: архитектурное проектирование и детальное проектирование. Архитектурное проектирование формирует абстракции высокого уровня, детальное проектирование уточняет эти абстракции, добавляет подробности алгоритмического уровня. Кроме того, во многих случаях выделяют интерфейсное проектирование, цель которого — сформировать графический интерфейс пользователя (GUI). Схема информационных связей процесса проектирования приведена на рис. 6.2.

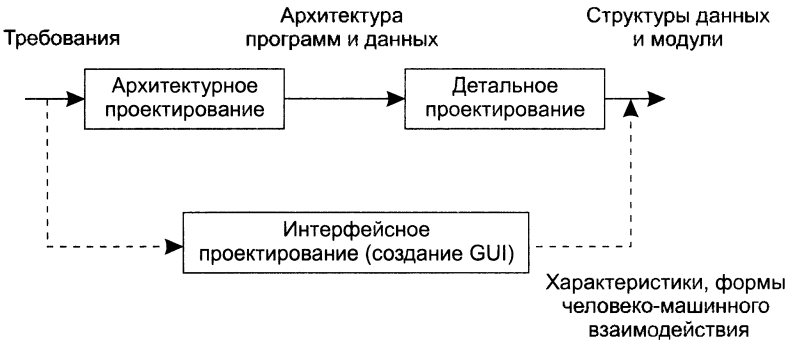


Рис. 6.2. Информационные связи процесса проектирования

Архитектурное проектирование обеспечивает понимание правильной организации системы и создает структуру под эту правильную организацию. Оно напрямик связывает весь этап проектирования с детальными требованиями, поскольку архитектура выделяет основные структурные компоненты системы и формирует отношения между ними.

Между созданием детальных требований и архитектурным проектированием имеется существенное перекрытие. В идеале в требованиях не должно быть информации о структуре системы. На самом же деле это справедливо только для малых систем. Архитектурное разделение системы на части просто необходимо для структуризации и организации спецификации требований. Поэтому перед созданием детальных требований формируется абстрактная архитектура системы, с подсистемами которой ассоциируются целые группы функций и характеристик. Эта же архитектура используется для обсуждения требований и характеристик системы с заинтересованными лицами.

Многие авторы отмечают разностороннее и решающее воздействие архитектуры на качество программной системы [40, 88, 94]:

- Архитектура является планом для переговоров по требованиям к системе и служит средством упорядочения обсуждений с клиентами, разработчиками и менеджерами.
- Архитектура — основной инструмент для управления сложностью и качеством системы. Подсистемы, составные части архитектуры, отвечают за реализацию функциональных требований и сильно влияют на нефункциональные требования. Именно архитектура определяет такие характеристики системы в целом, как производительность, защищенность, безопасность, устойчивость к отказам и сопровождаемость.
- Архитектура — высокоуровневый инструмент повторного использования программного обеспечения. Она обеспечивает целые семейства систем со схожей функциональностью, упрощая разработку конкретного экземпляра, с конкретными функциями и характеристиками.

Различным требованиям к интегральным характеристикам системы должны соответствовать разные варианты архитектуры. Обсудим эти варианты.

Если главным требованием является производительность системы, следует спроектировать архитектуру, которая локализует критические операции в пределах небольшого количества подсистем с минимальным числом взаимодействий между ними. Эти подсистемы следует развернуть на одном и том же компьютере, а не распределять по сети. Для сокращения внешних коммуникаций подсистемы должны быть крупными, с большой степенью автономности.

Для обеспечения максимальной защищенности архитектура должна быть многоуровневой, причем самые ценные подсистемы следует размещать на внутренних уровнях, с высокой степенью защиты.

При ориентации на максимальную безопасность поддержку безопасности нужно сосредоточить в одной подсистеме (или малом числе подсистем). Это уменьшит стоимость проектирования и позволит применить эффективный механизм проверки надежности. Такой механизм обеспечит максимальную сохранность данных при отказе в системе.

Для создания устойчивости к отказам (обеспечения бесперебойности работы) в архитектуру вводятся избыточные (резервные) подсистемы. Их наличие позволяет выполнять замену отказавших элементов без остановки системы.

Наконец, удобство сопровождения повышается при проектировании архитектуры из малых подсистем, которые легче проверять и обновлять. Для облегчения диагностики ошибок желателен отказ от глобальных структур данных. Заметим, что при этом возникает конфликт с архитектурой, нацеленной на максимальную производительность. Там нужны большие подсистемы, здесь — малые. Приходится идти на какой-то компромисс.

Поиск компромиссного решения — типичная задача архитектурного проектирования, поскольку, как правило, требуется максимизация многих параметров системы.

Архитектура системы часто моделируется с помощью простых блочных диаграмм. Каждый блок в диаграмме представляет подсистему (компонент). Блоки

внутри блоков означают, что компонент разбивается на субкомпоненты. Стрелки показывают передачу данных и управляющих сигналов между компонентами. Конечно, блочные диаграммы — скудное средство. За кадром остаются такие важные вопросы, как типы взаимодействий между компонентами, многочисленные характеристики компонентов-подсистем. Фактически блочная диаграмма отражает только структуру системы. Зато она проста и наглядна. Многие считают, что это «святая простота»:

- Она полезна для обсуждения системы с заинтересованными лицами и для планирования проекта, потому что не обременена деталями.
- Не специалист, но заинтересованное лицо сможет понять главные идеи организации системы, не вдаваясь в детали.
- Менеджер видит ключевые компоненты, которые должны быть разработаны, и начинает понимать, каких сотрудников надо привлечь к разработке.

Скудность блочной диаграммы можно компенсировать документированием архитектуры. За счет документирования формируется подробное архитектурное описание, которое поясняет различные компоненты системы, их интерфейсы, соединения и облегчает понимание и развитие системы. Правда, на практике иногда полезность архитектурных документов недооценивают, то есть игнорируют эту работу. Отметим, что в гибких процессах разработки ранняя стадия работ нацелена на создание минимальной версии архитектуры системы, которая в дальнейших итерациях изменяется и развивается до полномасштабного варианта. Как правило, пошаговое изменение архитектуры проблематично и очень часто заканчивается провалом.

Во время архитектурного проектирования системные архитекторы должны принимать такие решения, которые глубоко затрагивают систему и процесс ее разработки. Основываясь на своих знаниях и опыте, они решают широкий спектр задач. Все эти задачи группируются вокруг трех типов базисной деятельности.

Базисная деятельность архитектурного проектирования включает:

1. *Структурирование системы*[1]. Система структурируется на несколько подсистем, где под подсистемой понимается независимый программный компонент. Определяются взаимодействия подсистем.
2. *Моделирование управления.* Формируется стратегия управления частями системы.
3. *Декомпозиция подсистем на модули.* Каждая подсистема разбивается на модули. Определяются типы модулей и межмодульные соединения.

Рассмотрим вопросы структурирования, моделирования и декомпозиции более подробно.

Структурирование системы

В ходе архитектурного проектирования создается структурная организация системы, которая будет отвечать всем функциональным и нефункциональным требованиям. Успех этого творческого процесса зависит от типа разрабатываемой

[1] Под структурированием понимают комплекс задач, связанных с формированием структурной организации системы.

системы, подготовки и опыта системного архитектора, а также от конкретных требований к системе.

Каждая программная система уникальна, но системы в одной и той же прикладной области часто имеют сходные архитектуры, отражающие фундаментальные положения этой области. Например, семейство продуктов — это приложения, которые строятся вокруг основной архитектуры с вариантами, удовлетворяющими специфическим требованиям клиента. Проектируя системную архитектуру, нужно выяснить общие черты создаваемой системы, а также более широкой категории приложений и решить, что можно позаимствовать из этой прикладной архитектуры.

Во встроенных системах и системах для персональных компьютеров обычно имеется только один процессор, и не нужно проектировать распределенную архитектуру. Однако современные большие системы являются распределенными системами, в которых системное программное обеспечение распределено по многим компьютерам. Выбор распределенной архитектуры — серьезное решение, которое влияет на производительность и надежность системы.

Архитектура программной системы может быть основана на определенном архитектурном паттерне. *Архитектурный паттерн* — это описание типовой организации системы. Архитектурные паттерны фиксируют сущность архитектуры, которая использовалась в различных программных системах. Идея паттернов как способа многократного использования знаний о программных системах в настоящее время находит широко применение.

Архитектурный паттерн можно рассматривать как обобщенное описание хорошей практики, опробованной и проверенной в различных системах и средах. Таким образом, архитектурный паттерн должен описать системную организацию, которая была успешна в предыдущих системах.

Таблица 6.1. Паттерн Модель-представление-контроллер (Model-View-Controller) MVC

Имя	**MVC (Модель-представление-контроллер)**
Описание	Отделяет представление системы и взаимодействие с системой от данных системы. Система разделяется на три логических компонента, которые взаимодействуют друг с другом. Компонент Модель управляет системными данными и операциями над данными. Компонент Представление отображает данные для пользователя. Компонент Контроллер взаимодействует с пользователем, инициирует операции в модели и управляет работой представления. Структуру паттерна поясняет рис. 6. 3
Пример	Архитектура веб-системы на основе паттерна MVC показана на рис. 6.4
Когда используется	1. Когда требуется несколько вариантов обработки и представления данных. 2. Когда неизвестны требования к взаимодействию и представлению данных
Преимущества	Позволяет изменять данные независимо от их представления. Изменение данных, сделанное в одном представлении, отображается во всех остальных представлениях
Недостатки	Избыточность программного кода при простой модели данных и простых схемах взаимодействия

Например, в табл. 6.1 описан популярный паттерн модель-представление-контроллер (MVC). Этот паттерн определяет архитектуру многих систем, ориентированных на

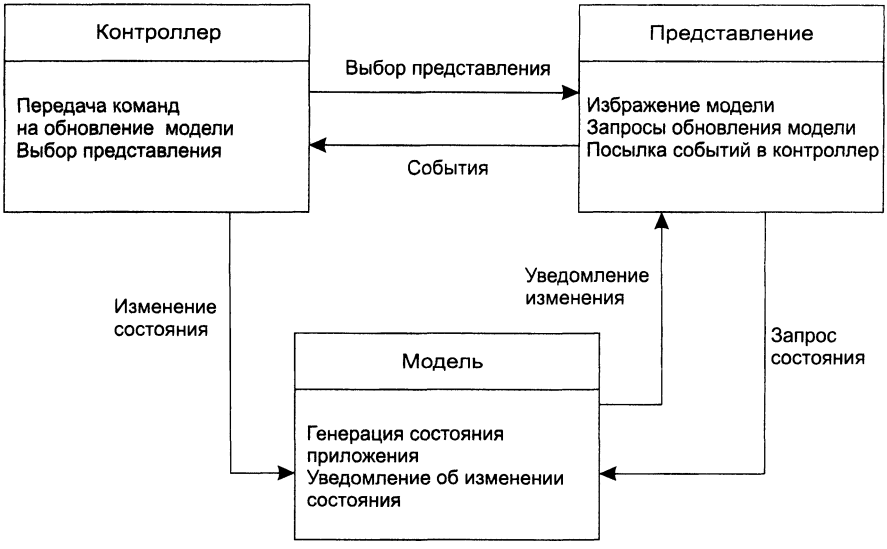


Рис. 6.3. Структура паттерна модель-представление-контроллер

Рис. 6.4. Архитектура веб-системы на основе паттерна модель-представление-контроллер

обслуживание клиентов. Поскольку паттерн является образцом типового решения, с которым знакомятся и которое могут применить многие архитекторы, его описание должно содержать развернутую характеристику. Обычно описание состоит из следующих разделов:

- *Имя паттерна* (должно характеризовать его суть).
- *Описание* (краткое и понятное описание функциональных возможностей и структуры паттерна, структура поясняется графической диаграммой).
- *Пример* (приводится пример типового применения паттерна, поясняемый диаграммой).
- *Когда используется* (описываются условия применения, дается обобщенная характеристика предметных областей).
- *Преимущества* (перечисляются преимущества применения).
- *Недостатки* (указываются слабые стороны данного паттерн-решения).

Структура паттерна модель-представление-контроллер приведена на рис. 6.3, а архитектура веб-системы, построенной на его основе, показана на рис. 6.4. Здесь паттерн реализует механизм управления взаимодействием с пользователем.

Рассмотрим дополнительные примеры паттерн-решений архитектурного структурирования.

Архитектура с хранилищем данных

Во многих системах подсистемы разделяют данные, находящиеся в общем хранилище. Как правило, данные образуют базу данных (БД). Предусматривается система управления этой базой. Архитектурный паттерн хранилища данных (табл. 6.2) описывает организацию взаимодействия подсистем через БД.

Таблица 6.2. Паттерн Хранилища данных

Имя	Хранилище данных
Описание	В центральном хранилище находятся все данные системы. Эти данные доступны всем подсистемам. Подсистемы взаимодействуют друг с другом косвенно, через хранилище
Пример	Архитектура Case-системы на основе хранилища показана на рис. 6.5. Здесь каждая подсистема генерирует данные, которые доступны другим подсистемам
Когда используется	Когда требуется создавать и долгое время хранить большие объемы информации. Удобно использовать в системах, порядок работы которых определяется данными
Преимущества	1. Предоставляет подсистемам простую схему для получения устойчиво существующих объектов и управления их жизненным циклом. 2. Убирает необходимость в технической поддержке целостности объектов, разных вариантов технологий СУБД и даже разных источников данных. 3. Скрывает сложность механизма доступа к устойчивым объектам 4. Обеспечивает функциональную независимость подсистем
Недостатки	Трудности размещения хранилища на нескольких компьютерах. Возможно понижение скорости доступа к данным. Проблемы хранилища прямо влияют на всю систему

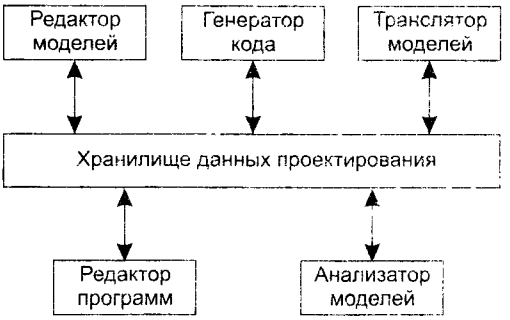


Рис. 6.5. Архитектура Case-системы на основе паттерна хранилище данных

Этот паттерн удобен для систем, где одни компоненты генерируют данные, а другие используют их (например, систем управления, информационных систем, среды для разработки программ и т. д.).

На рис. 6.5 демонстрируется архитектура системы автоматизации разработки ПО на базе паттерна хранилища. Обрамление хранилища набором утилит позволяет организовать эффективную обработку больших объемов данных, находящихся в совместном использовании. В этом случае нет нужды передавать данные непосредственно от утилиты к утилите. С другой стороны, форматы данных утилит должны соответствовать формату данных хранилища. Согласование форматов может понижать производительность утилит. Если же согласование невозможно, интегрировать новую утилиту в систему нельзя.

Еще одна проблема возникает при необходимости размещения хранилища на нескольких компьютерах, поскольку появляются трудности в обеспечении резервирования и целостности данных.

Чаще всего хранилище является ведомым элементом системы, который управляется другими подсистемами. Возможно и другое решение, получившее название «классной доски». Хранилище, играющее роль классной доски, само вызывает подсистемы по мере готовности данных.

Клиент-серверная архитектура

Популярным архитектурным решением для распределенных систем является паттерн клиент-сервер (табл. 6.3).

Таблица 6.3. Паттерн клиент-сервер

Имя	**Клиент-сервер**
Описание	Функциональность системы обеспечивается набором услуг (сервисов). Каждый сервис располагается на своем сервере. Клиенты являются пользователями этих сервисов. Для получения услуги клиент обращается к серверу
Пример	Архитектура сетевой библиотечной системы на основе паттерна клиент-сервер показана на рис. 6.6
Когда используется	1. Когда услуги должны быть доступны из разных мест. 2. Когда требуется гибкий механизм перестройки системы по загружаемым начальным данным

Имя	Клиент-сервер
Преимущества	1. Предоставление клиентам различных услуг через сеть. 2. Устраняется необходимость тиражирования реализации услуг среди серверов
Недостатки	Возможно понижение скорости доступа к данным из-за проблем в сети. Поломка сервера лишает клиента услуги

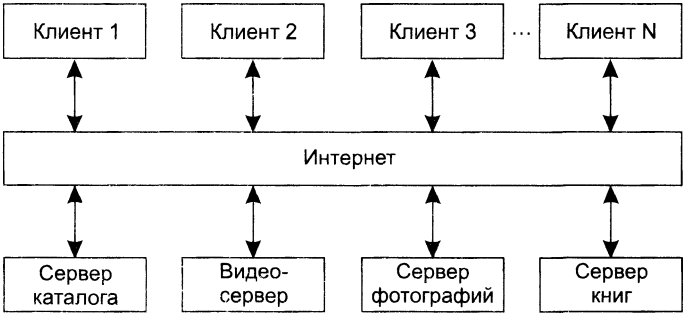


Рис. 6.6. Клиент-серверная архитектура сетевой библиотеки

Система здесь организована в виде набора сервисов, находящихся на серверах, и клиентов, которые обращаются к серверам и используют сервисы. Главные элементы этой архитектуры:

1. Набор серверов, предлагающих услуги клиентам (например, серверы услуг печати, серверы для записи и хранения файлов, серверы-переводчики).
2. Набор клиентов, которые обращаются к сервисам серверов. В роли клиентов выступают клиентские программы, часто запускаемые одновременно на различных компьютерах.
3. Сеть, которая позволяет клиентам обращаться к сервисам. Чаще всего клиент-серверные системы реализованы как распределенные системы, соединенные с помощью интернет-протокола.

В принципе логическую модель независимых сервисов, работающих на отдельных серверах, можно реализовать и на одном компьютере. Зачем? Такое решение несет разделение и независимость. Сервисы и серверы могут быть изменены, не затрагивая клиентов.

В этой архитектуре реализуется схема несимметричного именования: клиент должен знать имя сервера и сервиса, а серверу нет нужды знать имя клиента. Запрос клиента к серверу синхронен, он ждет получения ответа.

Различают «тонкого» и «толстого» клиента. В случае «тонкого» клиента вся обработка и управление данными выполняются на сервере. На клиентском компьютере запускаются только функции приема и отображения данных — реализуется пользовательский интерфейс. При «толстом» клиенте сервер только управляет данными. На клиентском компьютере организована обработка данных и взаимодействие с пользователем системы.

Главный недостаток модели «тонкого» клиента — большая загруженность сервера и сети (все вычисления выполняются на сервере). Напротив, модель «толстого» клиента максимально использует вычислительную мощность клиентских

компьютеров: на них перемещаются и операции обработки, и операции представления. Примером «толстых» клиентов являются банкоматы.

Обсудим клиент-серверную архитектуру сетевой библиотеки (рис. 6.6). Эта система предлагает клиентам электронные версии книг, фильмы и фотографии. В системе имеется несколько серверов, осуществляющих обработку, хранение и выдачу различных типов данных. Аудио- и видеоинформацию следует передавать синхронно, в реальном масштабе времени, что обеспечивает видеосервер. Фотографии позволяет просматривать с высоким разрешением отдельный сервер. Сервер каталога обслуживает запросы поиска, а сервер книг — запросы книжных изданий. В качестве клиентов рассматриваются экземпляры веб-браузера пользователей. Отметим, что благодаря распределенности архитектуры интеграция нового сервера в библиотеку осуществляется достаточно легко.

В обычной, двухъярусной системе клиент-сервер могут возникнуть существенные проблемы с размещением на аппаратуре трех логических ярусов — представления, обработки и хранения данных. Чтобы избежать этих проблем, применяют трехъярусную архитектуру клиент-сервер (рис. 6.7). В этой архитектуре ярусам представления, обработки и хранения данных соответствуют отдельные подсистемы.

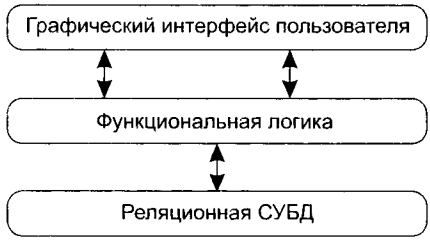


Рис. 6.7. Трехъярусная архитектура клиент-сервер

Ярус графического интерфейса пользователя запускается на машине клиента. Функциональную логику образуют модули, осуществляющие функциональные обязанности системы (снятие денег, перевод денег, изменение процентной ставки и т. д.). Этот ярус запускается на сервере приложения. Реляционная СУБД хранит данные, требуемые ярусу функциональной логики. Этот ярус запускается на втором сервере — сервере базы данных. Впрочем, на одном компьютере-сервере можно запустить и функциональную логику, и управление данными как отдельные логические серверы. Если же требования к системе возрастут, достаточно просто разделить эти логические серверы и выполнять их на разных машинах.

Преимущества трехъярусной модели:

- упрощается такая модификация яруса, которая не влияет на другие ярусы;
- отделение прикладных функций от функций управления БД упрощает оптимизацию всей системы.

Многоуровневая архитектура

Паттерн многоуровневой архитектуры (табл. 6.4) предлагает организовать функциональность в виде отдельных уровней. Каждый уровень реализуется с исполь-

зованием средств и сервисов, обеспечиваемых уровнем, который расположен непосредственно под ним.

Таблица 6.4. Паттерн многоуровневая архитектура

Имя	Многоуровневая архитектура
Описание	Система организуется в виде набора уровней. С каждым уровнем связывается определенная функциональность. Каждый уровень предлагает услуги вышестоящему уровню и использует услуги нижестоящего уровня. Структуру паттерна поясняет рис. 6. 8
Пример	Архитектура многоуровневой библиотеки диссертационного фонда показана на рис. 6.9
Когда используется	1. Когда разработка обеспечивается чередой команд, причем каждая команда создавала функциональность одного уровня. 2. Когда новые возможности создаются на базе существующих систем. 3. Когда требуется многоуровневая защищенность
Преимущества	1. Позволяет замещать целые уровни (при условии сохранения интерфейса). 2. Для повышения надежности в каждый уровень можно добавить дополнительные возможности (например, аутентификацию)
Недостатки	1. Трудно обеспечить ясное разделение уровней. Текущий уровень может взаимодействовать не с соседом снизу, а с более низкими уровнями. 2. Возможно понижение производительности, поскольку запрос услуги может последовательно обрабатываться на каждом уровне

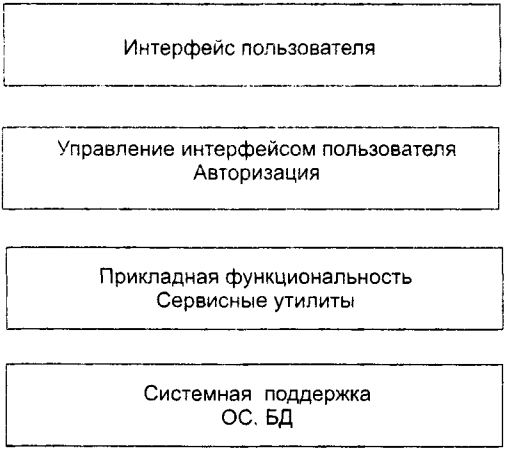


Рис. 6.8. Многоуровневая архитектура

К достоинствам архитектуры следует отнести изменяемость и переносимость. Изменения могут быть легко локализованы: при сохранении интерфейса можно модифицировать содержание целого уровня. Кроме того, изменение интерфейса уровня затрагивает лишь смежный уровень. Перенос системы в другую операционную среду требует замены только нижнего уровня (именно он зависит от компьютерной платформы).

Обсудим реализацию многоуровневой архитектуры (см. рис. 6.8). Самый нижний уровень содержит средства системной поддержки (взаимодействие с операционной

системой, база данных). Второй уровень включает средства прикладной функциональности и сервисные утилиты, обслуживающие разные подсистемы приложения. Третий уровень управляет пользовательским интерфейсом и обеспечивает авторизацию пользователя, а верхний уровень предлагает средства пользовательского интерфейса.

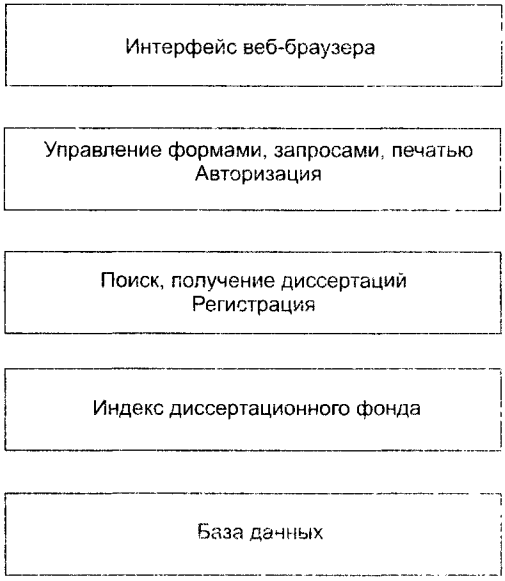


Рис. 6.9. Архитектура библиотеки диссертационного фонда

Еще одна иллюстрация варианта многоуровневой архитектуры показана на рис. 6.9. Здесь отображены средства ограничения доступа и сервисные средства библиотеки диссертационного фонда, которые размещены на пяти уровнях системы.

Архитектура канала и фильтра

Паттерн канала или фильтра (табл. 6.5) предлагает рассматривать архитектурную структуру как воплощение диаграммы потоков данных (обсуждалась в пятой главе): обработка выполняется функциональными преобразователями (именуются фильтрами), данные между ними переносятся по каналам. Потоки данных перемещаются от одного фильтра к другому и преобразуются по мере движения по последовательности фильтров. Каждый шаг обработки — это преобразование, реализуемое фильтром. Фильтр выделяет из потока только те данные, которые может обработать. Сформированный результат фильтр возвращает в поток. Потоки входных данных перемещаются через эти фильтры до тех пор, пока не будут преобразованы в выходные данные.

Таблица 6.5. Паттерн канала и фильтра

Имя	Канал и фильтр
Описание	Обработка данных в системе организуется с помощью обрабатывающих компонентов (фильтров) — преобразователей. Каждый фильтр выполняет один тип преобразования данных. Для обработки между компонентами устанавливаются потоки данных (каналы)
Пример	Архитектура системы угловой стабилизации ЛА на основе паттерна канала и фильтра показана на рис. 6.10
Когда используется	Когда создается система обработки, управляемая потоком данных. Процесс обработки образуется последовательностью этапов, в которой результаты предыдущего этапа являются входными данными последующего этапа
Преимущества	1. Простота системы. 2. Поддержка механизма повторного использования. 3. Прозрачное наращивание количества преобразований
Недостатки	Должен быть согласован формат данных, перемещаемых между взаимодействующими преобразователями. Каждый преобразователь должен «разбирать» входные данные и «собирать» результаты в соответствии с согласованным форматом. Это увеличивает накладные расходы. Нельзя применять преобразователи с несовместимыми структурами данных

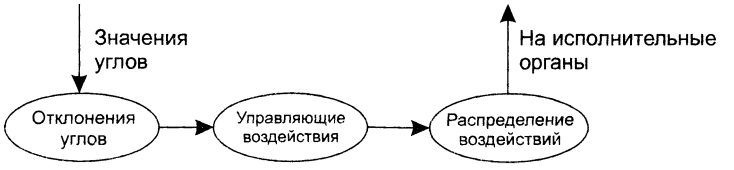


Рис. 6.10. Архитектура системы угловой стабилизации ЛА

Пример этого типа системной архитектуры, используемой в приложении с пакетной обработкой, показан на рис. 6.10. Система управляет угловым положением летательного аппарата (ЛА). С измерительных устройств поступает информация о текущих значениях углов по трем каналам. Первый фильтр формирует отклонения углов от их программных значений. Второй фильтр вычисляет управляющие воздействия для компенсации отклонений. Третий фильтр распределяет управляющие воздействия по исполнительным органам (рулевым машинкам) ЛА.

Принципиальный недостаток паттерна связан с необходимостью использования некоторого формата данных, который должен распознаваться всеми фильтрами. Каждое преобразование либо следует согласовывать со смежными фильтрами относительно формата обрабатываемых данных, либо нужно предложить стандартный формат для всех данных. Во втором случае каждый фильтр должен выполнять грамматический разбор входных данных и синтезировать выходные данные в соответствующем формате, при этом вычислительная нагрузка на систему возрастает. В систему нельзя интегрировать фильтры, работающие с несовместимыми форматами данных.

Диалоговые системы на основе каналов и фильтров трудны для написания из-за потребности представления данных в виде потока. Хотя текстовый ввод-вывод

и может быть оформлен таким образом, однако графическим интерфейсам пользователя присущи более сложные форматы и стратегия управления, основанная на событиях (щелчок мыши, выбор пункта меню). Это трудно транслировать в форму потоков данных.

Моделирование управления

Структурная организация системы показывает составляющие ее части — внутренние подсистемы. Для того чтобы подсистемы функционировали как единое целое, ими надо управлять. В структурных моделях нет никаких сведений по управлению, поэтому архитектор должен ввести модель управления, которая дополняла бы имеющуюся модель структуры. В моделях архитектурного управления проектируется поток управления между подсистемами.

Известны два основных типа управления в программных системах: централизованное управление и событийное управление.

При централизованном управлении одна подсистема выделяется как системный контроллер. Ее обязанности — руководить работой других подсистем.

При событийном управлении системой управляют внешние события. На внешние события может реагировать любая подсистема. События, на которые откликается система, могут происходить либо в ее подсистемах, либо во внешнем окружении системы.

Типовые решения управления представляются паттернами управления.

Паттерны управления дополняют структурные паттерны. Все описанные ранее структурные паттерны можно реализовать с помощью централизованного управления или управления, основанного на событиях.

Паттерны централизованного управления

В паттернах централизованного управления одна из подсистем назначается главной и управляет работой других подсистем. Различают две разновидности паттернов, зависящие от режима работы управляемых подсистем (последовательная работа, параллельная работа).

Паттерн вызов-возврат (табл. 6.6). Это известная схема организации вызова программных компонентов «сверху вниз», в которой управление начинается на вершине иерархии компонентов и через вызовы передается на более низкие уровни иерархии. Данный паттерн применим только в системах с последовательным режимом обработки.

Таблица 6.6. Паттерн вызов-возврат

Имя	Вызов-возврат
Описание	Вызов компонентов осуществляется «сверху вниз», то есть управление начинается на вершине иерархии компонентов и через многократные вызовы передается на нижние уровни иерархии
Пример	Иллюстрация организации управления приведена на рис. 6.11
Когда используется	Применима только в системах последовательной обработки, то есть в таких системах, в которых процессы должны протекать последовательно

Имя	Вызов-возврат
Преимущества	Простой анализ потоков управления. Такие системы легче проектировать и тестировать
Недостатки	Сложно обрабатывать исключительные ситуации

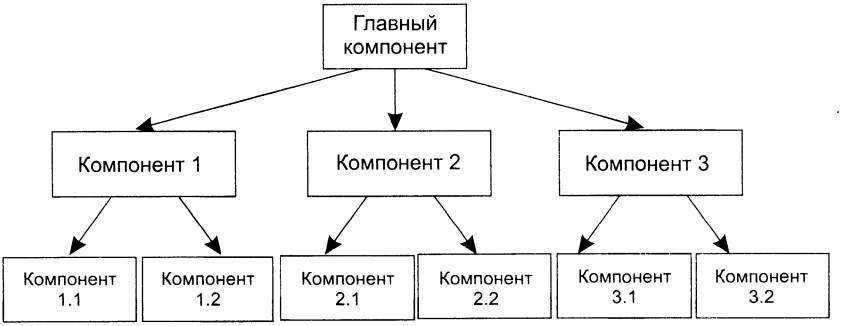


Рис. 6.11. Паттерн вызов-возврат

Организация вызова-возврата представлена на рис. 6.11. Из главного компонента можно вызвать компоненты 1, 2 и 3, из компонента 2 — компоненты 2.1 и 2.2 и т. д. Управление переходит от компонента самого верхнего уровня иерархии к компоненту более низкого уровня. Затем происходит возврат управления в точку вызова компонента. За управление отвечает тот компонент, который выполняется в текущий момент; он может вызывать другие компоненты или вернуть управление вызвавшему его компоненту. Накладные расходы времени при последовательном порядке вызовов и возврате к определенной точке системы велики. Такая схема выполнения компонентов не воспроизводит схему вложенности компонентов (компонент 2.1 не обязательно является частью компонента 2).

Паттерн менеджера (табл. 6.7). Применяется в системах параллельной обработки. Один компонент назначается менеджером и управляет запуском, финализацией и координацией работы других компонентов. Работа «другого» компонента может протекать параллельно работе третьего компонента.

Таблица 6.7. Паттерн менеджера

Имя	Менеджер
Описание	Один системный компонент назначается менеджером и управляет запуском и завершением других компонентов системы, координирует их работу. Компоненты могут работать параллельно
Пример	Иллюстрация системы с менеджером приведена на рис. 6.12
Когда используется	Применяется в системах, в которых необходимо организовать параллельные процессы, но может использоваться и для систем последовательной обработки, в которых менеджер вызывает отдельные подсистемы в зависимости от значений некоторых переменных
Преимущества	Можно использовать в системах реального времени, где нет чересчур строгих временных ограничений (в так называемых «мягких» системах реального времени)
Недостатки	Малая скорость реакции на внешние события

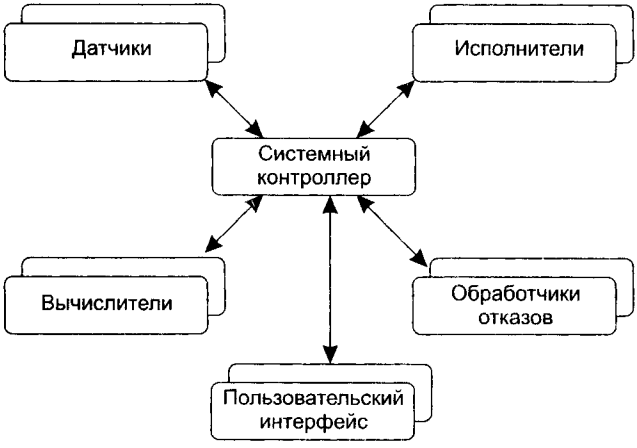


Рис. 6.12. Система реального времени

На рис. 6.12 представлена параллельная система с выделенным контроллером. Подобная модель часто используется в таких системах реального времени, где нет предельных требований по скорости обработки. Системный контроллер управляет работой набора компонентов, связанных с датчиками и исполнительными органами. Контроллер системы, в зависимости от переменных состояния системы, определяет моменты запуска или завершения процессов в компонентах. Он проверяет, генерируются ли в остальных компонентах данные для обработки, и управляет передачей данных компонентам-приемникам. Дополнительно контроллер отслеживает все аварийные ситуации и реагирует на них.

Паттерны событийного управления

При централизованном подходе управление системой обычно зависит от ее состояния. При событийном подходе управление системами основано на внешних событиях (событиях внешней среды) и на аварийных событиях, возникших внутри системы. Рассмотрим два паттерна событийного управления системами.

Паттерн широковещательного управления (табл. 6.8). Здесь каждая подсистема уведомляет обработчика о своем интересе к конкретным событиям. Когда событие происходит, обработчик пересылает его подсистеме, которая может обработать это событие. Функции управления в обработчик не встраиваются.

Таблица 6.8. Паттерн широковещательного управления

Имя	Широковещательное управление
Описание	Все события и сообщения поступают в обработчик, который передает их конкретным адресатам. Адресатом является подсистема, предварительно объявившая о своем интересе к событию. Адресат, который обрабатывает данное событие, отвечает на него
Пример	Иллюстрация системы широковещательного управления приведена на рис. 6.13

Имя	Широковещательное управление
Когда используется	Данный подход эффективен при сетевой интеграции подсистем, распределенных на разные компьютеры
Преимущества	Простота изменения системы, легкость ее размещения
Недостатки	Возможны конфликты в реакции на внешние события

Рис. 6.13. Система широковещательного управления

В системе широковещательного управления (см. рис. 6.13) подсистемы реагируют на определенные события. Если произошло некоторое событие, управление переходит к подсистеме, обрабатывающей данное событие. В отличие от системы с менеджером, здесь алгоритм управления в обработчик событий-сообщений не встраивается. Подсистемы сами определяют требуемые события, а обработчик лишь следит за отправкой событий нужным подсистемам.

Данный паттерн поддерживает простую интеграцию новой подсистемы (достаточно лишь зарегистрировать ее события в обработчике). Подсистемы могут размещаться на разных компьютерах.

Недостатком паттерна является возможный конфликт интересов: заявку на обработку некоторого события могут подать несколько подсистем.

Паттерн управления на основе прерываний (табл. 6.9). Предполагается применение набора специализированных обработчиков, оперативно реагирующих на прерывания — внешние события.

Таблица 6.9. Паттерн управления на основе прерываний

Имя	Управление на основе прерываний
Описание	В системе присутствует набор обработчиков. Специализация обработчиков задается системой. Каждая категория прерываний обслуживается своим обработчиком. Обработчик только регистрирует прерывание, а затем передает управление заранее определенному процессу
Пример	Иллюстрация системы, управляемой прерываниями, приведена на рис. 6.14
Когда используется	Используется в системах реального времени со строгими требованиями по времени реакции. Данный паттерн может быть скомбинирован с паттерном менеджер. Менеджер управляет нормальной работой системы, а в критических ситуациях применяет управление, основанное на прерываниях
Преимущества	Быстрая реакция системы на внешние события
Недостатки	Система сложна в программировании и проверке. При тестировании системы затруднительно имитировать все прерывания. Число прерываний ограничено используемой аппаратурой (после достижения предела, связанного с аппаратными ограничениями, никакие другие прерывания не обрабатываются)

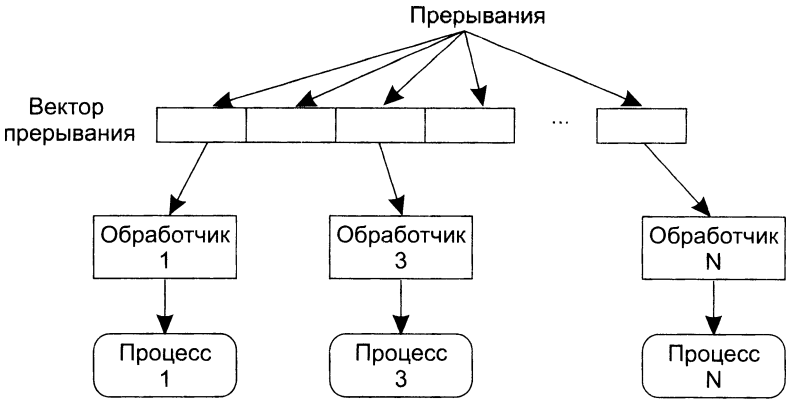


Рис. 6.14. Система, управляемая прерываниями

В системе с управлением на основе прерываний все прерывания разбиты на группы – типы, которые образуют вектор прерываний (см. рис. 6.14). Для каждого типа прерываний известна ячейка памяти, где хранится адрес обработчика прерывания. При получении прерывания аппаратный переключатель немедленно передает управление конкретному обработчику прерывания. После анализа (выясняется, какое событие вызвало прерывание)[1] обработчик запускает необходимые процессы.

Данный паттерн используется в жестких системах реального времени, где требуется немедленная реакция на события.

Декомпозиция подсистем на модули

После проектирования системной структуры и определения принципов управления структурой следует выполнить разделение подсистем на модули. Фактически эту работу можно считать введением в детальное проектирование, которое конкретизирует архитектурные решения. Задача декомпозиции – это задача определения внутреннего содержания каждой подсистемы (компонента). Результатом ее решения является формирование структуры подсистемы – набора модулей и отношений их взаимодействия.

Известны два подхода и два типа моделей для декомпозиции подсистем на модули:

- модель потока данных;
- объектно-ориентированная модель.

В основе модели потока данных лежит разбиение по функциям. При выборе этой модели получим набор функциональных модулей и определим связи между ними. Например, результат может быть похож на реализацию диаграммы потоков данных.

[1] Одному типу прерывания может соответствовать несколько сигналов прерывания и, соответственно, несколько внешних событий.

Объектно-ориентированная модель основана на объектах (слабо сцепленных сущностях, имеющих собственные наборы данных, состояния, наборы операций) и их описаниях — классах.

Применение модели потока данных в ходе проектирования рассмотрим в следующей главе. Использование объектно-ориентированной модели будет подробно обсуждаться во многих последующих главах.

Какой тип декомпозиции следует выбирать? Ответ на этот вопрос зависит от большого количества факторов, в том числе и от сложности разбиваемой подсистемы.

ВНИМАНИЕ

Когда мы говорим о проектных действиях, проектных решениях и проектных результатах, мы имеем в виду действия, решения и результаты этапа *проектирования* (в жизненном цикле процесса разработки). Речь не идет о действиях и артефактах других этапов *программного проекта*. Проектирование — это лишь один из этапов программного проекта.

Разделение понятий

В 1972 году великий Э. Дейкстра ввел очень важный принцип проектирования — разделение понятий (separation of concerns) [3]:

Любую сложную проблему проще понять, разделив ее на части, каждую из которых можно решать и оптимизировать независимо.

Понятие — это черта (свойство, отличительная особенность), или поведение, которое определяется как часть модели требований к ПО. Разделяя понятия на небольшие и поэтому более управляемые части, мы экономим усилия и время на решение проблемы. Проиллюстрируем эту точку зрения.

Пусть $C(x)$ — функция сложности решения проблемы x, $T(x)$ — функция затрат времени на решение проблемы x. Для двух проблем p_1 и p_2 из соотношения $C(p_1) > C(p_2)$ следует, что

$$T(p_1) > T(p_2). \tag{6.1}$$

Этот вывод интуитивно ясен: решение сложной проблемы требует большего времени.

Далее. Из практики решения проблем человеком следует

$$C(p_1 + p_2) > C(p_1) + C(p_2).$$

Отсюда, с учетом соотношения (6.1), запишем

$$T(p_1 + p_2) > T(p_1) + T(p_2). \tag{6.2}$$

Соотношение (6.2) — это обоснование модульности. Оно приводит к заключению «разделяй и властвуй» — сложную проблему легче решить, разделив ее на управляемые части. Результат, выраженный неравенством (6.2), имеет важное значение для модульности и ПО. Фактически это аргумент в пользу модульности.

Разделение понятий провозглашается во многих принципах проектирования, например в модульности, аспектах и пошаговой детализации. Каждый из этих принципов мы обсудим в следующих разделах.

Модульность

Модульность — это наиболее общая демонстрация разделения понятий. Программная система делится на именуемые и адресуемые компоненты, часто называемые модулями, которые затем интегрируются для совместного решения проблемы.

Модуль — фрагмент программного текста, являющийся строительным блоком для физической структуры системы. Как правило, модуль состоит из интерфейсной части и части-реализации.

Модульность — свойство системы, которая может подвергаться декомпозиции на ряд внутренне связанных и слабо зависящих друг от друга модулей.

По определению Г. Майерса модульность — свойство ПО, обеспечивающее интеллектуальную возможность создания сколь угодно сложной программы [76]. Монолитное ПО, то есть состоящее из одного модуля, не поддается восприятию программным инженером. Огромное количество альтернативных ветвей, множество ссылок, переменных, наконец, общая сложность во многих случаях недоступны для понимания. Вы вынуждены разбивать «монолит» на модули в расчете на упрощение понимания и, как следствие, на уменьшение стоимости создания ПО.

Возвращаясь к обсуждению разделения понятий, заключаем, что путем последовательного дробления ПО можно добиться, что затраты на разработку станут ничтожно малы. Однако это отражает лишь часть реальности — ведь здесь не учитываются затраты на дальнейшую интеграцию модулей. Как показано на рис. 6.15, с увеличением количества модулей (и уменьшением их размера) такие затраты также растут.

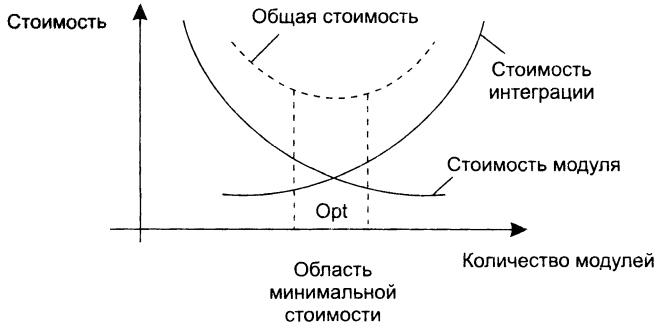


Рис. 6.15. Затраты на модульность

Таким образом, существует оптимальное количество модулей Opt, которое приводит к минимальной стоимости разработки. Увы, у нас нет необходимого опыта для гарантированного предсказания Opt. Впрочем, разработчики знают, что оптимальный модуль должен удовлетворять двум критериям:

- снаружи он проще, чем внутри;
- его проще использовать, чем построить.

Информационная закрытость

Принцип информационной закрытости (автор – Д. Парнас, 1972) утверждает: содержание модулей должно быть скрыто друг от друга [84]. Как показано на рис. 6.16, модуль должен определяться и проектироваться так, чтобы его содержимое (процедуры и данные) было недоступно тем модулям, которые не нуждаются в такой информации (клиентам).

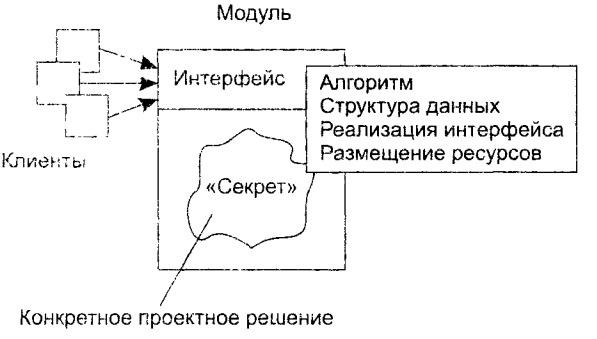


Рис. 6.16. Информационная закрытость модуля

Информационная закрытость означает:

1) все модули независимы, обмениваются только информацией, необходимой для работы;
2) доступ к операциям и структурам данных модуля ограничен.

Достоинства информационной закрытости:

- обеспечивается возможность разработки модулей различными независимыми коллективами;
- обеспечивается легкая модификация системы (вероятность распространения ошибок очень мала, так как большинство данных и процедур скрыто от других частей системы).

Идеальный модуль играет роль черного ящика, содержимое которого невидимо клиентам. Он прост в использовании – количество «ручек и органов управления» им невелико (аналогия с эксплуатацией телевизора). Его легко развивать и корректировать в процессе сопровождения программной системы. Для обеспечения таких возможностей система внутренних и внешних связей модуля должна отвечать особым требованиям. Обсудим характеристики внутренних и внешних связей модуля.

Связность модуля

Связность модуля (Cohesion) – это мера зависимости его частей [82, 95, 104]. Связность – внутренняя характеристика модуля. Чем выше связность модуля, тем лучше результат проектирования, то есть тем «черней» его ящик (капсула,

защитная оболочка модуля), тем меньше «ручек управления» на нем находится и тем проще эти «ручки».

Для измерения связности используют понятие силы связности (СС). Существует 7 типов связности:

1. **Связность по совпадению** (СС = 0). В модуле отсутствуют явно выраженные внутренние связи. Например, разработчик спешил на обед, вот и забросил в капсулу модуля все элементы, которые были у него на столе.
2. **Логическая связность** (СС = 1). Части модуля объединены по принципу функционального подобия. Например, модуль состоит из разных подпрограмм обработки ошибок. При использовании такого модуля клиент выбирает только одну из подпрограмм.

   Недостатки:

   - сложное сопряжение;
   - большая вероятность внесения ошибок при изменении сопряжения ради одной из функций.
3. **Временная связность** (СС = 3). Части модуля не связаны, но необходимы в один и тот же период работы системы. Например, в модуле «Утро» могут быть элементы «умыться», «одеться», «позавтракать».

   Недостаток: сильная взаимная связь с другими модулями, отсюда — сильная чувствительность к внесению изменений.
4. **Процедурная связность** (СС = 5). Части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения. Например, в модуле «Одеваться» могут быть элементы «надеть нижнее белье», «надеть верхнюю одежду».
5. **Коммуникативная связность** (СС = 7). Части модуля связаны по данным (работают с одной и той же структурой данных). Например, в модуле «Анализ текста» могут быть элементы «подсчитать количество гласных», «подсчитать количество согласных», «подсчитать количество слов».
6. **Информационная (последовательная) связность** (СС = 9). Выходные данные одной части используются как входные данные в другой части модуля. Например, в модуле «Обработка массива» могут быть элементы «инициировать массив», «упорядочить массив», «распечатать массив».
7. **Функциональная связность** (СС = 10). Части модуля вместе реализуют одну функцию. Функция может быть предельно простой, может быть сложной, то есть распадаться на многие части, но с точки зрения внешнего клиента — это всегда единое действие.

Отметим, что типы связности 1–3 — результат неправильного планирования проектного решения, а тип связности 4 — результат небрежного планирования проектного решения приложения.

Общая характеристика типов связности представлена в табл. 6.10.

Таблица 6.10. Характеристика связности модуля

Тип связности	Сопровождаемость	Роль модуля
Функциональная	Лучшая	«Черный ящик»
Информационная (последовательная)		Не совсем «черный ящик»
Коммуникативная		«Серый ящик»
Процедурная	Худшая	«Белый» или «просвечивающийся ящик»
Временная		«Белый ящик»
Логическая		
По совпадению		

Функциональная связность

Функционально связный модуль содержит элементы, участвующие в выполнении одной и только одной проблемной задачи. Примеры функционально связных модулей:

- Вычислять синус угла;
- Проверять орфографию;
- Читать запись файла;
- Вычислять координаты цели;
- Вычислять зарплату сотрудника;
- Определять место пассажира.

Каждый из этих модулей имеет единичное назначение. Когда клиент вызывает модуль, выполняется только одна работа, без привлечения внешних обработчиков. Например, модуль Определять место пассажира должен делать только это; он не должен распечатывать заголовки страницы.

Некоторые из функционально связных модулей очень просты (например, Вычислять синус угла или Читать запись файла), другие сложны (например, Вычислять координаты цели). Модуль Вычислять синус угла, очевидно, реализует единичную функцию, но как может модуль Вычислять зарплату сотрудника выполнять только одно действие? Ведь каждый знает, что приходится определять начисленную сумму, вычеты по рассрочкам, подоходный налог, социальный налог, алименты и т. д.! Дело в том, что несмотря на сложность модуля и на то, что его обязанность исполняют несколько подфункций, если его действия можно представить как единую проблемную функцию (с точки зрения клиента), тогда считают, что модуль функционально связен.

Приложения, построенные из функционально связных модулей, легче всего сопровождать. Соблазнительно думать, что любой модуль можно рассматривать как однофункциональный, но не надо заблуждаться. Существует много разновидностей модулей, которые выполняют для клиентов перечень различных работ, и этот перечень нельзя рассматривать как единую проблемную функцию. Критерий при

определении уровня связности этих нефункциональных модулей — как связаны друг с другом различные действия, которые они исполняют.

Информационная связность

При информационной (последовательной) связности элементы-обработчики модуля образуют конвейер для обработки данных — результаты одного обработчика используются как исходные данные для следующего обработчика. Приведем пример:

```
Модуль Прием и проверка записи
    прочитать запись из файла
    проверить контрольные данные в записи
    удалить контрольные поля в записи
    вернуть обработанную запись
Конец модуля
```

В этом модуле три элемента. Результаты первого элемента (прочитать запись из файла) используются как входные данные для второго элемента (проверить контрольные данные в записи) и т. д.

Сопровождать модули с информационной связностью почти так же легко, как и функционально связные модули. Правда, возможности повторного использования здесь ниже, чем в случае функциональной связности. Причина — совместное применение действий модуля с информационной связностью полезно далеко не всегда.

Коммуникативная связность

При коммуникативной связности элементы-обработчики модуля используют одни и те же данные, например внешние данные. Пример коммуникативно связного модуля:

```
Модуль Отчет и средняя зарплата
    используется Таблица зарплаты служащих
    сгенерировать Отчет по зарплате
    вычислить параметр Средняя зарплата
    вернуть Отчет по зарплате, Средняя зарплата
Конец модуля
```

Здесь все элементы модуля работают со структурой Таблица зарплаты служащих.

С точки зрения клиента проблема применения коммуникативно связного модуля состоит в избыточности получаемых результатов. Например, клиенту требуется только отчет по зарплате, он не нуждается в значении средней зарплаты. Такой клиент будет вынужден выполнять избыточную работу — выделение в полученных данных материала отчета. Почти всегда разбиение коммуникативно связного модуля на отдельные, функционально связные модули улучшает сопровождаемость системы.

Попытаемся провести аналогию между информационной и коммуникативной связностью.

Модули с коммуникативной и информационной связностью подобны в том, что содержат элементы, связанные по данным. Их удобно использовать, потому что

лишь немногие элементы в этих модулях связаны с внешней средой. Главное различие между ними — информационно связный модуль работает подобно сборочной линии; его обработчики действуют в определенном порядке; в коммуникативно связном модуле порядок выполнения действий безразличен. В нашем примере не имеет значения, когда генерируется отчет (до, после или одновременно с вычислением средней зарплаты).

Процедурная связность

При достижении процедурной связности мы попадаем в пограничную область между хорошей сопровождаемостью (для модулей с более высокими уровнями связности) и плохой сопровождаемостью (для модулей с более низкими уровнями связности). Процедурно связный модуль состоит из элементов, реализующих независимые действия, для которых задан порядок работы, то есть порядок передачи управления. Зависимости по данным между элементами нет. Например:

```
Модуль Вычисление средних значений
    используется Таблица-А, Таблица-В
    вычислить среднее по Таблица-А
    вычислить среднее по Таблица-В
    вернуть среднееТабл-А, среднееТабл-В
Конец модуля
```

Этот модуль вычисляет средние значения для двух полностью несвязанных таблиц Таблица-А и Таблица-В, каждая из которых имеет по 300 элементов.

Теперь представим себе программиста, которому поручили реализовать данный модуль. Соблазнившись возможностью минимизации кода (использовать один цикл в интересах двух обработчиков, ведь они находятся внутри единого модуля!), программист пишет:

```
Модуль Вычисление средних значений
    используется Таблица-А, Таблица-В
    суммаТабл-А := 0
    суммаТабл-В := 0
    для i := 1 до 300
        суммаТабл-А := суммаТабл-А + Таблица-А(i)
        суммаТабл-В := суммаТабл-В + Таблица-В(i)
    конец для
    среднееТабл-А := суммаТабл-А / 300
    среднееТабл-В := суммаТабл-В / 300
    вернуть среднееТабл-А, среднееТабл-В
Конец модуля
```

Для процедурной связности этот случай типичен — независимый (на уровне проблемы) код стал зависимым (на уровне реализации). Прошли годы, продукт сдали заказчику. И вдруг возникла задача сопровождения — модифицировать модуль под уменьшение размера таблицы В. Оцените, насколько удобно ее решать.

Временная связность

При связности по времени элементы-обработчики модуля привязаны к конкретному периоду времени (из жизни программной системы).

Классическим примером временной связности является модуль инициализации:

```
Модуль Инициализировать Систему
    перемотать магнитную ленту 1
    Счетчик магнитной ленты 1 := 0
    перемотать магнитную ленту 2
    Счетчик магнитной ленты 2 := 0
    Таблица текущих записей := пробел..пробел
    Таблица количества записей := 0..0
    Переключатель 1 := выкл
    Переключатель 2 := вкл
Конец модуля
```

Элементы данного модуля почти не связаны друг с другом (за исключением того, что должны выполняться в определенное время). Они все — часть программы запуска системы. Зато элементы более тесно взаимодействуют с другими модулями, что приводит к сложным внешним связям.

Модуль со связностью по времени испытывает те же трудности, что и процедурно связный модуль. Программист соблазняется возможностью совместного использования кода (действиями, которые связаны только по времени), модуль становится трудно использовать повторно.

Так, при желании инициализировать магнитную ленту 2 в другое время вы столкнетесь с неудобствами. Чтобы не сбрасывать всю систему, придется или ввести флажки, указывающие инициализируемую часть, или написать другой код для работы с лентой 2. Оба решения ухудшают сопровождаемость.

Процедурно связные модули и модули с временной связностью очень похожи. Степень их непрозрачности изменяется от темно-серого до светло-серого цвета, так как трудно объявить функцию такого модуля без перечисления ее внутренних деталей. Различие между ними подобно различию между информационной и коммуникативной связностью. Порядок выполнения действий более важен в процедурно связных модулях. Кроме того, процедурные модули имеют тенденцию к совместному использованию циклов и ветвлений, а модули с временной связностью чаще содержат более линейный код.

Логическая связность

Элементы логически связного модуля принадлежат к действиям одной категории, и из этой категории клиент выбирает выполняемое действие. Рассмотрим следующий пример:

```
Модуль Пересылка сообщения
    переслать по электронной почте
    переслать по факсу
    послать в телеконференцию
    переслать по ftp-протоколу
Конец модуля
```

Как видим, логически связный модуль — мешок доступных действий. Действия вынуждены совместно использовать один и тот же интерфейс модуля. В строке вызова модуля значение каждого параметра зависит от используемого действия. При вызове отдельных действий некоторые параметры должны иметь значение пробела, нулевые значения и т. д. (хотя клиент все же должен использовать их и знать их типы).

Действия в логически связном модуле попадают в одну категорию, хотя имеют не только сходства, но и различия. К сожалению, это заставляет программиста «завязывать код действий в узел», ориентируясь на то, что действия совместно используют общие строки кода. Поэтому логически связный модуль имеет:

- уродливый внешний вид с различными параметрами, обеспечивающими, например, четыре вида доступа;
- запутанную внутреннюю структуру со множеством переходов, похожую на волшебный лабиринт.

В итоге модуль становится сложным как для понимания, так и для сопровождения.

Связность по совпадению

Элементы связного по совпадению модуля вообще не имеют никаких отношений друг с другом:

```
Модуль Разные функции (какие-то параметры)
    поздравить с Новым годом (...)
    проверить исправность аппаратуры (...)
    заполнить анкету героя (...)
    измерить температуру (...)
    вывести собаку на прогулку (...)
    запастись продуктами (...)
    приобрести Ягуар (...)
конец модуля
```

Связный по совпадению модуль похож на логически связный модуль. Его элементы-действия не связаны ни потоком данных, ни потоком управления. Но в логически связном модуле действия, по крайней мере, относятся к одной категории; в связном по совпадению модуле даже это не так. Словом, связные по совпадению модули имеют все недостатки логически связных модулей и даже усиливают их. Применение таких модулей вселяет ужас, поскольку один параметр используется для разных целей.

Чтобы клиент мог воспользоваться модулем Разные функции, этот модуль (подобно всем связным по совпадению модулям) должен быть «белым ящиком», чья реализация полностью видима. Такие модули делают системы менее понятными и труднее сопровождаемыми, чем системы без модульности вообще!

К счастью, связность по совпадению встречается редко. Среди ее причин можно назвать:

- бездумный перевод существующего монолитного кода в модули;
- необоснованные изменения модулей с плохой (обычно временной) связностью, приводящие к добавлению флажков.

Определение связности модуля

Приведем алгоритм определения уровня связности модуля.

1. Если модуль — единичная проблемно-ориентированная функция, то уровень связности — функциональный; конец алгоритма. В противном случае перейти к пункту 2.

2. Если действия внутри модуля связаны, то перейти к пункту 3. Если действия внутри модуля никак не связаны, то перейти к пункту 6.
3. Если действия внутри модуля связаны данными, то перейти к пункту 4. Если действия внутри модуля связаны потоком управления, перейти к пункту 5.
4. Если порядок действий внутри модуля важен, то уровень связности – информационный. В противном случае уровень связности – коммуникативный. Конец алгоритма.
5. Если порядок действий внутри модуля важен, то уровень связности – процедурный. В противном случае уровень связности – временной. Конец алгоритма.
6. Если действия внутри модуля принадлежат к одной категории, то уровень связности – логический. Если действия внутри модуля не принадлежат к одной категории, то уровень связности – по совпадению. Конец алгоритма.

Возможны более сложные случаи, когда с модулем ассоциируются несколько уровней связности. В этих случаях следует применять одно из двух правил:

- *правило параллельной цепи.* Если все действия модуля имеют несколько уровней связности, то модулю присваивают самый сильный уровень связности;
- *правило последовательной цепи.* Если действия в модуле имеют разные уровни связности, то модулю присваивают самый слабый уровень связности.

Например, модуль может содержать некоторые действия, которые связаны процедурно, а также другие действия, связные по совпадению. В этом случае применяют правило последовательной цепи и в целом модуль считают связным по совпадению.

Сцепление модулей

Сцепление (Coupling) – мера взаимозависимости модулей по данным [82, 95, 104]. Сцепление – внешняя характеристика модуля, которую желательно уменьшать.

Количественно сцепление измеряется степенью сцепления СЦ. Выделяют 6 типов сцепления.

1. **Сцепление по данным** (СЦ = 1). Модуль А вызывает модуль В.

   Все входные и выходные параметры вызываемого модуля – простые элементы данных (рис. 6.17).

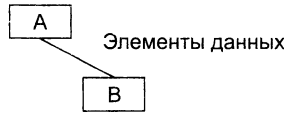


Рис. 6.17. Сцепление по данным

2. **Сцепление по образцу** (СЦ = 3). В качестве параметров используются структуры данных (рис. 6.18).
3. **Сцепление по управлению** (СЦ = 4). Модуль А явно управляет функционированием модуля В (с помощью флагов или переключателей), посылая ему управляющие данные (рис. 6.19).

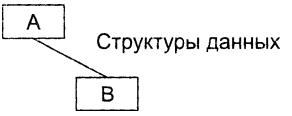


Рис. 6.18. Сцепление по образцу

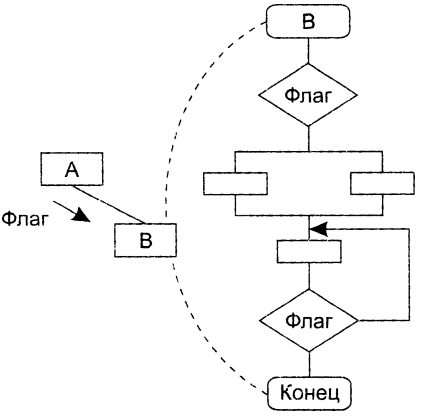


Рис. 6.19. Сцепление по управлению

4. **Сцепление по внешним ссылкам** (СЦ = 5). Модули A и B ссылаются на один и тот же глобальный элемент данных.
5. **Сцепление по общей области** (СЦ = 7). Модули разделяют одну и ту же глобальную структуру данных (рис. 6.20).
6. **Сцепление по содержанию** (СЦ = 9). Один модуль прямо ссылается на содержание другого модуля (не через его точку входа). Например, коды их команд перемежаются друг с другом (см. рис. 6.20).

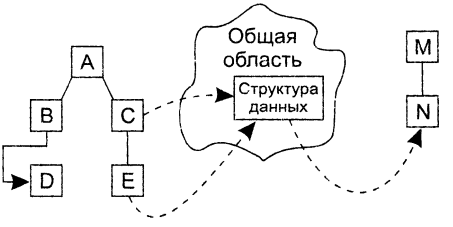


Рис. 6.20. Сцепление по общей области и содержанию

На рис. 6.20 видим, что модули B и D сцеплены по содержанию, а модули C, E и N сцеплены по общей области.

Сложность программной системы

В простейшем случае сложность программной системы (ПС) определяется как сумма мер сложности ее модулей. Сложность модуля может вычисляться различными способами.

Например, М. Холстед (1977) предложил меру длины N модуля [53]:

$$N \approx n_1 \log_2(n_1) + n_2 \log_2(n_2),$$

где n_1 — число различных операторов, n_2 — число различных операндов.

В качестве второй метрики М. Холстед рассматривал объем V модуля (количество символов для записи всех операторов и операндов текста программы):

$$V = N \times \log_2(n_1 + n_2).$$

Вместе с тем известно, что любая сложная система состоит из элементов и системы связей между элементами и что игнорировать внутрисистемные связи неразумно.

Том МакКейб (1976) при оценке сложности ПС предложил исходить из топологии внутренних связей [73]. Для этой цели он разработал метрику цикломатической сложности:

$$V(G) = E - N + 2,$$

где E — количество дуг, а N — количество вершин в управляющем графе ПС.

Это был шаг в нужном направлении. Дальнейшее уточнение оценок сложности потребовало, чтобы каждый модуль мог представляться как локальная структура, состоящая из элементов и связей между ними.

Таким образом, при комплексной оценке сложности ПС необходимо рассматривать меру сложности модулей, меру сложности внешних связей (между модулями) и меру сложности внутренних связей (внутри модулей) [48, 80]. Традиционно со внешними связями сопоставляют характеристику «сцепление», а с внутренними связями — характеристику «связность».

Вопросы комплексной оценки сложности обсудим в следующем разделе.

Характеристики иерархической структуры программной системы

Если принять, что в системе реализован паттерн централизованного управления «вызов-возврат», то основным результатом архитектурного проектирования становится иерархическая структура программной системы. Она определяет состав модулей ПС и управляющие отношения между модулями. В этой структуре модуль более высокого уровня (начальник) управляет модулем нижнего уровня (подчиненным).

Иерархическая структура не отражает процедурные особенности программной системы, то есть последовательность операций, их повторение, ветвления и т. д. Рассмотрим основные характеристики иерархической структуры, представленной на рис. 6.21.

Первичными характеристиками являются количество вершин n (модулей) и количество ребер e (связей между модулями). К ним добавляются две глобальные характеристики — высота и ширина:

- *высота* (*depth*) — количество уровней управления (количество вершин в самом длинном пути от вершины-корня до вершины-листа);
- *ширина* (*width*) — максимальное из количеств модулей, размещенных на уровнях управления.

В нашем примере *высота* = 4, *ширина* = 6.

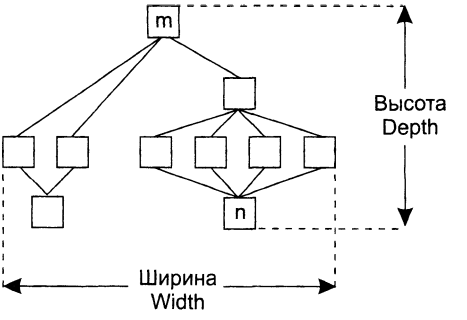


Рис. 6.21. Иерархическая структура программной системы

Н. Фентон [48] предлагает ряд простых морфологических метрик, которые позволяют сравнивать программные архитектуры на основе прямых измерений:

- *Размер Size* = n, где n — количество вершин архитектуры. Для архитектуры на рис. 6.21 *Size* = 10.
- *Плотность взаимодействия* $R = e/n$ (отношение ребер к вершинам), измеряет сцепление в архитектуре. Для архитектуры на рис. 6.21 $R = 1.3$.
- *Уровень взаимодействия.* Определяется как невязка структуры *Nev*. Формула для невязки рассматривается ниже.
- *Индивидуальный уровень взаимодействия* Com_i. Вычисляет степень взаимодействия i-й вершины с другими вершинами. Определяется как сумма коэффициентов *Fan_in*(i) и *Fan_out*(i), которые поясняются ниже.
- *Средний уровень взаимодействия Av_Com.* Вычисляется как среднее значение всех индивидуальных уровней взаимодействия архитектуры.

Возникает вопрос: как оценить качество топологии архитектуры? Из практики проектирования известно, что лучшее решение обеспечивается иерархической структурой в виде дерева.

Степень отличия реальной проектной структуры от дерева характеризуется невязкой структуры. Как определить невязку?

Вспомним, что полный граф (complete graph) с n вершинами имеет количество ребер

$$e_c = n(n-1)/2,$$

а дерево (tree) с таким же количеством вершин — существенно меньшее количество ребер

$$e_t = n-1.$$

Тогда формулу невязки можно построить, сравнивая количество ребер полного графа, реального графа и дерева.

Для проектной структуры с n вершинами и e ребрами невязка определяется по выражению:

$$Nev = \frac{e - e_t}{e_c - e_t} = \frac{(e - n + 1) \times 2}{n \times (n - 1) - 2 \times (n - 1)} = \frac{2 \times (e - n + 1)}{(n - 1) \times (n - 2)}.$$

Значение невязки лежит в диапазоне от 0 до 1. Если $Nev = 0$, то проектная структура является деревом, если $Nev = 1$, то проектная структура – полный граф.

Ясно, что невязка дает грубую оценку структуры. Для увеличения точности оценки следует применить характеристики связности и сцепления.

Хорошая структура должна иметь низкое сцепление и высокую связность.

Локальными характеристиками модулей структуры являются коэффициент объединения по входу и коэффициент разветвления по выходу.

Коэффициент объединения по входу $Fan\_in(i)$ — это количество модулей, которые прямо управляют i-м модулем.

В примере для модуля n: $Fan\_in(n) = 4$.

Коэффициент разветвления по выходу $Fan\_out(i)$ — это количество модулей, которыми прямо управляет i-й модуль.

В примере для модуля m: $Fan\_out(m) = 3$.

Л. Констентайн и Э. Йордан (1979) предложили оценивать структуру с помощью коэффициентов $Fan\_in(i)$ и $Fan\_out(i)$ модулей [104].

Большое значение $Fan\_in(i)$ – свидетельство высокого сцепления, так как является мерой зависимости модуля. Большое значение $Fan\_out(i)$ говорит о высокой сложности вызывающего модуля. Причиной является то, что для координации подчиненных модулей требуется сложная логика управления.

Основной недостаток коэффициентов $Fan\_in(i)$ и $Fan\_out(i)$ состоит в игнорировании веса связи. Здесь рассматриваются только управляющие потоки (вызовы модулей). В то же время информационные потоки, нагружающие ребра структуры, могут существенно изменяться, поэтому нужна мера, которая учитывает не только количество ребер, но и количество информации, проходящей через них.

С. Генри и Д. Кафура (1981) ввели информационные коэффициенты $ifan\_in(i)$ *и* $ifan\_out(j)$ [56]. Они учитывают количество элементов и структур данных, из которых i-й модуль берет информацию и которые обновляются j-м модулем соответственно.

Информационные коэффициенты суммируются со структурными коэффициентами $sfan\_in(i)$ и $sfan\_out(j)$, которые учитывают только вызовы модулей.

В результате формируются полные значения коэффициентов:

$$Fan\_in(i) = sfan\_in(i) + ifan\_in(i),$$

$$Fan\_out(j) = sfan\_out(j) + ifan\_out(j).$$

На основе полных коэффициентов модулей вычисляется метрика общей сложности структуры:

$$S = \sum_{i=1}^{n} length(i) \times (Fan\_in(i) + Fan\_out(i))^2,$$

где $length(i)$ – оценка размера i-го модуля (в виде LOC или FP-оценки).

Д. Кард и Р. Гласс [41] определяют три метрики для измерения сложности проектного решения: структурную сложность, сложность по данным и системную сложность.

Структурная сложность i-го модуля определяется по следующей формуле:

$$S(i) = Fan\_out(i)^2.$$

Сложность по данным показывает сложность внешнего интерфейса i-го модуля:

$$D(i) = \frac{v(i)}{Fan\_out(i) + 1},$$

где $v(i)$ – количество входных и выходных переменных i-го модуля.

Наконец, *системная сложность* определяется как сумма структурной сложности и сложности по данным:

$$C(i) = S(i) + D(i).$$

Рост значения любой из метрик сложности приводит к росту общей архитектурной сложности. Это, в свою очередь, увеличивает вероятность роста затрат на интеграцию и тестирование.

Пошаговая детализация

Пошаговая детализация – это нисходящая стратегия разработки, предложенная Н. Виртом [103]. Программа разрабатывается путем добавления уровней процедурной детализации. Иерархия создается путем многошаговой декомпозиции макроопределения функции (процедурной абстракции) на составные детализирующие части. Процесс прекращается при достижении уровня операторов языка программирования.

Детализация по сути является процессом развития. Вы начинаете с определения функции (или описания информации) на высоком уровне абстракции. Это означает, что определение описывает функцию или информацию концептуально, без объяснения внутренней работы функции или внутренней структуры информации. Далее вы развиваете оригинальное определение, добавляя новые уточнения на каждом шаге детализации.

Абстракция и детализация дополняют друг друга. Абстракция позволяет описывать процедуру или данные изнутри, но побуждает непрофессионалов к получению знаний о деталях. Детализация помогает переходить на более низкий уровень, обеспечивающий дополнительные уточнения и развивающий проектное решение. Оба принципа способствуют созданию полной проектной модели как эволюционного итога проектирования.

Аспекты

В ходе анализа требований обнаруживается набор понятий. Эти понятия включают требования, варианты использования, характеристики, структуры данных, границы интеллектуальной собственности, сотрудничества, паттерны и контракты. В идеале модель требований должна быть организована так, чтобы позволить вам изолировать каждое понятие (требование) и рассматривать его отдельно (независимо от других). На практике некоторые из этих понятий пересекают всю систему и не поддаются изоляции.

С началом проектирования требования трансформируются в модульные проектные представления. Рассмотрим два требования *X* и *У*. Требование *У* пересекает требование *X*, если в программной декомпозиции, которая выбрана для *X*, нельзя реализовать *X* без учета *У*.

Например, требование *X* задает «*функцию отображения выполнения заказа в электронном магазине*». Требование же *У* определяет: «*зарегистрированный пользователь должен быть проверен перед любым использованием средств электронного магазина*». Это требование применимо ко всем функциям, которые доступны зарегистрированным пользователям. В итоге проектирования мы получим проектное представление (решение) для требования *X* и проектное представление (решение) для требования *У*. Поскольку эти представления реализуют конкретные понятия, представление для *У* пересекает представление для *X*.

Представлением пересекающего понятия является аспект. Поэтому проектным представлением для требования «*зарегистрированный пользователь должен быть проверен перед любым использованием средств электронного магазина*» становится аспект этого электронного магазина. Очень важно вовремя выявить все аспекты, чтобы в ходе проектирования можно было применять их по мере детализации и оформления модульных решений. Аспект — это отдельный программный модуль, который *сплетается* со многими другими модулями. Аспект содержит не только функциональность, которую он должен поместить в другие модули-приемники, но и указания о том, куда он должен *вплести* свою функциональность.

Рефакторинг

Рефакторинг — важное проектное действие, востребованное во многих гибких процессах разработки. Рефакторинг является техникой реорганизации, упрощения структуры проектного решения или программного кода в компоненте (без изменения его функций или поведения). Известный авторитет в этой области М. Фаулер дает такое определение [16]:

«Рефакторинг представляет собой процесс такого изменения программной системы, при котором не меняется внешнее поведение кода, но улучшается его внутренняя структура. Это способ систематического приведения кода в порядок, при котором шансы появления новых ошибок минимальны. В сущности, при проведении рефакторинга кода вы улучшаете его дизайн уже после того, как он написан».

При рефакторинге в существующем проектном решении ищется избыточность, неиспользуемые элементы, неэффективные алгоритмы, плохие структуры данных и другие характеристики и элементы, которые можно модифицировать для улуч-

ления качества системы. Например, обнаруживается низкая связность нескольких функций внутри компонента. После тщательного анализа принимается решение разделить компонент на несколько компонентов с высокой связностью. Далее итог рефакторинга подвергается автоматическому тестированию. В результате получают продукт, который легче интегрировать и сопровождать. В дальнейшем мы приведем подробный пример применения этой технологии.

Контрольные вопросы и упражнения

1. Какова цель синтеза программной системы? Перечислите этапы синтеза.
2. Дайте определение модели данных, модели архитектуры и модели подсистем.
3. Какие особенности имеет этап проектирования?
4. Решение каких задач обеспечивает архитектурное проектирование?
5. Что такое архитектурный паттерн? Зачем его применяют?
6. Какие паттерны системного структурирования вы знаете?
7. Чем отличается паттерн клиент-сервер от трехъярусного паттерна?
8. Чем отличается паттерн клиент-сервер от многоуровневого паттерна?
9. Какие паттерны управления вы знаете?
10. Какие существуют разновидности паттернов централизованного управления?
11. Поясните разновидности паттернов событийного управления.
12. Поясните принцип разделения понятий.
13. Поясните понятия модуля и модульности. Зачем используют модули?
14. В чем состоит принцип информационной закрытости? Какие достоинства он имеет?
15. Что такое связность модуля?
16. Какие существуют типы связности?
17. Дайте характеристику функциональной связности.
18. Дайте характеристику информационной связности.
19. Охарактеризуйте коммуникативную связность.
20. Охарактеризуйте процедурную связность.
21. Дайте характеристику временной связности.
22. Дайте характеристику логической связности.
23. Охарактеризуйте связность по совпадению.
24. Что значит «улучшать связность»?
25. Определите связность модуля «Начало дня», имеющего следующее описание: встать, умыться, одеться, позавтракать.
26. Определите связность модуля «Одеваться», имеющего следующее описание: надеть нижнее белье, надеть брюки и рубашку, надеть пиджак, надеть пальто.

27. Определите связность модуля «Анализ текста», имеющего следующее описание: подсчитать количество гласных букв, подсчитать количество согласных букв, подсчитать количество предложений.
28. Определите связность модуля «Написать учебник», имеющего следующее описание: написать параграфы, отформатировать параграфы, собрать из параграфов главы, написать контрольные вопросы к главам.
29. Что такое сцепление модуля?
30. Какие существуют типы сцепления?
31. Дайте характеристику сцепления по данным.
32. Дайте характеристику сцепления по образцу.
33. Охарактеризуйте сцепление по управлению.
34. Охарактеризуйте сцепление по внешним ссылкам.
35. Дайте характеристику сцепления по общей области.
36. Дайте характеристику сцепления по содержанию.
37. Что значит «улучшать сцепление»?
38. Дан псевдокод программы:

```
Program Main
Module A;
        Begin A
            …
            Вызов B(x, y);
        End A;
Module B (n:Integer; m:Real);
        Begin B
            …
        End B;
Begin Main
    Вызов A;
End Main.
```

Определите сцепление модулей A и B.

39. Дан псевдокод программы:

```
Program Main
Module A;
        Begin A
            …
            Вызов B(x, y);
        End A;
Module B (n:array (1..10) of Integer; m:Real);
        Begin B
            …
        End B;
Begin Main
    Вызов A;
End Main.
```

Определите сцепление модулей A и B.

Дан псевдокод программы:

```
Program Main
Var Glob:Real;
Module A;
        Begin A
            Glob:=3.14;
            Вызов B;
        End A;
Module B;
        Begin B
            …
            x:=Glob;
        End B;
Begin Main
    Вызов A;
End Main.
```

Определите сцепление модулей A и B.

Какие подходы к оценке сложности системы вы знаете?

Что определяет иерархическая структура программной системы?

Поясните подходы к оценке иерархической структуры проектирования.

Поясните понятия коэффициента объединения по входу и коэффициента разветвления по выходу.

Что определяет невязка структуры?

Поясните информационные коэффициенты объединения и разветвления.

В чем суть пошаговой детализации?

Что такое пересекающее понятие?

В чем заключается специфика аспектов?

С какой целью выполняется рефакторинг?

Глава 7

Классические методы проектирования

В этой главе рассматриваются классические методы проектирования, ориентированные на процедурную реализацию программных систем (ПС). Повторим, что эти методы появились в период революции структурного программирования. Учитывая, что на современном этапе программной инженерии процедурно-ориентированные ПС имеют преимущественно историческое значение, *конспективно* обсуждаются только два (наиболее популярных) метода: метод структурного проектирования и метод проектирования Майкла Джексона (этот Джексон не имеет никакого отношения к известному певцу). Зачем мы это делаем? Да чтобы знать исторические корни современных методов проектирования.

Метод структурного проектирования

Этот метод поддерживает проектирование, ориентированное на потоки данных. Назначение метода — обеспечить систематический подход к созданию программной структуры, фундаменту архитектурного проектирования. В этом методе информация представляется как непрерывный поток, который подвергается серии преобразований по мере продвижения от входа к выходу. В качестве графического средства для отображения информационного потока используется диаграмма потока данных. Область применения — последовательная обработка не иерархических структур данных, которая характерна для управляющих приложений, методов численного анализа, управления процессами. Исходными данными для метода структурного проектирования являются компоненты модели анализа ПС, которая представляется иерархией диаграмм потоков данных [54, 76, 82, 99, 104]. Результат структурного проектирования — иерархическая структура ПС.

Шаги метода:

1) определение типа информационного потока;
2) выделение границ потока;
3) отображение диаграммы потока данных в начальную структуру системы;

4) определение иерархии управления (разложением на элементы);

5) уточнение полученной структуры (используются проектные эвристики, то есть продиктованные опытом рекомендации, ориентированные на повышение качества результата).

Действия на шаге 3 зависят от типа информационного потока в модели анализа.

Типы информационных потоков

Различают два типа информационных потоков:

- поток преобразований;
- поток запросов.

Информация должна войти в ИС и выйти из нее в формате «внешнего мира». Примеры форматов информации внешнего мира: данные с клавиатуры, наборы для телефонной линии, рисунки для компьютерного графического дисплея. Для организации обработки входные данные из внешнего формата должны быть преобразованы во внутренний формат. Далее результаты обработки подвергаются обратному преобразованию, то есть возвращаются в формат «внешнего мира». Этим трем этапам соответствуют три элемента в *потоке преобразований*: Входящий поток, Преобразуемый поток и Выходящий поток (рис. 7.1).

Рис. 7.1. Элементы потока преобразований

Информационные потоки могут также иметь особые элементы данных — запросы. Назначение элемента-запроса состоит в том, чтобы запустить поток данных по одному из нескольких путей. Потоки с элементами-запросами называют *потоками запросов*.

Анализ запроса и переключение потока данных на один из путей действий происходит в центре запросов. Центр запросов принимает входящий поток, выделяет

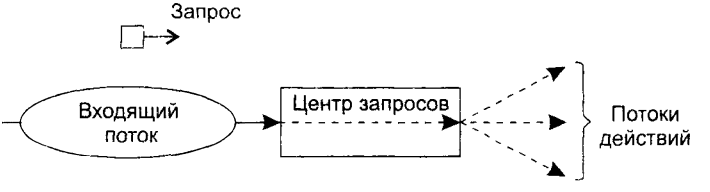


Рис. 7.2. Структура потока запроса

в нем запрос, проводит оценку запроса и по итогам оценки отправляет поток по одному из путей. Иными словами, центр запросов играет роль стрелочника.

Структуру потока запроса иллюстрирует рис. 7.2.

Проектирование для потока данных типа «преобразование»

Шаг 1. Проверка основной системной модели. Модель включает: контекстную диаграмму ПДД0, словарь данных и спецификации процессов. Оценивается их согласованность с системной спецификацией.

Шаг 2. Проверки и уточнения диаграмм потоков данных уровней 1 и 2. Оценивается согласованность диаграмм, достаточность детализации преобразователей.

Шаг 3. Определение типа основного потока диаграммы потоков данных. Основной признак потока преобразований — отсутствие переключения по путям действий.

Шаг 4. Определение границ входящего и выходящего потока, отделение центра преобразований. Входящий поток — отрезок, на котором информация преобразуется из внешнего во внутренний формат представления. Выходящий поток обеспечивает обратное преобразование — из внутреннего формата во внешний. Границы входящего и выходящего потоков достаточно условны. Вариация одного преобразователя на границе слабо влияет на конечную структуру ПС.

Шаг 5. Определение начальной структуры ПС. Иерархическая структура ПС формируется нисходящим распространением управления. В иерархической структуре:

- модули верхнего уровня принимают решения;
- модули нижнего уровня выполняют работу по вводу, обработке и выводу;
- модули среднего уровня реализуют как функции управления, так и функции обработки.

Начальная структура ПС (для потока преобразования) стандартна и включает *главный контроллер* (находится на вершине структуры) и три подчиненных контроллера:

1. *Контроллер входящего потока* (контролирует получение входных данных)
2. *Контроллер преобразуемого потока* (управляет операциями над данными во внутреннем формате).
3. *Контроллер выходящего потока* (управляет получением выходных данных).

Данный минимальный набор модулей покрывает все функции управления, обеспечивает хорошую связность и слабое сцепление структуры.

Начальная структура ПС представлена на рис. 7.3.

Шаг 6. Детализация структуры ПС. Выполняется отображение преобразователей ПДД в модули структуры ПС. Отображение выполняется движением по ПДД от границ центра преобразования вдоль входящего и выходящего потоков. Входящий поток проходится от конца к началу, а выходящий поток — от начала к концу. В ходе движения преобразователи отображаются в модули подчиненных уровней структуры (рис. 7.4).

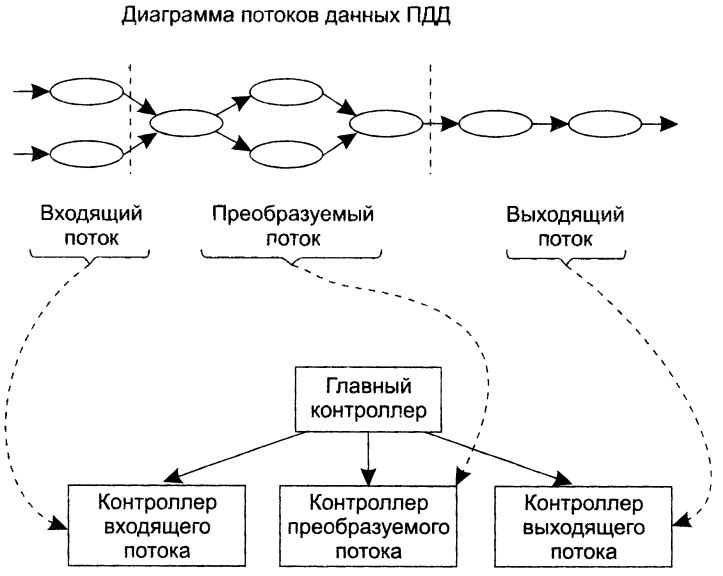


Рис. 7.3. Начальная структура ПС для потока «преобразование»

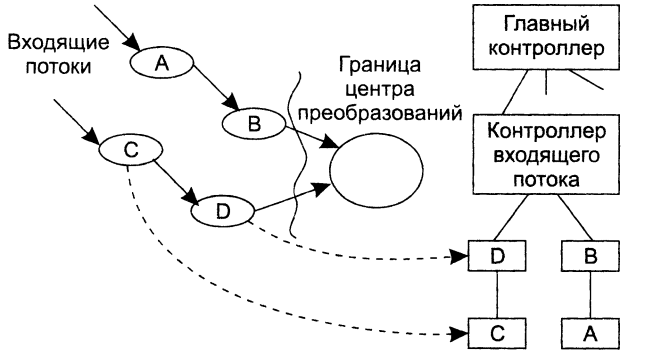


Рис. 7.4. Отображение преобразователей ПДД в модули структуры

Центр преобразования ПДД отображается иначе (рис. 7.5). Каждый преобразователь отображается в модуль, непосредственно подчиненный контроллеру центра.

Преобразуемый поток проходится слева направо.

Возможны следующие варианты отображения:

- 1 преобразователь отображается в 1 модуль;
- 2–3 преобразователя отображаются в 1 модуль;
- 1 преобразователь отображается в 2–3 модуля.

Для каждого модуля полученной структуры на базе спецификаций процессов модели анализа пишется сокращенное описание обработки. Описание включает: входную и выходную информацию модуля (интерфейсное описание); информацию,

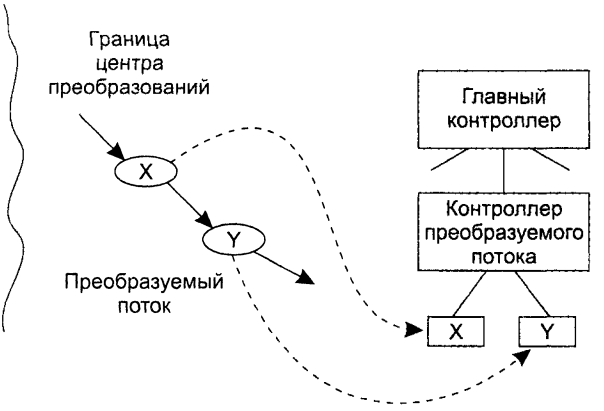


Рис. 7.5. Отображение центра преобразования ПДД

которая сохраняется модулем (данные, сохраняемые в локальной структуре данных); процедурное описание, которое указывает основные особенности решения и задачи; короткое обсуждение ограничений и специальных свойств (файлы ввода-вывода, аппаратно-зависимые характеристики, специальные требования к параметрам времени).

Шаг 7. Уточнение иерархической структуры ПС. Модули разделяются и объединяются для:

1) повышения связности и уменьшения сцепления;
2) упрощения реализации;
3) упрощения тестирования;
4) повышения удобства сопровождения.

Целью семи шагов является создание структурной организации системы, которую можно оценивать и уточнять. Модификации в это время требуют малой дополнительной работы, но оказывают глубокое воздействие на качество и сопровождаемость программного продукта.

Проектирование для потока данных типа «запрос»

Во многих приложениях некоторые элементы данных (запросы) выполняют переключающую функцию, то есть функцию, управляющую переключением информационных потоков.

Шаг 1. Проверка основной системной модели. Модель включает: контекстную диаграмму ПДД0, словарь данных и спецификации процессов. Оценивается их согласованность с системной спецификацией.

Шаг 2. Проверки и уточнения диаграмм потоков данных уровней 1 и 2. Оценивается согласованность диаграмм, достаточность детализации преобразователей.

Шаг 3. Определение типа основного потока диаграммы потоков данных. Основной признак потоков запросов — явное переключение данных на один из путей действий.

Шаг 4. Определение центра запросов и типа для каждого из потоков действия. Если конкретный поток действия имеет тип «преобразование», то для него указываются границы входящего, преобразуемого и выходящего потоков.

Шаг 5. Определение начальной структуры ПС. В начальную структуру отображается та часть диаграммы потоков данных, в которой распространяется поток запросов. Начальная структура ПС для потока запросов стандартна и включает входящую ветвь и диспетчерскую ветвь.

Структура входящей ветви формируется так же, как и в предыдущей методике.

Диспетчерская ветвь включает диспетчер, находящийся на вершине ветви, и контроллеры потоков действия, подчиненные диспетчеру; их должно быть столько, сколько имеется потоков действий.

Шаг 6. Детализация структуры ПС. Производится отображение в структуру каждого потока действия. Каждый поток действия имеет свой тип. Могут встретиться поток-«преобразование» (отображается по предыдущей методике) и поток

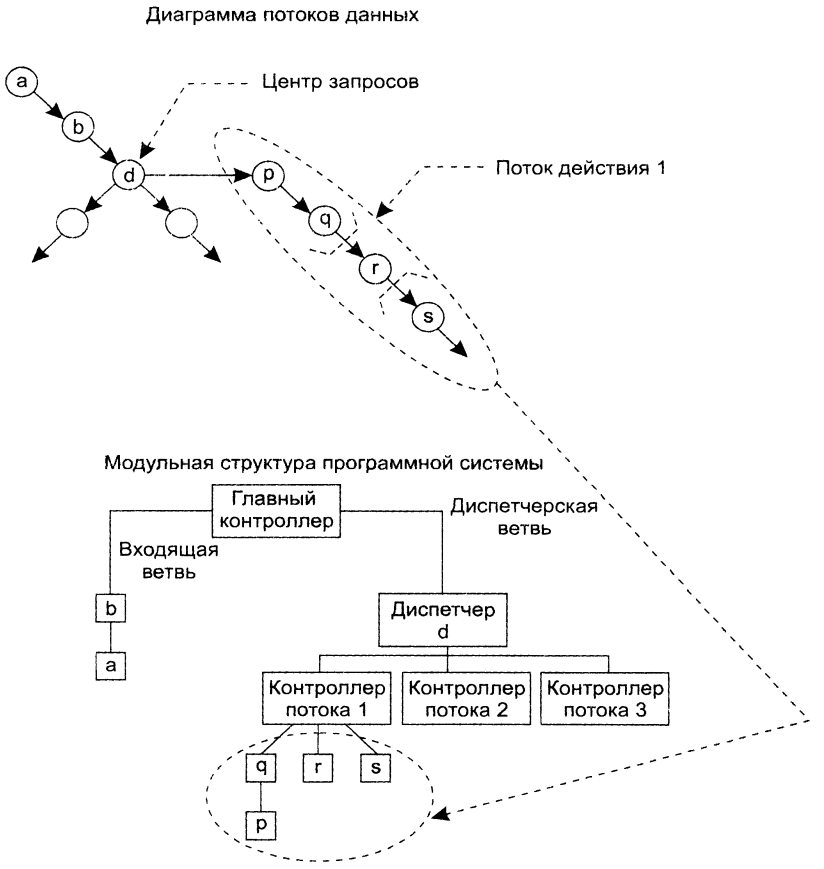


Рис. 7.6. Отображение в модульную структуру ПС потока действия 1

запросов. На рис. 7.6 приведен пример отображения потока действия 1. Подразумевается, что он является потоком преобразования.

Шаг 7. Уточнение иерархической структуры ПС. Уточнение выполняется для повышения качества системы. Как и при предыдущей методике, критериями уточнения служат: независимость модулей, эффективность реализации и тестирования, улучшение сопровождаемости.

Метод проектирования Джексона

Для иллюстрации проектирования по этому методу продолжим пример с системой обслуживания перевозок, начатый в главе 5.

Метод Джексона включает шесть шагов [61]. Три первых шага относятся к этапу анализа. Это шаги: *объект — действие*, *объект — структура*, *начальное моделирование*. Их мы уже рассмотрели.

Доопределение функций

Следующий шаг — доопределение функций. Этот шаг развивает диаграмму системной спецификации этапа анализа. Уточняются процессы-модели. В них вводятся дополнительные функции. Джексон выделяет три типа сервисных функций:

1. Встроенные функции (задаются командами, вставляемыми в структурный текст процесса-модели).
2. Функции впечатления (наблюдают вектор состояния процесса-модели и вырабатывают выходные результаты).
3. Функции диалога. Они решают следующие задачи:
   - наблюдают вектор состояния процесса-модели;
   - формируют и выводят поток данных, влияющий на действия в процессе-модели;
   - выполняют операции для выработки некоторых результатов.

Встроенную функцию введем в модель ТРАНСПОРТ-1. Предположим, что в модели есть панель с лампочкой, сигнализирующей о прибытии. Лампочка включается командой LON(i), а выключается командой LOFF(i). По мере перемещения транспорта между остановками формируется поток LAMP-команд. Модифицированный структурный текст модели ТРАНСПОРТ-1 принимает вид:

```
ТРАНСПОРТ-1
    LON(1);
    опрос SV;
    ЖДАТЬ цикл ПОКА ПРИБЫЛ(1)
        опрос SV;
    конец ЖДАТЬ;
    LOFF(1);
    покинуть(1);
    ТРАНЗИТ цикл ПОКА УБЫЛ(1)
        опрос SV;
    конец ТРАНЗИТ;
    ТРАНСПОРТ цикл
        ОСТАНОВКА;
```

```
            прибыть(i);
            LON(i);
            ЖДАТЬ цикл ПОКА ПРИБЫЛ(i)
                опрос SV;
            конец ЖДАТЬ;
            LOFF(i);
            покинуть(i);
            ТРАНЗИТ цикл ПОКА УБЫЛ(i)
                опрос SV;
            конец ТРАНЗИТ;
        конец ОСТАНОВКА;
    конец ТРАНСПОРТ;
    прибыть(1);
конец ТРАНСПОРТ-1;
```

Теперь введем функцию впечатления. В нашем примере она может формировать команды для мотора транспорта: START, STOP.

Условия выработки этих команд:

- Команда STOP формируется, когда датчики регистрируют прибытие транспорта на остановку.
- Команда START формируется, когда нажата кнопка для запроса транспорта и транспорт ждет на одной из остановок.

Видим, что для выработки команды STOP необходима информация только от модели транспорта. В свою очередь, для выработки команды START нужна информация и от модели КНОПКА-1, так и от модели ТРАНСПОРТ-1. В силу этого для реализации функции впечатления введем функциональный процесс М-УПРАВЛЕНИЕ. Он будет обрабатывать внешние данные и формировать команды START и STOP.

Ясно, что процесс М-УПРАВЛЕНИЕ должен иметь внешние связи с моделями ТРАНСПОРТ-1 и КНОПКА. Соединение с моделью КНОПКА организуем через вектор состояния BV. Соединение с моделью ТРАНСПОРТ-1 организуем через поток данных S1D.

Для обеспечения М-УПРАВЛЕНИЯ необходимой информацией опять надо изменить структурный текст модели ТРАНСПОРТ-1. В нем предусмотрим занесение сообщения Прибыл в буфер S1D:

```
ТРАНСПОРТ-1
    LON(1);
    опрос SV;
    ЖДАТЬ цикл ПОКА ПРИБЫЛ(1)
        опрос SV;
    конец ЖДАТЬ;
    LOFF(1);
    покинуть(1);
    ТРАНЗИТ цикл ПОКА УБЫЛ(1)
        опрос SV;
    конец ТРАНЗИТ;
    ТРАНСПОРТ цикл
        ОСТАНОВКА;
            прибыть(i);
            записать Прибыл в S1D;
            LON(i);
            ЖДАТЬ цикл ПОКА ПРИБЫЛ(i)
```

```
            опрос SV;
        конец ЖДАТЬ;
        LOFF(i);
        покинуть(i);
        ТРАНЗИТ цикл ПОКА УБЫЛ(i)
            опрос SV;
        конец ТРАНЗИТ;
    конец ОСТАНОВКА;
конец ТРАНСПОРТ;
прибыть(1);
записать Прибыл в S1D;
конец ТРАНСПОРТ-1;
```

Очевидно, что при такой связи процессов необходимо гарантировать, что процесс ТРАНСПОРТ-1 выполняет операции опрос SV, а процесс М-УПРАВЛЕНИЕ читает сообщения Прибытия в S1D с частотой, достаточной для своевременной остановки транспорта. Временные ограничения, планирование и реализация должны рассматриваться в последующих шагах проектирования.

В заключение введем функцию диалога. Свяжем эту функцию с необходимостью развития модели КНОПКА-1. Следует различать первое нажатие на кнопку (оно формирует запрос на поездку) и последующие нажатия на кнопку (до того, как поездка действительно началась).

Диаграмма дополнительного процесса КНОПКА-2, в котором учтено это уточнение, показана на рис. 7.7.

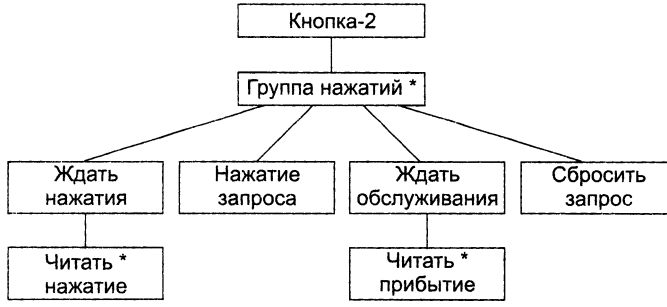


Рис. 7.7. Диаграмма дополнительного процесса КНОПКА-2

Внешние связи модели КНОПКА-2 должны включать:

- одно соединение с моделью КНОПКА-1 — организуется через поток данных B1D (для приема сообщения о нажатии кнопки);
- два соединения с процессом М-УПРАВЛЕНИЕ — одно организуется через поток данных MBD (для приема сообщения о прибытии транспорта), другое организуется через вектор состояния BV (для передачи состояния переключателя Запрос).

Таким образом, КНОПКА-2 читает два буфера данных, заполняемых процессами КНОПКА-1 и М-УПРАВЛЕНИЕ, и формирует состояние внутреннего электронного переключателя Запрос. Она реализует функцию диалога.

Структурный текст модели КНОПКА-2 может иметь следующий вид:

```
КНОПКА-2
   Запрос := НЕТ;
   читать B1D;
   ГрНАЖ цикл
       ЖдатьНАЖ цикл ПОКА Не НАЖАТА
           читать B1D;
       конец ЖдатьНАЖ;
       Запрос := ДА;
       читать MBD;
       ЖдатьОБСЛУЖ цикл ПОКА Не ПРИБЫЛ
           читать MBD;
       конец ЖдатьОБСЛУЖ;
       Запрос := НЕТ;
       читать B1D;
   конец ГрНАЖ;
конец КНОПКА-2;
```

Диаграмма системной спецификации, отражающая все изменения, представлена рис. 7.8.

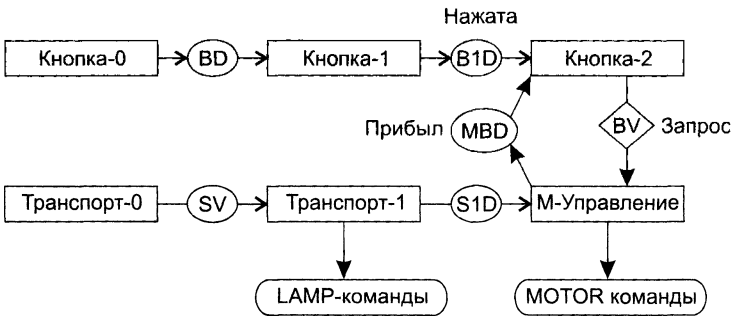


Рис. 7.8. Полная диаграмма системной спецификации

Встроенная в ТРАНСПОРТ-1 функция вырабатывает LAMP-команды, функция впечатления модели М-УПРАВЛЕНИЕ генерирует команды управления мотором, а модель КНОПКА-2 реализует функцию диалога (совместно с процессом М-УПРАВЛЕНИЕ).

Учет системного времени

На шаге учета системного времени проектировщик определяет временные ограничения, накладываемые на систему, фиксирует дисциплину планирования. Дело в том, что на предыдущих шагах проектирования была получена система, составленная из последовательных процессов. Эти процессы связывали только потоки данных, передаваемые через буфер, и взаимные наблюдения векторов состояния. Теперь необходимо произвести дополнительную синхронизацию процессов во времени, учесть влияние внешней программно-аппаратной среды и совместно используемых системных ресурсов.

Временные ограничения для системы обслуживания перевозок, в частности, включают:

1) временной интервал на выработку команды STOP. Он должен выбираться путем анализа скорости транспорта и ограничения мощности;
2) время реакции на включение и выключение ламп панели.

Для рассмотренного примера нет необходимости вводить специальный механизм синхронизации. Однако при расширении может потребоваться некоторая синхронизация обмена данными.

Контрольные вопросы и упражнения

1. В чем состоит суть метода структурного проектирования?
2. Какие различают типы информационных потоков?
3. Что такое входящий поток?
4. Что такое выходящий поток?
5. Что такое центр преобразования?
6. Как производится отображение входящего потока?
7. Как производится отображение выходящего потока?
8. Как производится отображение центра преобразования?
9. Какие задачи решают главный контроллер, контроллер входящего потока, контроллер выходящего потока и контроллер центра преобразования?
10. Поясните шаги метода структурного проектирования.
11. Что такое входящая ветвь?
12. Что такое диспетчерская ветвь?
13. Какие существуют различия в методике отображения потока преобразований и потока запросов?
14. Какие задачи уточнения иерархической структуры программной системы вы знаете?
15. С помощью аппарата структурного проектирования создайте проектную модель системы управления летательного аппарата. В качестве исходных данных используйте модель анализа, созданную в упражнении 10 главы 5.
16. Какие шаги предусматривает метод Джексона на этапе проектирования?
17. В чем состоит суть развития диаграммы системной спецификации Джексона?
18. Поясните понятие встроенной функции.
19. Поясните понятие функции впечатления.
20. Поясните понятие функции диалога.
21. В чем состоит учет системного времени (в методе Джексона)?
22. Сравните метод Джексона с методом структурного проектирования. Выделите их достоинства и недостатки.
23. На основе модели анализа, полученной в упражнении 35 главы 5, создайте по методу Джексона проектную модель для университета, размещаемого на трех территориях.

Глава 8

Основы объектно-ориентированного представления программных систем

Данная глава вводит в круг вопросов объектно-ориентированного представления программных систем (ПС). В этой главе рассматриваются: абстрагирование понятий проблемной области, приводящее к формированию классов; инкапсуляция объектов, обеспечивающая скрытость их характеристик; модульность как средство упаковки набора классов; особенности построения иерархической структуры объектно-ориентированных систем. Последовательно обсуждаются объекты и классы как основные строительные элементы объектно-ориентированного ПО. Значительное внимание уделяется описанию отношений между объектами и классами. Кроме того, здесь определяются базовые понятия унифицированного языка моделирования.

Принципы объектно-ориентированного представления программных систем

Рассмотрение любой сложной системы требует применения техники декомпозиции — разбиения на составляющие элементы. Известны две схемы декомпозиции: алгоритмическая декомпозиция и объектно-ориентированная декомпозиция.

В основе алгоритмической декомпозиции лежит разбиение по действиям — алгоритмам. Эта схема представления применяется в обычных (процедурных) ПС.

Объектно-ориентированная декомпозиция обеспечивает разбиение по автономным лицам — объектам реального (или виртуального) мира. Эти лица (объекты) — более «крупные» элементы, каждый из них несет в себе и описания действий, и описания данных.

Объектно-ориентированное представление ПС основывается на принципах абстрагирования, инкапсуляции, модульности и иерархической организации. Каждый

из этих принципов не нов, но их совместное применение рассчитано на проведение объектно-ориентированной декомпозиции. Это определяет модификацию их содержания и механизмов взаимодействия друг с другом. Обсудим данные принципы [38, 52, 63, 83, 88, 90].

Абстрагирование

Аппарат абстракции — удобный инструмент для борьбы со сложностью реальных систем. Создавая понятие в интересах какой-либо задачи, мы отвлекаемся (абстрагируемся) от несущественных характеристик конкретных объектов, определяя только существенные характеристики. Например, в абстракции «часы» мы выделяем характеристику «показывать время», отвлекаясь от таких характеристик конкретных часов, как форма, цвет, материал, цена, изготовитель.

Итак, абстрагирование сводится к формированию абстракций. Каждая абстракция фиксирует основные характеристики объекта, которые отличают его от других видов объектов и обеспечивают ясные понятийные границы.

Абстракция концентрирует внимание на внешнем представлении объекта, позволяет отделить основное в поведении объекта от его реализации. Абстракцию удобно строить путем выделения обязанностей объекта.

Пример 8.1. Физический объект — один из трех датчиков скорости, устанавливаемый на борту летательного аппарата (ЛА) по одному из трех направлений. Создадим его абстракцию. Для этого сформулируем обязанности датчика:

- знать проекцию скорости ЛА в заданном направлении;
- показывать текущую скорость;
- подвергаться настройке.

Анализируя обязанности, делаем следующие выводы: абстракция должна содержать два действия (показывать текущую скорость, настраивать) и два поля данных (скорость, направление). Описание абстракции датчика запишем как класс на языке Java:

```
public class ДатчикСкорости {
    // поля данных
    private Скорость скорость;
    private Направление направление;
    // конструктор, задает направление датчика
    ДатчикСкорости(Направление направление) {
        this.направление = направление;
    }
    // отображение текущей скорости
    public Скорость текущаяСкорость() {
        return скорость;
    }
    // настройка датчика
    public void настраивать(Скорость действитСкорость) {
        this.скорость = действитСкорость;
    }
}
```

ПРИМЕЧАНИЕ

Напомним, запись private Скорость скорость означает поле данных скорость с типом Скорость, причем это поле скрыто от клиента.

Подробности выполнения обязательств абстракцией ДатчикСкорости зависят от ...е внутреннего представления и не должны интересовать внешних клиентов. Эта ...нформация является секретом класса и реализуется его закрытой (приватной) ...астью совместно с содержанием действий-методов.

Класс ДатчикСкорости еще не объект. Собственно датчики — это его экземпляры, ...и их нужно создать, прежде чем с ними можно будет работать. Например, можно ...аписать так:

```
ДатчикСкорости датчикПродольнойСкорости = new ДатчикСкорости();
ДатчикСкорости датчикПоперечнойСкорости = new ДатчикСкорости();
ДатчикСкорости датчикНормальнойСкорости = new ДатчикСкорости();
```

Итак, абстракция отражает суть объекта, опуская второстепенные детали. Вы...ор и правильное описание набора абстракций для заданной предметной области ...вляется главной задачей объектно-ориентированной разработки.

Инкапсуляция

Инкапсуляция и абстракция — взаимодополняющие понятия: абстракция выде...яет внешнее поведение объекта, а инкапсуляция содержит и скрывает реализа...цию, которая обеспечивает это поведение. Инкапсуляция достигается с помощью ...нформационной закрытости. Обычно скрывается структура данных объектов ... реализация их методов.

Инкапсуляция является процессом разделения элементов абстракции на сек...ии с различной видимостью. Инкапсуляция служит для отделения интерфейса ...бстракции от ее реализации.

Пример 8.2. Продолжим заниматься предметной областью системы управле...ия летательного аппарата. Применим принцип разделения понятий (separation of ...oncerns): разобравшись с одной абстракцией, переходим к обсуждению физического ...бъекта «Регулятор скорости». Как и в прошлый раз, определим его обязанности.

Обязанности регулятора:

- знать свой номер, а также порт, через который он получает команды управления;
- включаться;
- выключаться;
- увеличивать скорость по своему каналу;
- уменьшать скорость по своему каналу;
- отображать свое состояние.

Исходя из обязанностей, описание класса Регулятор скорости примет вид:

```
...blic class РегуляторСкорости {
    enum Режим {
        Увеличение, Уменьшение
    }
```

```
    private Размещение номер;
    private Режим состояние;
    private Порт управление;

    public РегуляторСкорости(Размещение номер, Режим состояние, Порт управление) {
        this.номер = номер;
        this.управление = управление;
        ...
    }
    public void включить() {
        ...
    }
    public void выключить() {
        ...
    }
    public void увеличитьСкорость() {
        ...
    }
    public void уменьшитьСкорость() {
        ...
    }
    public Режим опросСостояния() {
        return состояние;
    }
}
```

Три поля номер, состояние, управление формулируют инкапсулируемое представление данных класса РегуляторСкорости. При попытке клиента получить к этим полям доступ фиксируется семантическая ошибка.

Полное инкапсулированное представление класса РегуляторСкорости включает описание реализаций его методов, которое для краткости здесь опущено.

Модульность

В языках C++, Delphi Pascal, Ada 2005, Java и C# абстракции классов и объектов формируют логическую структуру системы. При производстве физической структуры эти абстракции помещаются в модули. В больших системах, где сотни классов, модули помогают управлять сложностью. Модули служат физическими контейнерами, в которых объявляются классы и объекты логической разработки.

Модульность определяет способность системы подвергаться декомпозиции на ряд сильно связанных и слабо сцепленных модулей.

Общая цель декомпозиции на модули: уменьшение сроков разработки и стоимости ПС за счет выделения модулей, которые проектируются и изменяются независимо. Каждая модульная структура должна быть достаточно простой, чтобы быть полностью понятой. Изменение реализации модулей должно проводиться без знания реализации других модулей и без влияния на их поведение. Правильно выбрать модули почти так же сложно, как выбрать правильный набор абстракций.

Определение классов и объектов выполняется в ходе логической разработки, а определение модулей — в ходе физической разработки системы. Эти действия сильно взаимосвязаны, осуществляются итеративно.

В некоторых языках программирования (например, в Smalltalk) модулей нет и единственной физической единицей декомпозиции считается класс. Во многих

...их языках, включая Delphi Pascal, C++ и Ada 2005, модуль является самостоятельной языковой конструкцией, обеспечивающей создание отдельных проектных ...ений. В языке Java мощным средством обеспечения модульности являются ...тренние классы и пакеты.

Пример 8.3. Пусть имеется несколько классов управления полетом летательного ...рата (ЛА) – класс угловой стабилизации ЛА и класс управления движением ...ра масс ЛА. Нужно создать модуль, чье назначение — собрать все эти классы. ...можны два способа.

Встраивание классов управления непосредственно в класс-контейнер:

```
public class УпрПолетом {
    private class УгловаяСтабилизация {
        ...
    }

    private class ДвиженЦентраМасс {
        ...
    }
}
```

Размещение в пакете:

```
package упрполетом;

    class УгловаяСтабилизация {
        ...
    }

    class ДвиженЦентраМасс {
        ...
    }
```

Иерархическая организация

...рассмотрели три механизма для борьбы со сложностью:

- ...бстракцию (она упрощает представление физического объекта);
- ...инкапсуляцию (закрывает детали внутреннего представления абстракций);
- ...модульность (...ает путь группировки логически связанных абстракций).

...Прекрасным дополнением к этим механизмам является иерархическая организ... – формирование из абстракций иерархической структуры. Определением ...рхии в проекте упрощается понимание проблем заказчика и их реализация — ...ая система становится обозримой человеком.

...Иерархическая организация задает размещение абстракций на различных уров...писания системы.

...Двумя важными инструментами иерархической организации в объектно-...ированных системах являются:

- ...труктура из классов («*is a*»-иерархия);
- ...труктура из объектов («*part of*»-иерархия).

...Чаще всего «*is a*»-иерархическая структура строится с помощью наследова... Наследование определяет отношение между классами, где класс разделяет

структуру или поведение, определенные в одном другом (единичное наследование) или в нескольких других (множественное наследование) классах.

Пример 8.4. Положим, что программа управления полетом 2-й ступени ракеты-носителя в основном похожа на программу управления полетом 1-й ступени, но все же отличается от нее. Определим класс управления полетом 2-й ступени, который инкапсулирует ее специализированное поведение:

```
public class УпрПолетом2 extends УпрПолетом1 {
    enum Траектория {
        Гибкая, Свободная
    }

    private Траектория типТраектории;
    private БортовоеВремя доОтделения;
    private Boolean выполнениеКоманд;

    УпрПолетом2(Траектория тип, Углы ориентация, Координаты_Скорость параметры,
                График команды) {
        ...
    }

    public void условияОтделения3ступени(КритерийОтделения критерий) {
        ...
    }

    public БортовоеВремя прогнозОтделения3ступени() {
        ...
    }

    public boolean исполнениеКоманд() {
        ...
    }
}
```

Видим, что класс УпрПолетом2 — это «*is a*»-разновидность класса УпрПолетом1, который называется родительским классом или суперклассом.

В класс УпрПолетом2 добавлены:

- вспомогательный тип Траектория;
- три новых поля данных (типТраектории, доОтделения, выполнениеКоманд);
- три новых метода (условияОтделения3ступени(), прогнозОтделения3ступени(), исполнениеКоманд()).

Кроме того, в классе УпрПолетом2 переопределен конструктор суперкласса. Подразумевается, что в наследство классу УпрПолетом2 достался набор методов и атрибутов класса УпрПолетом1. В частности, класс УпрПолетом2 включает поля класса УпрПолетом1, обеспечивающие прием данных о координатах и скорости ракеты-носителя, ее угловой ориентации и графике выдаваемых команд, а также о критерии отделения следующей ступени.

Классы УпрПолетом1 и УпрПолетом2 образуют наследственную иерархическую организацию. В ней общая часть структуры и поведения сосредоточены в верхнем наиболее общем классе (суперклассе). Суперкласс соответствует общей абстракции, а подкласс — специализированной абстракции, в которой элементы суперкласса

дополняются, изменяются и даже скрываются. Поэтому наследование часто называют отношением *обобщение-специализация*.

Иерархию наследования можно продолжить. Например, используя класс УпрПолетом2, можно объявить еще более специализированный подкласс — УпрПолетомКосмическогоАппарата.

Другая разновидность иерархической организации — «*part of*»-иерархическая структура — базируется на отношении агрегации. Агрегация не является понятием, уникальным для объектно-ориентированных систем. Например, любой язык программирования, разрешающий структуры типа «запись», поддерживает агрегацию. И все же агрегация особенно полезна в сочетании с наследованием:

1) агрегация обеспечивает физическую группировку логически связанной структуры;
2) наследование позволяет легко и многократно использовать эти общие группы в других абстракциях.

Пример 8.5. Представим класс ИзмерительСУ (измеритель системы управления ЛА) на языке C++ (поскольку в языке Java нет указателей):

```
class ИзмерительСУ {
    public: // видимая секция
        // описание методов
        ИзмерительСУ (); // конструктор
        ~ИзмерительСУ (); // деструктор
    private: // скрытная секция данных
        Датчик *датчики [30]; // массив указателей на датчики
        НастройкаДатчиков процедураНастройки;
```

Очевидно, что объекты класса ИзмерительСУ являются агрегатами, состоящими из массива датчиков и процедуры настройки. Наша абстракция ИзмерительСУ позволяет использовать в системе управления различные датчики. Изменение датчика не меняет индивидуальности измерителя в целом. Ведь датчики вводятся в агрегат с помощью указателей, а не величин. Таким образом, объект класса ИзмерительСУ и объекты класса Датчик имеют относительную независимость. Например, время жизни измерителя и время жизни датчиков независимы друг от друга. Напротив, объект класса НастройкаДатчиков физически включается в объект класса ИзмерительСУ и независимо существовать не может. Отсюда вывод — разрушая объект класса ИзмерительСУ мы, в свою очередь, разрушаем экземпляр НастройкиДатчиков.

Интересно сравнить элементы иерархий наследования и агрегации с точки зрения уровня сложности. При наследовании нижний элемент иерархии (подкласс) имеет больший уровень сложности (большие возможности), при агрегации — наоборот, агрегат ИзмерительСУ обладает большими возможностями, чем его элементы — датчики и процедура настройки).

Объекты

Рассмотрим более пристально объекты — конкретные сущности, которые существуют во времени и пространстве. В понятие программного объекта вкладывалась простая идея. Программные объекты должны моделировать физические объекты

реального мира, поэтому они должны вобрать в себя черты и характеристики физических объектов.

Общая характеристика объектов

Объект — это конкретное представление абстракции. Объект обладает индивидуальностью, состоянием и поведением. Структура и поведение подобных объектов определены в их общем классе. Термины «экземпляр класса» и «объект» взаимозаменяемы. На рис. 8.1 приведен пример объекта по имени стул, имеющего определенный набор атрибутов и операций.

Индивидуальность — это характеристика объекта, которая отличает его от всех других объектов.

Состояние объекта характеризуется перечнем всех атрибутов объекта и текущими значениями каждого из этих атрибутов (рис. 8.1).

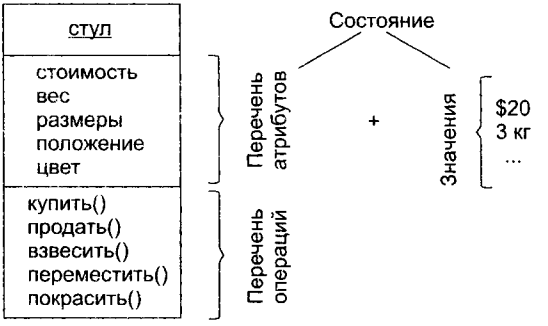


Рис. 8.1. Представление объекта с именем стул

Объекты не существуют изолированно друг от друга. Они подвергаются воздействию или сами воздействуют на другие объекты.

Поведение характеризует то, как объект воздействует на другие объекты (или подвергается воздействию) в терминах изменений его состояния и передачи сообщений. Поведение объекта — это его деятельность с точки зрения окружающей среды. Поведение объекта является функцией как его состояния, так и выполняемых им операций (купить(), продать(), взвесить(), переместить(), покрасить()). Говорят, что состояние объекта представляет суммарный результат его поведения.

Операция обозначает обслуживание, которое объект предлагает своим клиентам. Возможны пять видов операций клиента над объектом:

1) модификатор (изменяет состояние объекта);
2) селектор (дает доступ к состоянию, но не изменяет его);
3) итератор (дает доступ к содержанию объекта по частям, в строго определенном порядке);
4) конструктор (создает объект и инициализирует его состояние);
5) деструктор (разрушает объект и освобождает занимаемую им память).

Примеры операций приведены в табл. 8.1.

В чистых объектно-ориентированных языках программирования операции могут объявляться только как методы — элементы классов, экземплярами которых являются объекты. Гибридные языки (C++, Ада 2005) позволяют писать операции как свободные подпрограммы (вне классов). Соответствующие примеры показаны на рис. 8.2.

Таблица 8.1. Разновидности операций

Вид операции	Пример операции
Модификатор	пополнеть (кг)
Селектор	какойВес () : integer
Итератор	показатьАссортиментТоваров () : string
Конструктор	создатьРобот (параметры)
Деструктор	уничтожитьРобот ()

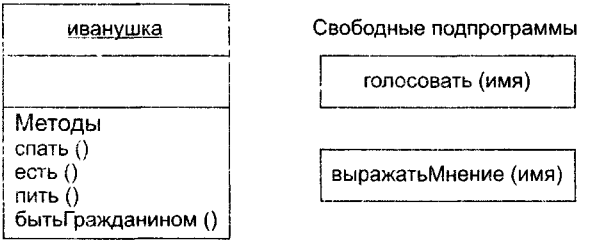


Рис. 8.2. Методы и свободные подпрограммы

В общем случае все методы и свободные подпрограммы, ассоциированные с конкретным объектом, образуют его *протокол*. Таким образом, протокол определяет сферу допустимого поведения объекта и поэтому заключает в себе целостное (статическое и динамическое) представление объекта.

Большой протокол полезно разделять на логические группировки поведения. Эти группировки, разделяющие пространство поведения объекта, обозначают *роли*, которые может играть объект. Принцип выделения ролей иллюстрирует рис. 8.3.

С точки зрения внешней среды важное значение имеет такое понятие, как обязанности объекта. *Обязанности* означают обязательства объекта обеспечить определенное поведение. Обязанностями объекта являются все виды обслуживания, которые он предлагает клиентам. В мире объект играет определенные роли, выполняя свои обязанности.

В заключение отметим: наличие у объекта внутреннего состояния означает, что порядок выполнения им операций очень важен. Иначе говоря, объект может представляться как независимый автомат. По аналогии с автоматами можно выделять активные и пассивные объекты (рис. 8.4).

Активный объект имеет собственный канал (поток) управления, пассивный — нет. Активный объект автономен, он может проявлять свое поведение без воздействия со стороны других объектов. Пассивный объект, наоборот, может изменять свое состояние только под воздействием других объектов.

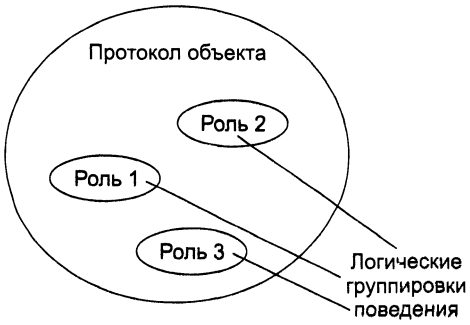


Рис. 8.3. Пространство поведения объекта

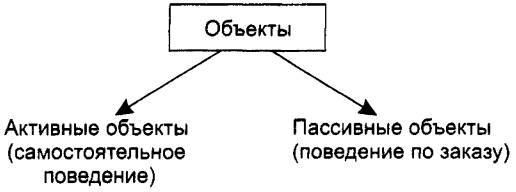


Рис. 8.4. Активные и пассивные объекты

Виды отношений между объектами

В поле зрения разработчика ПО находятся не объекты-одиночки, а взаимодействующие объекты, ведь именно взаимодействие объектов реализует поведение системы. У Г. Буча есть отличная цитата из Галла: «самолет — это набор элементов, каждый из которых по своей природе стремится упасть на землю, но ценой совместных непрерывных усилий преодолевает эту тенденцию» [38]. Отношения между парой объектов основываются на взаимной информации о разрешенных операциях и ожидаемом поведении. Особо интересны два вида отношений между объектами: связи и агрегация.

Связи

Связь — это физическое или понятийное соединение между объектами. Объект сотрудничает с другими объектами через соединяющие их связи. Связь обозначает соединение, с помощью которого:

- объект-клиент вызывает операции объекта-поставщика;
- один объект перемещает данные к другому объекту.

Можно сказать, что связи являются рельсами между станциями-объектами, по которым ездят «трамвайчики сообщений».

Связи между объектами показаны на рис. 8.5 с помощью соединительных линий. Связи представляют возможные пути для передачи сообщений. Сами сообщения показаны стрелками, отмечающими их направления, и помечены именами вызываемых операций. Чаще всего сообщение воспринимается принимающим объектом как заказ на выполнение операции. На нашей диаграмме объект том просит объект

эти выполнить операцию танцевать(), а объект музыкальный центр просит объект колонки исполнить операцию звучать().

ПРИМЕЧАНИЕ

Записывая имена Мэри и Том с маленькой буквы, мы вовсе не выказываем свое неуважение. Просто в программной среде имена объектов, их атрибутов и операций принято начинать со строчной буквы. Это одно из «правил приличия» проектирования и программирования.

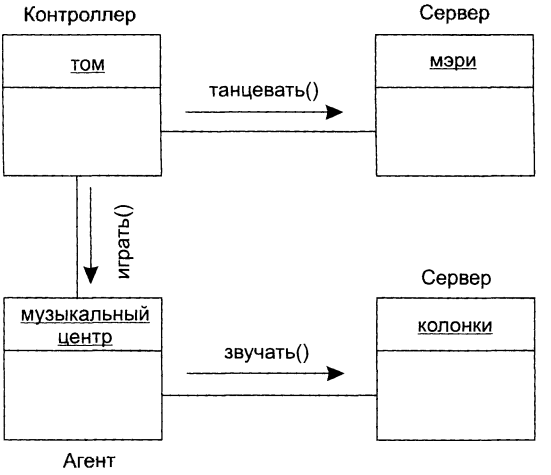


Рис. 8.5. Связи между объектами

Как участник связи объект может играть одну из трех ролей:

- контроллер — объект, который может воздействовать на другие объекты, но никогда не подвержен воздействию других объектов;
- сервер — объект, который никогда не воздействует на другие объекты, он только используется другими объектами;
- агент — объект, который может как воздействовать на другие объекты, так и использоваться ими. Агент создается для выполнения работы от имени контроллера или другого агента.

На рис. 8.5 том — это контроллер, мэри, колонки — серверы, музыкальный центр — агент.

ПРИМЕЧАНИЕ

Полезно помнить о разной организации действий в физическом мире и в объектно-ориентированном программном мире. Поясним это на примере фразы «мама мыла раму». В физическом мире мама будет прикладывать действие к раме, изменяя ее состояние. В программном мире рама должна быть «самомоющейся». Объект мама может только послать сообщение «мыться» в объект раму, приняв которое рама начнет выполнять свою собственную операцию. Иными словами, в объектно-ориентированном подходе какому-либо объекту запрещается выполнять операцию, изменяющую значения атрибутов другого объекта. Обоснуйте, пожалуйста, смысл такого запрета.

Пример 8.6. Допустим, что нужно обеспечить следующий график разворота первой ступени ракеты по углу тангажа, представленный на рис. 8.6.

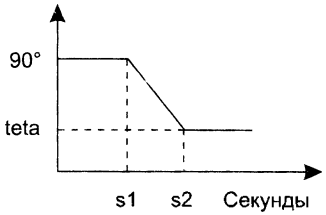


Рис. 8.6. График разворота первой ступени ракеты

Запишем абстракцию графика разворота:

```
import ДатчикУглаТангажа;
    public class ГрафикРазворота {
        public void очистить() {
            ...
        }

        public void связать(Угол teta, Секунда s1, Секунда s2) {
            ...
        }

        public Угол уголНаМомент(Секунда s) {
            ...
        }

        ...
    }
```

Для решения задачи надо обеспечить сотрудничество трех объектов: экземпляра ГрафикРазворота Регулятора Угла и Контроллера Угла.

Описание класса КонтроллерУгла может иметь следующий вид:

```
public class КонтроллерУгла {
    ...
    private РегуляторУгла регулятор =
        new РегуляторУгла(new Размещение(1), new Режим(1), new Порт(10));

    public void обрабатывать(ГрафикРазворота графикРазворота) {
        ...
    }

    public Секунда запланировано(ГрафикРазворота графикРазворота) {
        ...
    }
    ...
}
```

ПРИМЕЧАНИЕ

Операция запланировано() позволяет клиентам запросить у экземпляра Контроллера-Угла время обработки следующего графика.

И наконец, описание класса РегуляторУгла представим в следующей форме:

```
public class РегуляторУгла {
    enum Режим {
        Увеличение, Уменьшение
    }
    private Размещение номер;
    private Режим состояние;
    private Порт управление;

    public РегуляторУгла(Размещение номер, Режим состояние, Порт управление)
    {
        ...
    }

    public void включить() { ... }

    public void выключить() { ... }

    public void увеличитьУгол() { ... }

    public void уменьшитьУгол() { ... }

    public Режим опросСостояния() { return состояние; }

    ...
```

Теперь, когда сделаны необходимые приготовления, объявим нужные экземпляры классов, то есть объекты:

```
ГрафикРазворота рабочийГрафик = new ГрафикРазворота();
КонтроллерУгла рабочийКонтроллер = new КонтроллерУгла();
```

Далее мы должны определить конкретные параметры графика разворота

```
рабочийГрафик.связать(new Угол(30), new Секунда(60), new Секунда(90));
```

Затем предложить объекту-контроллеру выполнить этот график:

```
рабочийКонтроллер.обрабатывать(рабочийГрафик);
```

Рассмотрим отношение между объектом рабочийГрафик и объектом рабочийКонтроллер. Экземпляр рабочийКонтроллер — это агент, отвечающий за выполнение графика разворота и поэтому использующий объект рабочийГрафик как сервер. В данном отношении объект рабочийКонтроллер использует объект рабочийГрафик как аргумент в одной из своих операций.

Видимость объектов

Рассмотрим два объекта A и B, между которыми имеется связь. Для того чтобы объект A мог послать сообщение в объект B, надо чтобы B был виден для A.

В примере из предыдущего подраздела объект рабочийКонтроллер должен видеть объект рабочийГрафик (чтобы иметь возможность использовать его как аргумент операции обрабатывать).

Различают четыре формы видимости между объектами.

- Объект-поставщик (сервер) глобален для клиента.
- Объект-поставщик (сервер) является параметром операции клиента.

3. Объект-поставщик (сервер) является частью объекта-клиента.
4. Объект-поставщик (сервер) является локально объявленным объектом в операции клиента.

На этапе анализа вопросы видимости обычно опускают. На этапах проектирования и реализации вопросы видимости по связям должны обязательно рассматриваться.

Агрегация

Связи обозначают равноправные (клиент-серверные) отношения между объектами. Агрегация обозначает отношения объектов в иерархии «целое/часть». Агрегация обеспечивает возможность перемещения от целого (агрегата) к его частям (атрибутам).

В примере из подраздела «Связи» объект рабочийКонтроллер имеет атрибут регулятор, чьим классом является РегуляторУгла. Поэтому объект рабочийКонтроллер является агрегатом (целым), а экземпляр РегулятораУгла — одной из его частей. Из рабочегоКонтроллера всегда можно попасть в его регулятор. Обратный же переход (из части в целое) обеспечивается не всегда.

Агрегация может обозначать, а может и не обозначать физическое включение части в целое. На рис. 8.7 приведен пример физического включения (композиции) частей (двигателя, сидений, колес) в агрегат автомобиль. В этом случае говорят, что части включены в агрегат по величине.

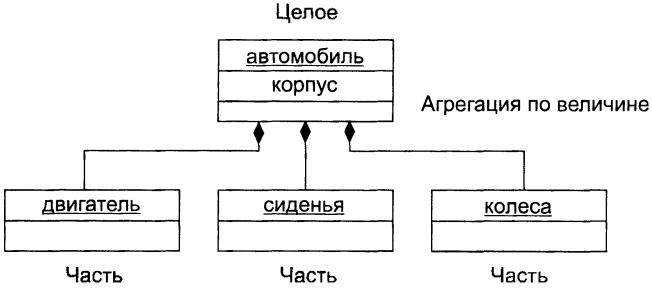


Рис. 8.7. Физическое включение частей в агрегат

На рис. 8.8 приведен пример нефизического включения частей (студента, преподавателя) в агрегат институт. Очевидно, что студент и преподаватель являются элементами института, но они не входят в него физически. В этом случае говорят, что части включены в агрегат по ссылке.

Итак, между объектами существует два вида отношений — связи и агрегация. Какое из них выбрать?

При выборе вида отношения должны учитываться следующие факторы:

- связи обеспечивают низкое сцепление между объектами;
- агрегация инкапсулирует части как секреты целого.

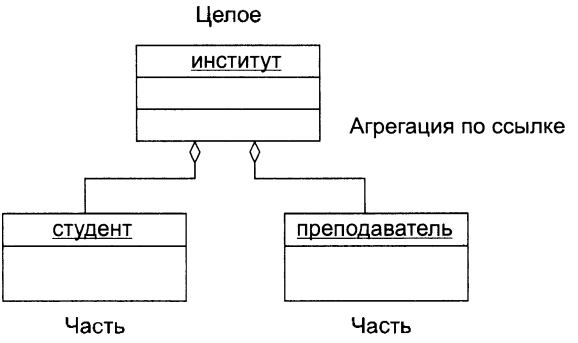


Рис. 8.8. Нефизическое включение частей в агрегат

Классы

Понятия объекта и класса тесно связаны. Тем не менее существует важное различие между этими понятиями. Класс — это абстракция существенных характеристик объекта.

Общая характеристика классов

Класс — описание множества объектов, которые разделяют одинаковые атрибуты, операции, отношения и семантику (смысл). Любой объект — просто экземпляр класса.

Как показано на рис. 8.9, различают внутреннее представление класса (реализацию) и внешнее представление класса (интерфейс).

Интерфейс объявляет возможности (услуги) класса, но скрывает его структуру и поведение. Иными словами, интерфейс демонстрирует внешнему миру абстракцию класса, его внешний облик. Интерфейс в основном состоит из объявлений всех операций, применимых к экземплярам класса. Он может также включать объявления типов (классов), переменных, констант и исключений, необходимых для полноты данной абстракции. В конкретных языках программирования константы и переменные, формирующие структуру данных класса, имеют разные названия. Например, в языке Smalltalk используется термин *переменная экземпляра* (instance variable), в языке Delphi Pascal и Java — *поле* (field), а в языке C++ — *элемент данных* (data-member). Будем использовать эти названия как синонимы.

Рис. 8.9. Структура представления класса

Интерфейс может быть разделен на 4 части:

1) публичную (*public*), объявления которой доступны всем клиентам;
2) защищенную (*protected*), объявления которой доступны только самому классу, его подклассам и друзьям;
3) приватную (*private*), объявления которой доступны только самому классу и его друзьям;
4) пакетную (*package*), объявления которой доступны только классам, входящим в один и тот же пакет.

Другом класса называют класс (функцию), который (которая) имеет доступ ко всем частям этого класса (публичной, защищенной и приватной). Иными словами, от друга у класса нет секретов.

ПРИМЕЧАНИЕ

Уровни видимости интерфейса (реализации) в разных языках программирования имеют свои особенности. Все без исключения уровни поддерживает язык C++. В нем же реализован механизм друга по отношению к функциям и классам. По существу, этот механизм нарушает принцип инкапсуляции, поэтому применять его надо предельно осторожно. В языке Java поддерживаются публичный, защищенный и приватный уровни видимости; здесь нет явного разделения класса на интерфейс и реализацию (оба раздела включены в общее описание класса), но уровень видимости каждого элемента класса задается индивидуально. Друзья в языке Java не предусмотрены, но обеспечивается пакетный доступ (все классы, входящие в один и тот же пакет, имеют доступ друг к другу). В языке Ada 2005 разрешены лишь публичный и приватный уровни видимости. В языке SmallTalk все переменные экземпляров считаются приватными, а все методы остаются публичными.

Реализация класса описывает секреты поведения класса. Она включает реализации всех операций, определенных в интерфейсе класса.

Виды отношений между классами

Классы, подобно объектам, не существуют в изоляции. Напротив, с отдельной предметной областью связывают ключевые абстракции, отношения между которыми формируют структуру из классов системы.

Всего существует четыре основных вида отношений между классами:

- ассоциация (устанавливает семантические соединения, фиксируя структурные отношения – связи между экземплярами классов);
- зависимость (отображает влияние одного класса на другой класс);
- обобщение-специализация («*is a*»-отношение);
- целое-часть («*part of*»-отношение).

Для покрытия основных отношений большинство объектно-ориентированных языков программирования поддерживает следующие отношения:

1) ассоциация;
2) наследование;
3) агрегация;

4) зависимость;

5) конкретизация;

6) метакласс;

7) реализация.

Ассоциации обеспечивают взаимодействия объектов, принадлежащих разным классам. Они являются клеем, соединяющим воедино все элементы программной системы. Благодаря ассоциациям мы получаем работающую систему. Без ассоциаций система превращается в набор изолированных классов-одиночек.

Наследование — наиболее популярная разновидность отношения *обобщение-специализация*. Альтернативой наследованию считается делегирование. При делегировании объекты *делегируют* свое поведение родственным объектам. При этом классы становятся не нужны.

Агрегация обеспечивает отношения *целое-часть*, объявляемые для экземпляров классов.

Зависимость часто представляется в виде частной формы — *использования*, которая фиксирует отношение между клиентом, запрашивающим услугу, и сервером, предоставляющим эту услугу.

Конкретизация выражает другую разновидность отношения *обобщение-специализация*. Применяется в таких языках, как Ada 2005, C++, Java и Eiffel.

Отношения метаклассов поддерживаются в языках SmallTalk и CLOS. Метакласс — это класс классов, понятие, позволяющее обращаться с классами как с объектами.

Реализация определяет отношение, при котором класс-приемник обеспечивает свою собственную реализацию интерфейса другого класса-источника. Иными словами, здесь идет речь о наследовании интерфейса. Семантически реализация — это «скрещивание» отношений зависимости и обобщения-специализации.

Ассоциации классов

Ассоциация обозначает семантическое соединение классов.

Пример 8.7. В системе обслуживания читателей имеются две ключевые абстракции — Книга и Библиотека. Класс Книга играет роль элемента, хранимого в библиотеке. Класс Библиотека играет роль хранилища для книг.

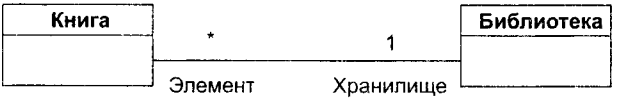


Рис. 8.10. Ассоциация

Отношение ассоциации между классами изображено на рис. 8.10. Очевидно, что ассоциация предполагает двусторонние отношения:

- для данного экземпляра Книги выделяется экземпляр Библиотеки, обеспечивающий ее хранение;
- для данного экземпляра Библиотеки выделяются все хранимые Книги.

Здесь показана ассоциация «один-ко-многим». Каждый экземпляр Книги имеет указатель на экземпляр Библиотеки. Каждый экземпляр Библиотеки имеет набор указателей на несколько экземпляров Книги.

Ассоциация обозначает только семантическую связь. Она не указывает направление и точную реализацию отношения. Ассоциация пригодна для анализа проблемы, когда нам требуется лишь идентифицировать связи. С помощью создания ассоциаций мы приходим к пониманию участников семантических связей, их ролей, мощности (количества элементов).

Ассоциация «один-ко-многим», введенная в примере, означает, что для каждого экземпляра класса Библиотека есть 0 или более экземпляров класса Книга, а для каждого экземпляра класса Книга есть один экземпляр Библиотеки. Эту множественность обозначает *мощность ассоциации*. Мощность ассоциации бывает одного из трех типов:

- «один-к-одному»;
- «один-ко-многим»;
- «многие-ко-многим».

Рис. 8.11. Ассоциации с различными типами мощности

Примеры ассоциаций с различными типами мощности приведены на рис. 8.11 они имеют следующий смысл:

- у европейской жены один муж, а у европейского мужа одна жена;
- у восточной жены один муж, а у восточного мужа сколько угодно жен;
- у заказа один клиент, а у клиента сколько угодно заказов;
- человек может посещать сколько угодно зданий, а в здании может находиться сколько угодно людей.

Наследование

Наследование — это отношение, при котором один класс разделяет структуру и поведение, определенные в одном другом (простое наследование) или во многих других (множественное наследование) классах.

Между *n* классами наследование определяет иерархию *«является»* (*«is a»*), при которой подкласс наследует от одного или нескольких более общих суперклассов. Говорят, что подкласс *является* специализацией его суперкласса (за счет дополнения или переопределения существующей структуры или поведения).

Пример 8.8. Дана система для записи параметров полета в черный ящик, установленный в самолете. Организуем систему в виде иерархии классов, построенной на базе наследования. Абстракция «верхнего» класса иерархии имеет вид:

```
public class ПараметрыПолета {
    private Integer имя;
    private БортовоеВремя отметкаВремени;

    public ПараметрыПолета() {
        ...
    }

    public void записывать() {
        ...
    }

    public БортовоеВремя текущВремя() {
        ...
    }
}
```

Запись параметров кабины самолета может обеспечиваться следующим классом:

```
public class Кабина extends ПараметрыПолета {
    private Давление параметр1;
    private Кислород параметр2;
    private Температура параметр3;

    public Кабина(Давление давление, Кислород кислород, Температура температура) {
        this.параметр1 = давление;
        this.параметр2 = кислород;
        this.параметр3 = температура;
    }

    public void записывать() {
        ...
    }

    public Давление перепадДавления(){
        ...
    }
}
```

Этот класс наследует структуру и поведение класса ПараметрыПолета, но наращивает его структуру (вводит три новых элемента данных), переопределяет его поведение (метод записывать()) и дополняет его поведение (метод перепадДавления()).

Иерархическая структура классов системы для записи параметров полета, находящихся в отношении наследования, показана на рис. 8.12.

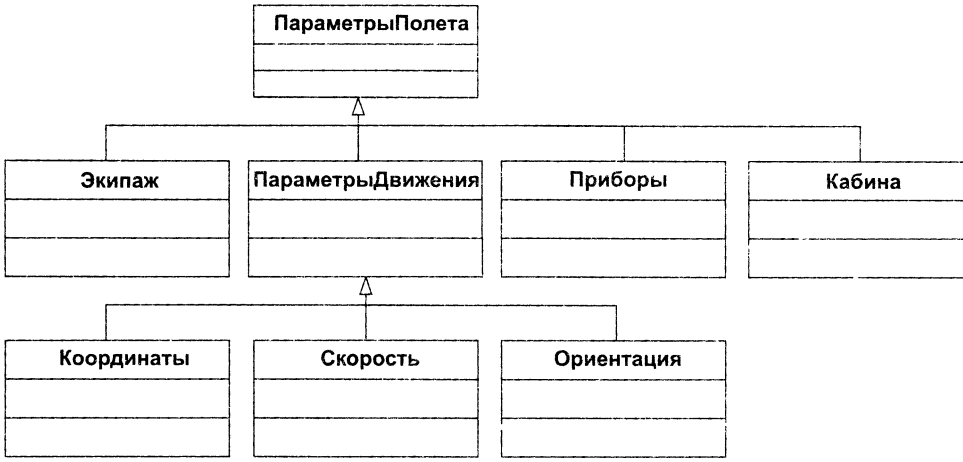


Рис. 8.12. Иерархия простого наследования

Здесь ПараметрыПолета — базовый (корневой) суперкласс, подклассами которого являются Экипаж, ПараметрыДвижения, Приборы, Кабина. В свою очередь, класс ПараметрыДвижения является суперклассом для его подклассов Координаты, Скорость, Ориентация.

Полиморфизм

Полиморфизм — возможность с помощью одного имени обозначать операции из различных классов (но относящихся к общему суперклассу). Вызов обслуживания по полиморфному имени приводит к исполнению одной из некоторого набора операций.

Пример 8.9. Рассмотрим различные реализации метода записывать. Для класса ПараметрыПолета реализация имеет вид:

```
public void записывать() {
    // записывать имя параметра
    // записывать отметку времени
}
```

В классе Кабина предусмотрена другая реализация метода:

```
public void записывать() {
    super.записывать(); // вызов метода суперкласса
    // записывать значение давления
    // записывать процентное содержание кислорода
    // записывать значение температуры
}
```

Предположим, что мы имеем по экземпляру каждого из этих двух классов:

```
ПараметрыПолета вПолете = new ПараметрыПолета();
Кабина вКабине =
        new Кабина(new Давление(768),new Кислород(21),new Температура(20));
```

Будем считать, что где-то возле входной точки программы имеется метод сохранятьНовДанные():

```
private void сохранятьНовДанные(ПараметрыПолета d, БортовоеВремя t) {
    if (текущВремя(d).больше(t)) {
                d.записывать();
    }
}
```

Что случится при выполнении следующих операторов?

- сохранятьНовДанные (вПолете, new БортовоеВремя(60));
- сохранятьНовДанные (вКабине, new БортовоеВремя(120));

Каждый из операторов вызывает операцию записывать нужного класса. В первом случае заработает операция записывать() из класса ПараметрыПолета. Во втором случае будет выполняться операция из класса Кабина. Как видим, в методе сохранятьНовДанные() переменная d может обозначать объекты разных классов, значит, здесь записан вызов полиморфной операции.

Агрегация

Отношения агрегации между классами аналогичны отношениям агрегации между объектами.

Пример 8.10. Повторим пример класса КонтроллерУгла, переписав его на языке C++:

```
class КонтроллерУгла {
    public: // видимая секция
        // описание методов
        void обрабатывать(ГрафикРазворота *уг) {...}
        Секунда запланировано(ГрафикРазворота *уг) {...}
        ...
    private: // скрытная секция данных
        ...
        РегуляторУгла регулятор;
};
```

Видим, что класс КонтроллерУгла является агрегатом, а экземпляр класса РегуляторУгла — это одна из его частей. Агрегация здесь определена как включение по величине. Это — пример физического включения, означающий, что объект регулятор не существует независимо от включающего его экземпляра КонтроллераУгла. Время жизни этих двух объектов неразрывно связано.

Графическая иллюстрация отношения агрегации по величине (композиции) представлена на рис. 8.13.

Возможен косвенный тип агрегации — включением по ссылке.

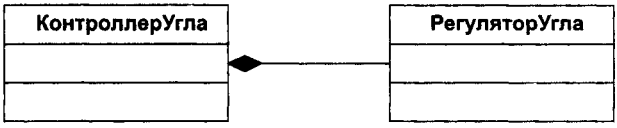


Рис. 8.13. Отношение агрегации по величине (композиция)

Если мы запишем в приватной части класса КонтроллерУгла:

```
...
private: // скрытная секция данных
        ...
        РегуляторУгла *регулятор; // указатель на регулятор
};
```

то регулятор как часть контроллера будет доступен косвенно.

Теперь сцепление объектов уменьшено. Экземпляры каждого класса создаются и уничтожаются независимо.

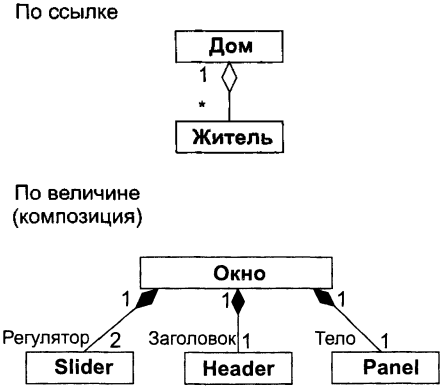


Рис. 8.14. Агрегация классов

Еще два примера агрегации по ссылке и по величине (композиции) приведены на рис. 8.14. Здесь показаны класс-агрегат Дом и класс-агрегат Окно, причем указаны роли и множественность частей агрегата (соответствующие пометки имеют линии отношений).

Как показано на рис. 8.15, возможны и другие формы представления агрегации по величине — композиции. Композицию можно отобразить графическим вложением символов частей в символ агрегата (левая часть рис. 8.15). Вложенные части демонстрируют свою множественность (мощность, кратность) в правом верхнем углу своего символа. Если метка множественности опущена, по умолчанию считают, что ее значение «много». Вложенный элемент может иметь роль в агрегате. Используется синтаксис:

```
роль:имяКласса
```

Эта роль соответствует той роли, которую играет часть в неявном (в этой нотации) отношении композиции между частью и целым (агрегатом).

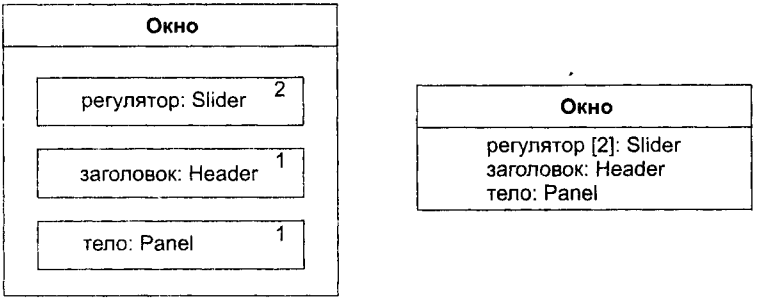


Рис. 8.15. Формы представления композиции

Отметим, что, как представлено в правой части рис. 8.15, в сущности, атрибуты класса находятся в отношении композиции между всем классом и его элементами-атрибутами. Тем не менее в общем случае атрибуты должны иметь примитивные значения (числа, строки, даты), а не ссылаться на другие классы, так как в «атрибутной» нотации не видны другие отношения классов-частей. Кроме того, атрибуты классов не могут находиться в совместном использовании несколькими классами.

Зависимость

Зависимость — это отношение, которое показывает, что изменение в одном классе (независимом) может влиять на другой класс (зависимый), который использует его. Графически зависимость изображается как пунктирная стрелка, направленная на класс, от которого зависят. С помощью зависимости уточняют, какая абстракция является клиентом, а какая — поставщиком определенной услуги. Пунктирная стрелка зависимости направлена от клиента к поставщику.

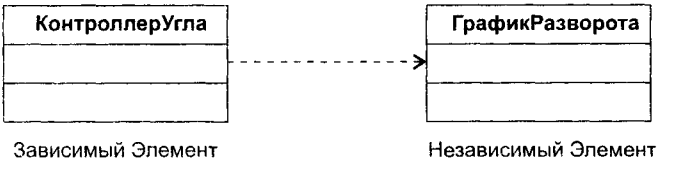


Рис. 8.16. Отношение Зависимости

Наиболее часто зависимости показывают, что один класс использует другой класс как аргумент в сигнатуре своей операции. В предыдущем примере класс ГрафикРазворота появляется как аргумент в методах обрабатывать() и запланировано() класса КонтроллерУгла. Поэтому, как показано на рис. 8.16, КонтроллерУгла зависит от класса ГрафикРазворота.

Конкретизация

Г. Буч определяет конкретизацию как процесс наполнения шаблона (родового или параметризованного класса). Целью является получение класса, от которого возможно создание экземпляров [38].

Параметризованный класс служит заготовкой, шаблоном, параметры которого могут наполняться (настраиваться) другими классами, типами, объектами, операциями. Он может быть родоначальником большого количества обычных (конкретных) классов. Возможности настройки параметризованного класса представляются списком формальных родовых параметров. Эти параметры в процессе настройки должны заменяться фактическими родовыми параметрами. Процесс настройки родового класса называют конкретизацией.

В разных языках программирования родовые классы оформляются по-разному. Впервые идея настройки-параметризации была реализована в языке Ada, затем она проникла в язык C++. Мы же воспользуемся возможностями языка Java. Здесь формальные родовые параметры записываются между угловыми скобками и помещаются в заголовок класса.

Пример 8.11. Представим родовой (параметризированный) класс Очередь.

```
public class Очередь<Элемент> {
    ...
    public void добавить(Элемент элемент) {
        ...
    }
    ...
}
```

У этого класса один формальный родовой параметр – тип Элемент. Вместо этого параметра можно подставить почти любой тип данных.

Произведем настройку, то есть объявим два конкретизированных класса — ОчередьЦелыхЭлементов и ОчередьЛилипутов:

```
class ОчередьЦелыхЭлементов extends Очередь<Integer>{
    ...
}

class ОчередьЛилипутов extends Очередь<Лилипут>{
    ...
}
```

В первом случае мы настраивали класс на конкретный тип Integer (фактический родовой параметр), во втором случае — на конкретный тип Лилипут.

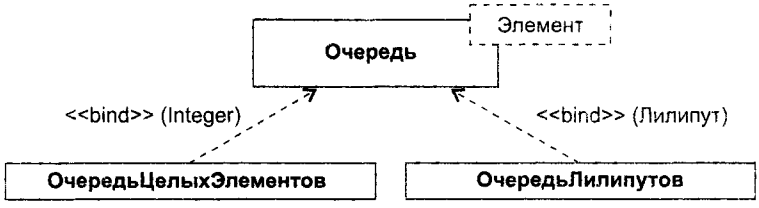


Рис. 8.17. Отношения конкретизации параметризованного класса

Классы ОчередьЦелыхЭлементов и ОчередьЛилипутов можно использовать как обычные классы. Они содержат все средства параметризованного класса, но только эти средства настроены на использование конкретного типа, заданного при конкретизации.

Графическая иллюстрация отношений конкретизации приведена на рис. 8.17. Отметим, что отношение конкретизации отображается с помощью подписанной стрелки отношения зависимости. Это логично, поскольку конкретизированный класс зависит от параметризованного класса (класса-шаблона).

Базис языка визуального моделирования

Для создания моделей анализа и проектирования объектно-ориентированных программных систем используют языки визуального моделирования. Появившись сравнительно недавно, в период с 1989 по 1997 год, эти языки уже имеют представительную историю развития.

В настоящее время различают три поколения языков визуального моделирования. И если первое поколение образовали 10 языков, то численность второго поколения уже превысила 50 языков. Среди наиболее популярных языков 2-го поколения можно выделить: язык Буча (G. Booch), язык Рамбо (J. Rumbaugh), язык Якобсона (I. Jacobson), язык Коада—Йордона (Coad-Yourdon), язык Шлеера—Меллора (Shlaer-Mellor) и т. д. [63, 88, 94]. Каждый язык вводил свои выразительные средства, ориентировался на собственный синтаксис и семантику, иными словами — претендовал на роль единственного и неповторимого языка. В результате разработчики (и пользователи этих языков) перестали понимать друг друга. Возникла острая необходимость унификации языков.

Идея унификации привела к появлению языков 3-го поколения. В качестве стандартного языка третьего поколения был выбран Unified Modeling Language (UML), создававшийся в 1994–1997 годах (основные разработчики — три «amigos» Г. Буч, Дж. Рамбо, А. Якобсон). Заботы о развитии UML взял на себя международный консорциум OMG. В 2005 году версия языка UML 1.4.2 была принята в качестве международного стандарта ISO/IEC 19501:2005. В мае 2010 года появилась версия UML 2.3, которая и описывается в этом учебнике [77].

Данный раздел посвящен определению базовых понятий языка UML.

Унифицированный язык моделирования

UML — стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем [39, 77, 91]. UML может использоваться для визуализации, спецификации, разработки и документирования результатов программных проектов. UML — это не визуальный язык программирования, но его модели прямо транслируются в текст на языках программирования (Java, C++, Visual Basic, Ada 2005, Delphi Pascal) и даже в таблицы для реляционной БД.

Основным инструментом представления модели в UML являются диаграммы. *Диаграмма* UML — это нагруженный связный граф. Вершины графа нагружаются элементами модели, а дуги (ребра) — отношениями между элементами.

Одна диаграмма способна изобразить лишь отдельную черту системы, поэтому используют набор диаграмм. Этот набор обеспечивает визуальное представление программной системы с разных точек зрения. Объединение всех точек зрения дает полную картину, достаточную для решения конкретных задач разработки ПО.

Диаграммы UML разделяются на две группы: структурные диаграммы и диаграммы поведения.

Структурные диаграммы отражают статическую структуру элементов в программной системе. Они могут показать структуру на нескольких уровнях:

1. *Архитектурный уровень:*
   1) диаграмма пакетов (package diagram, структура из подсистем, подсистема – это пакет);
   2) диаграмма компонентов (component diagram, структура из подсистем, подсистема – это компонент).
2. *Уровень детального проектирования:*
   3) диаграмма классов (class diagram, структура из классов);
   4) диаграмма объектов (object diagram, снимок объектов системы в конкретный момент времени);
   5) диаграмма композитной структуры (composite structure diagram, внутренняя структура компонента или класса).
3. *Уровень физического размещения:*
   6) диаграмма развертывания (deployment diagram, размещение артефактов разработки системы на компьютерных ресурсах).

Диаграммы поведения описывают динамику системы на следующих этапах разработки:

4. *Формирование требований:*
   7) диаграмма Use Case (Use Case diagram, последовательность действий системы записывается на естественном языке с точки зрения пользователей);
   8) диаграмма деятельности (activity diagram, последовательность действий системы записывается в виде алгоритма).
5. *Анализ требований:*
   9) диаграмма последовательности (sequence diagram, последовательность действий системы записывается в виде сообщений между участниками взаимодействия, акцент – на временной последовательности сообщений);
   10) диаграмма коммуникации (communication diagram, последовательность действий системы записывается в виде сообщений между участниками взаимодействия, акцент – на структурных связях участников);
   11) диаграмма обзора взаимодействий (interaction overview diagram, сочетание диаграммы деятельности и диаграммы последовательности для обзора потоков управления между участниками взаимодействия);
   12) диаграмма синхронизации (timing diagram, последовательность состояний системы во времени, изменения состояний вызываются последовательностью событий).
6. *Детальное проектирование:*
   13) диаграмма конечного автомата (state machine diagram, последовательность состояний в жизненном цикле объекта).

Как видите, список довольно внушителен, язык UML — это большой язык. Здесь уместно процитировать основателей языка — Грэди Буча, Джеймса Рамбо и Айвара Якобсона [91]:

- «UML запутан, неточен, сложен и громоздок. У этого есть как плохие, так и хорошие стороны. Все, что имеет столь же широкое применение, неизбежно будет запутанным.
- Вам не обязательно знать или использовать все возможности UML — точно так же, как не обязательно знать и использовать все возможности большого и сложного программного приложения или языка программирования. Есть ограниченное множество широко используемых ключевых концепций. Все остальные характеристики можно изучать постепенно и использовать по мере необходимости.
- В реальной разработке UML может использоваться и используется множеством способов.
- UML — это нечто большее, чем просто визуальная нотация. По UML-моделям можно генерировать код и варианты тестирования. Это требует применения подходящего профиля UML, наличия инструментальных средств, соответствующих целевой платформе, а также компромисса в выборе среди различных вариантов реализации.
- Не следует прислушиваться слишком внимательно к речам тех, кто претендует на роль законодателей в языке UML. Какого-то единственно верного способа его использования не существует. Это просто один из инструментов для хорошего разработчика. Его не обязательно использовать для всего. Его можно изменить под собственные нужды, заручившись помощью коллег и необходимыми программными средствами».

Механизмы расширения в UML

UML — развитый язык, имеет большие возможности, но даже он не может отразить все нюансы, которые могут возникнуть при создании различных моделей. Поэтому UML создавался как открытый язык, допускающий контролируемые расширения. Механизмами расширения в UML являются:

- ограничения;
- теговые величины;
- стереотипы.

Ограничение (constraint) расширяет семантику строительного UML-блока, позволяя добавить новые правила или модифицировать существующие. Ограничение показывают как текстовую строку, заключенную в фигурные скобки { }. Например, на рис. 8.18 введено простое ограничение на атрибут *сумма* класса *Сессия Банкомата* — его значение должно быть кратно 20. Кроме того, здесь показано ограничение на два элемента (две ассоциации), оно располагается возле пунктирной линии, соединяющей элементы, и имеет следующий смысл — владельцем конкретного счета не может быть и организация, и персона.

Теговая величина (tagged value) расширяет характеристики строительного UML-блока, позволяя создать новую информацию в спецификации конкретного

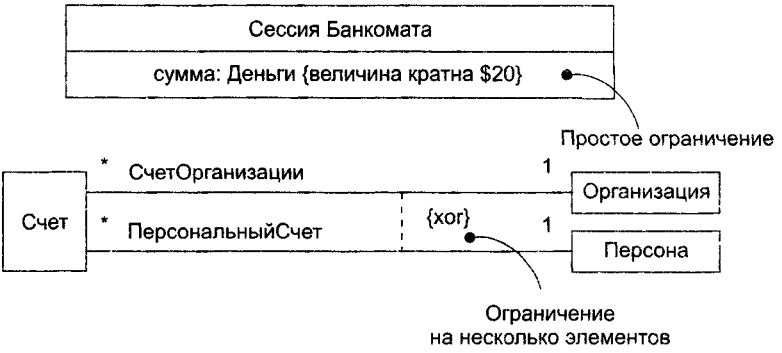


Рис. 8.18. Ограничения

элемента. Теговую величину показывают как строку в фигурных скобках { }. Строка имеет вид:

```
имя теговой величины = значение
```

Иногда (в случае предопределенных тегов) указывается только имя теговой величины.

Отметим, что при работе с продуктом, имеющим много реализаций, полезно отслеживать версию и автора определенных блоков. Версия и автор не принадлежат к основным понятиям UML. Они могут быть добавлены к любому строительному блоку (например, к классу) введением в блок новых теговых величин. Например, на рис. 8.19 класс ТекстовыйПроцессор расширен путем явного указания его версии и автора. Здесь две теговые величины размещены внутри символа примечания (прямоугольника с загнутым углом), которое прикреплено к прямоугольнику класса пунктирной линией. Впрочем, можно обойтись и без примечания: просто поместить теговую строку ниже имени класса.

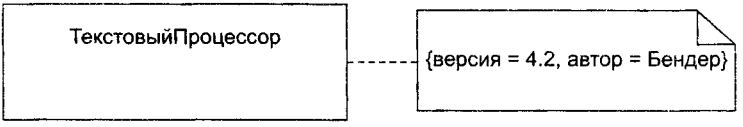


Рис. 8.19. Расширение класса

Стереотип (stereotype) расширяет словарь языка, позволяет создавать новые виды строительных блоков, производные от существующих и учитывающие специфику новой проблемы. Элемент со стереотипом является вариацией существующего элемента, имеющей такую же форму, но отличающуюся по сути. У него могут быть дополнительные ограничения и теговые величины, а также другое визуальное представление. Он иначе обрабатывается при генерации программного кода. Отображают стереотип как имя, указываемое в двойных угловых скобках (или в угловых кавычках).

Примеры элементов со стереотипами приведены на рис. 8.20. Стереотип <<exception>> говорит о том, что класс ПотеряЗначимости теперь рассматривается как специальный класс, которому, положим, разрешается только генерация и об-

ботка сигналов исключений. Особые возможности метакласса получил класс ЭлементМодели. Кроме того, здесь показано применение стереотипа <<call>> к отношению зависимости (у него появился новый смысл).

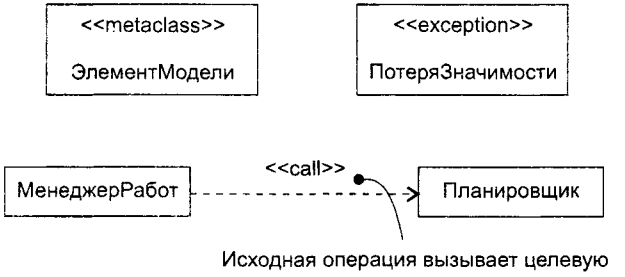


Рис. 8.20. Стереотипы

Таким образом, механизмы расширения позволяют адаптировать UML под нужды конкретных проектов и под новые программные технологии. Возможно появление новых строительных блоков, модификация спецификаций существующих блоков и даже изменение их семантики. Конечно, очень важно обеспечить контролируемое введение расширений.

Контрольные вопросы и упражнения

1. В чем отличие алгоритмической декомпозиции от объектно-ориентированной декомпозиции сложной системы?
2. В чем особенность объектно-ориентированного абстрагирования?
3. В чем особенность объектно-ориентированной инкапсуляции?
4. Каковы средства обеспечения объектно-ориентированной модульности?
5. Каковы особенности объектно-ориентированной иерархии? Какие разновидности этой иерархии вы знаете?
6. Дайте общую характеристику объектов.
7. Что такое состояние объекта?
8. Что такое поведение объекта?
9. Какие виды операций вы знаете?
10. Что такое протокол объекта?
11. Что такое обязанности объекта?
12. Чем отличаются активные объекты от пассивных объектов?
13. Что такое роли объектов?
14. Чем отличается объект от класса?
15. Охарактеризуйте связи между объектами.
16. Охарактеризуйте роли объектов в связях.
17. Какие формы видимости между объектами вы знаете?

18. Охарактеризуйте отношение агрегации между объектами. Какие разновидности агрегации вы знаете?
19. Дайте общую характеристику класса.
20. Поясните внутреннее и внешнее представление класса.
21. Какие вы знаете секции в интерфейсной части класса?
22. Какие виды отношений между классами вы знаете?
23. Поясните ассоциации между классами.
24. Поясните наследование классов.
25. Поясните понятие полиморфизма.
26. Поясните отношения агрегации между классами.
27. Объясните нетрадиционные формы представления агрегации.
28. Поясните отношения зависимости между классами.
29. Поясните отношение конкретизации между классами.
30. Даны два класса:
    - A (включает атрибуты pa1, pa2 и операции opA1(), opA2());
    - B (включает атрибут pb1 и операции opB1(), opB2()).

    При выполнении операции opA1() вызывается операция opB2(). Нарисовать диаграмму.
31. Даны два класса:
    - A (включает атрибуты pa1, pa2 и операции opA1(), opA2());
    - B (включает атрибуты pa1, pa2, pb1 и операции opA1(), opA2(), opB1()).

    Класс B является наследником класса A. Нарисовать диаграмму.
32. Даны три класса:
    - A (включает атрибуты pa1, pa2 и операции opA1(), opA2());
    - B (включает атрибут pb1 и операции opB1(), opB2());
    - C (включает атрибуты pc1, pc2 и операции opC1(), opC2()).

    Классы B и C являются частями (по величине) агрегата — класса A. Класс B входит в агрегат два раза, а класс C — три раза. Нарисовать диаграмму.
33. Даны три класса:
    - A (включает атрибуты pa1, pa2 и операции opA1(), opA2());
    - B (включает атрибут pb1 и операции opB1(), opB2());
    - C (включает атрибуты pc1, pc2 и операции opC1(), opC2()).

    Классы B и C являются частями (по ссылке) агрегата — класса A. Класс B входит в агрегат четыре раза, а класс C — два раза. Нарисовать диаграмму.
34. Даны два класса:
    - A (включает атрибуты pa1, pa2 и операции opA1(), opA2(x: Integer, y: B));
    - B (включает атрибут pb1 и операции opB1(), opB2()).

Класс B является типом второго параметра операции opA2(). При выполнении операции opA1() вызывается операция opB2(). Нарисовать диаграмму.

35. Сколько поколений языков визуального моделирования вы знаете?
36. Назовите численность языков визуального моделирования 2-го поколения.
37. Какая необходимость привела к созданию языка визуального моделирования третьего поколения?
38. Поясните назначение UML.
39. Для чего служат механизмы расширения в UML?
40. Поясните механизм ограничений в UML.
41. Объясните механизм теговых величин в UML.
42. В чем суть механизма стереотипов UML?

Глава 9

Объектно-ориентированная разработка требований

В данной главе рассматривается инструментарий языка UML для решения двух связанных задач: формирования исходных требований заказчика и анализа детальных требований. Подразумевается, что требования задаются в форме желаемого поведения системы. Именно поэтому должны строиться динамические модели, отражающие поведение системы во времени. Средства языка UML для создания динамических моделей многочисленны и разнообразны [22, 38, 63, 77, 91]. В качестве диаграмм для формирования требований обсуждаются диаграмма Use Case и диаграмма деятельности, а в качестве диаграмм для анализа требований — диаграмма коммуникации и диаграмма последовательности. Детально поясняется методика оценки затрат на программный проект, ориентированная на обработку диаграмм Use Case. Дополнительно описываются диаграммы конечных автоматов — мощный инструмент для моделирования объектов, управляемых событиями.

Формирование требований с помощью диаграммы Use Case

Диаграмма Use Case определяет поведение системы с точки зрения пользователя. Диаграмма Use Case рассматривается как главное средство для первичного моделирования динамики системы, используется для выяснения требований заказчика к разрабатываемой системе, фиксации этих требований в форме, которая позволит проводить дальнейшую разработку. В русской литературе диаграммы Use Case часто называют диаграммами прецедентов, или диаграммами вариантов использования.

В состав диаграмм Use Case входят элементы Use Case, актеры, а также отношения зависимости, обобщения и ассоциации. Как и другие диаграммы, диаграммы Use Case могут включать примечания и ограничения.

Актеры и элементы Use Case

Вершинами в диаграмме Use Case являются актеры и элементы Use Case. Их обозначения показаны на рис. 9.1.

Рис. 9.1. Обозначения актера и элемента Use Case

Актеры представляют внешний мир, нуждающийся в работе системы. Элементы Use Case представляют действия, выполняемые системой в интересах актеров.

Актер — это роль объекта вне системы, который прямо взаимодействует с ее частью — конкретным элементом (элементом Use Case). Различают актеров и пользователей. Пользователь — это физический объект, который использует систему. Он может играть несколько ролей и поэтому может моделироваться несколькими актерами. Справедливо и обратное — актером могут быть разные пользователи.

Например, для коммерческого летательного аппарата можно выделить двух актеров: пилота и кассира. Сидоров — пользователь, который иногда действует как пилот, а иногда — как кассир. Как изображено на рис. 9.2, в зависимости от роли Сидоров взаимодействует с разными элементами Use Case.

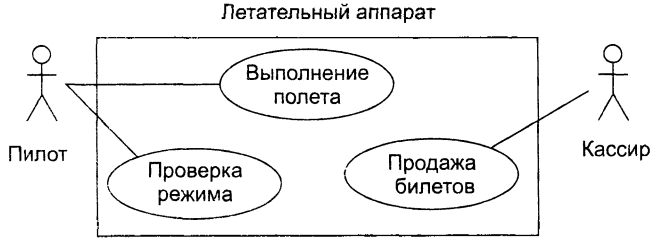


Рис. 9.2. Модель Use Case

Элемент Use Case — это описание последовательности действий (или нескольких последовательностей), которые выполняются системой и производят для отдельного актера видимый результат.

Один актер может использовать несколько элементов Use Case и наоборот: один элемент Use Case может иметь несколько актеров, использующих его. Каждый элемент Use Case задает определенный путь использования системы. Набор всех элементов Use Case определяет полные функциональные возможности системы.

Отношения в диаграммах Use Case

Между актером и элементом Use Case возможен только один вид отношения — ассоциация, отображающая их взаимодействие (рис. 9.3). Как и любая другая ассоциация, она может быть помечена именем, ролями, мощностью.

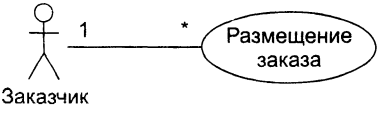


Рис. 9.3. Отношение ассоциации

Между актерами допустимо отношение обобщения (рис. 9.4), означающее, что экземпляр потомка может взаимодействовать с такими же разновидностями экземпляров элементов Use Case, что и экземпляр родителя.

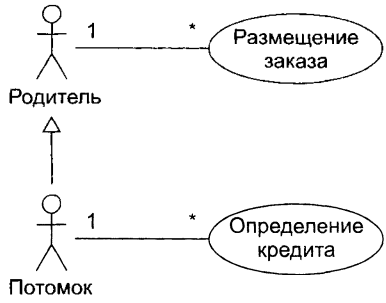


Рис. 9.4. Отношение обобщения между актерами

Между элементами Use Case определено отношение обобщения и две разновидности отношения зависимости — включения и расширения.

Отношение обобщения (рис. 9.5) фиксирует, что потомок наследует поведение родителя. Кроме того, потомок может дополнить или переопределить поведение родителя. Элемент Use Case, являющийся потомком, может замещать элемент Use Case, являющийся родителем, в любом месте диаграммы.

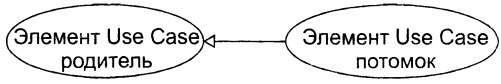


Рис. 9.5. Отношение обобщения между элементами Use Case

Отношение включения (рис. 9.6) между элементами Use Case означает, что базовый элемент Use Case *явно* включает поведение другого элемента Use Case в точке, которая определена в базе. Включаемый элемент Use Case никогда не используется самостоятельно — его содержание может быть только частью другого, большего элемента Use Case. Отношение включения является примером отношения делегации. При этом в отдельное место (включаемый элемент Use Case) помещается определенный набор обязанностей системы. Далее остальные части системы могут агрегировать в себя эти обязанности (при необходимости).

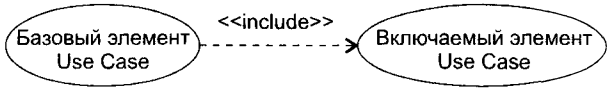


Рис. 9.6. Отношение включения между элементами Use Case

Отношение расширения (рис. 9.7) между элементами Use Case означает, что базовый элемент Use Case *неявно* включает поведение другого элемента Use Case в точке, которая определяется косвенно расширяющим элементом Use Case. Базовый элемент Use Case может быть автономен, но при определенных условиях его поведение может расширяться поведением из другого элемента Use Case. Базовый

Элемент Use Case может расширяться только в определенных точках — точках расширения. Отношение расширения применяется для моделирования выбираемого поведения системы. Таким способом можно отделить обязательное поведение от необязательного поведения. Например, можно использовать отношение расширения для отдельного подпотока, который выполняется только при определенных условиях, находящихся вне поля зрения базового элемента Use Case. Наконец, можно моделировать отдельные потоки, вставка которых в определенную точку управляется актером.

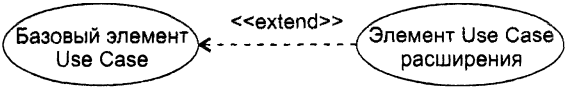


Рис. 9.7. Отношение расширения между элементами Use Case

Пример простейшей диаграммы Use Case, в которой использованы отношения включения и расширения, приведен на рис. 9.8.

Как показано на рис. 9.9, внутри элемента Use Case может быть дополнительная секция с заголовком Extention points. В этой области перечисляются точки расширения. В указанную здесь точку *дополнительные запросы* вставляется последовательность действий от расширяющего элемента Use Case Запрос каталога. Для справки отмечено, что точка расширения размещена после действий, обеспечивающих создание заказа. На этом же рисунке отображены отношения наследования между элементами Use Case. Видно, что элементы Use Case Оплата наличными и Оплата в кредит наследуют поведение элемента Use Case Произвести Оплату и являются его специализациями.

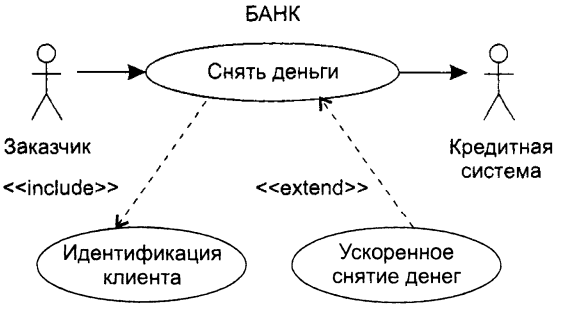


Рис. 9.8. Простейшая диаграмма Use Case для банка

Работа с элементами Use Case

Элемент Use Case описывает, *что* должна делать система, но не определяет, *как* она должна это делать. При моделировании это позволяет отделять внешнее представление системы от ее внутреннего представления.

Поведение элемента Use Case описывается потоком событий. Начальное описание выполняется в текстовой форме, прозрачной для пользователя системы.

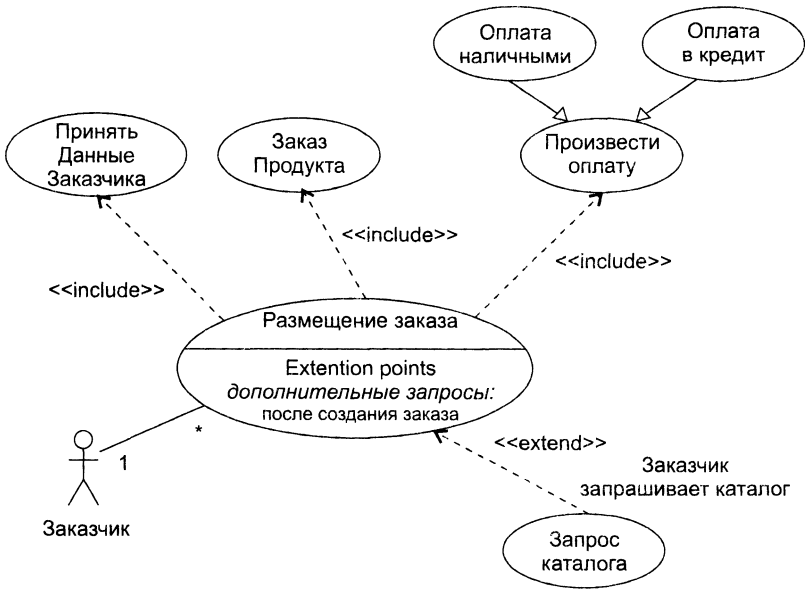


Рис. 9.9. Диаграмма Use Case для обслуживания заказчика

В потоке событий выделяют:

- основной поток и альтернативные потоки поведения;
- как и когда стартует и заканчивается элемент Use Case;
- когда элемент Use Case взаимодействует с актерами;
- какими данными обмениваются актер и система.

Для уточнения и формализации потоков событий используют диаграммы последовательности. Обычно одна диаграмма последовательности определяет главный поток в элементе Use Case, а другие диаграммы — потоки исключений.

В общем случае один элемент Use Case описывает набор последовательностей, в котором каждая последовательность представляет возможный поток событий. Каждая последовательность называется сценарием. Сценарий — конкретная последовательность действий, которая иллюстрирует поведение. Сценарии являются для элемента Use Case почти тем же, чем являются экземпляры для класса. Говоря т. что сценарий — это экземпляр элемента Use Case.

Спецификация элементов Use Case

Спецификация элемента Use Case — основной источник информации для выполнения анализа требований и проектирования системы. Очень важно, чтобы содержание спецификации было представлено в полной и конструктивной форме. В общем случае спецификация включает главный поток, подпотоки и альтернативные потоки поведения. В качестве шаблона спецификации представим описание элемента Use Case *«Покупать авиабилет»* для модели информационной системы авиакассы.

Предусловие: перед началом этого элемента Use Case должен быть выполнен элемент Use Case «Заполнить базу данных авиарейсов».

Главный поток

Этот элемент Use Case начинается, когда покупатель регистрируется в системе и вводит свой пароль. Система проверяет — правилен ли пароль (E-1), и предлагает покупателю выбрать одно из действий: СОЗДАТЬ, УДАЛИТЬ, ПРОВЕРИТЬ, ВЫПОЛНИТЬ, ВЫХОД.

1. Если выбрано действие СОЗДАТЬ, выполняется подпоток S-1: создать заказ авиабилета.
2. Если выбрано действие УДАЛИТЬ, выполняется подпоток S-2: удалить заказ авиабилета.
3. Если выбрано действие ПРОВЕРИТЬ, выполняется подпоток S-3: проверить заказ авиабилета.
4. Если выбрано действие ВЫПОЛНИТЬ, выполняется подпоток S-4: реализовать заказ авиабилета.
5. Если выбрано действие ВЫХОД, элемент Use Case заканчивается.

Подпотоки

S-1: создать заказ авиабилета. Система отображает диалоговое окно, содержащее поля для пункта назначения и даты полета. Покупатель вводит пункт назначения и дату полета (E-2). Система отображает параметры авиарейсов (E-3). Покупатель выбирает авиарейс. Система связывает покупателя с выбранным авиарейсом (E-4). Возврат к началу элемента Use Case.

S-2: удалить заказ авиабилета. Система отображает параметры заказа. Покупатель подтверждает решение о ликвидации заказа (E-5). Система удаляет связь с покупателем (E-6). Возврат к началу элемента Use Case.

S-3: проверить заказ авиабилета. Система выводит (E-7) и отображает параметры заказа авиабилета: номер рейса, пункт назначения, дата, время, место, цена. Когда покупатель указывает, что он закончил проверку, выполняется возврат к началу элемента Use Case.

S-4: реализовать заказ авиабилета. Система запрашивает параметры кредитной карты покупателя. Покупатель вводит параметры своей кредитной карты (E-8). Возврат к началу элемента Use Case.

Альтернативные потоки

E-1: введен неправильный ID-номер покупателя. Покупатель может повторить ввод ID-номера или прекратить элемент Use Case.

E-2: введен неправильный пункт назначения/дата полета. Покупатель может повторить ввод пункта назначения/даты полета или прекратить элемент Use Case.

E-3: нет подходящих авиарейсов. Покупатель информируется, что в данное время такой полет невозможен. Возврат к началу элемента Use Case.

E-4: не может быть создана связь между покупателем и авиарейсом. Информация сохраняется, система создаст эту связь позже. Элемент Use Case продолжается.

E-5: введен неправильный номер заказа. Покупатель может повторить ввод правильного номера заказа или прекратить элемент Use Case.

E-6: не может быть удалена связь между покупателем и авиарейсом. Информация сохраняется, система будет удалять эту связь позже. Элемент Use Case продолжается.

E-7: система не может вывести информацию заказа. Возврат к началу элемента Use Case.

E-8: некорректные параметры кредитной карты. Покупатель может повторить ввод параметров карты или прекратить элемент Use Case.

Таким образом, в данной спецификации зафиксировано, что внутри элемента Use Case находится один основной поток и двенадцать вспомогательных потоков действий. В дальнейшем разработчик может принять решение о выделении из этого элемента Use Case самостоятельных элементов Use Case. Очевидно, что если самостоятельный элемент Use Case содержит подпоток, то его следует подключать к базовому элементу Use Case отношением include. В свою очередь, самостоятельный элемент Use Case с альтернативным потоком подключается к базовому элементу Use Case отношением extend.

Банкомат — пример диаграммы Use Case

Наибольшие трудности при построении диаграмм Use Case вызывает применение отношений включения и расширения. Очень важно разобраться в отличительных особенностях этих отношений, специфике взаимодействия элементов Use Case, соединяемых с их помощью.

Пример диаграммы Use Case для банкомата, в которой использованы отношения включения и расширения, приведен на рис. 9.10.

В этой диаграмме один базовый элемент Use Case Сеанс Банкомата, который взаимодействует с актером Клиент. К базовому элементу Use Case подключены два расширяющих элемента Use Case (Состояние, Снять) и два включаемых элемента Use Case (Идентификация клиента, Проверка счета). В свою очередь, к элементу Use Case Идентификация клиента подключен включаемый элемент Use Case Проверить достоверность, а к элементу Use Case Снять — расширяющий элемент Use Case Захват карты (он же расширяет элемент Use Case Проверить достоверность).

Видим, что элемент Use Case Сеанс Банкомата имеет две точки расширения (диалог возможен, выдача квитанции), а элементы Use Case Снять и Проверить достоверность — по одной точке расширения (проверка снятия и проверка соответственно). В точки расширения возможна вставка поведения из расширяющего элемента Use Case. Вставка происходит, если выполняется условие расширения:

- для расширяющего элемента Use Case Состояние это условие запрос состояния;
- для расширяющего элемента Use Case Снять это условие запрос снятия;
- для расширяющего элемента Use Case Захват карты это условие список подозрений.

Для расширяемого (базового) элемента Use Case эти условия являются внешними, то есть формируемыми вне его. Иными словами, элементу Use Case Сеанс Банкомата ничего неизвестно об условиях запрос состояния и запрос снятия, а элементам Use Case Снять и Проверить достоверность — об условии список подозрений. Условия расширения являются следствиями событий, происходящих во внешней среде.

Стрелки расширения в диаграмме подписаны. Помимо стереотипа здесь указаны:

- в круглых скобках — имена точек расширения;
- в квадратных скобках — условие расширения.

Описание расширяющего элемента Use Case разделено на сегменты, каждый сегмент обслуживает одну точку расширения базового элемента Use Case.

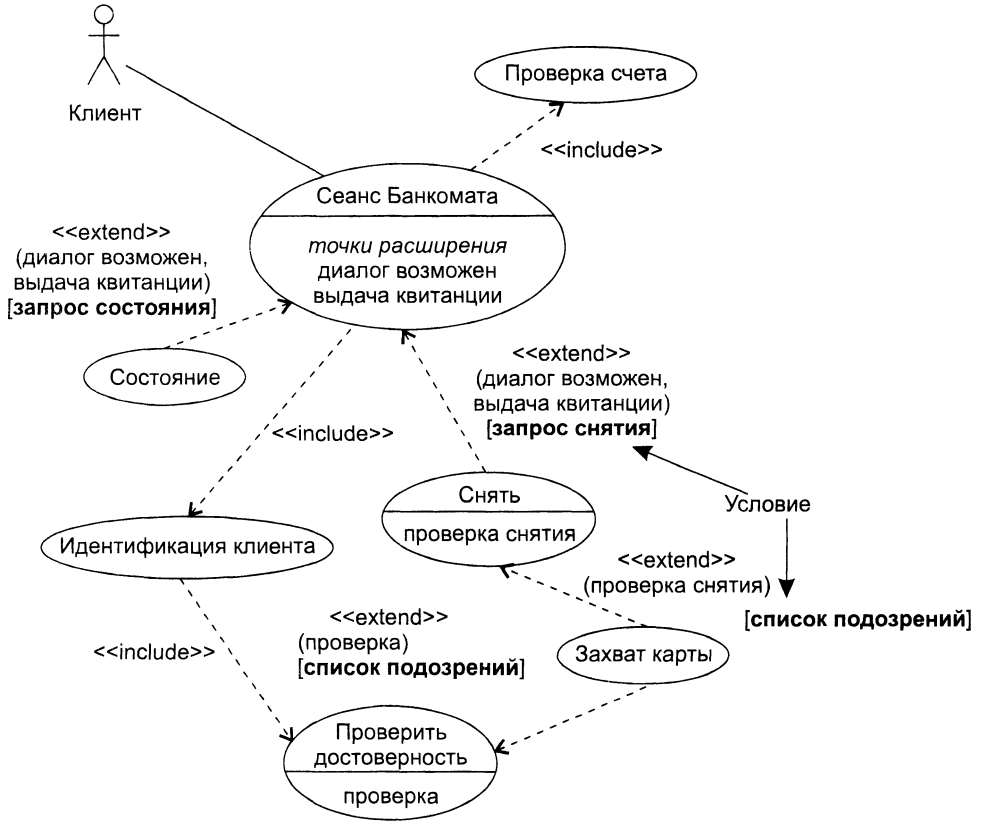


Рис. 9.10. Диаграмма Use Case для банкомата (с использованием включения и расширения)

Количество сегментов расширяющего элемента Use Case равно количеству точек расширения базового элемента Use Case. Первый сегмент расширяющего элемента Use Case начинается с условия расширения, условие записывается только один раз, его действие распространяется и на все остальные сегменты.

Поведение базового элемента Use Case задается внутренним потоком событий вплоть до точки расширения. В точке расширения возможно выполнение расширяющего элемента Use Case, после чего возобновляется работа внутреннего потока.

Спецификации элементов Use Case рассматриваемой диаграммы имеют следующий вид.

Элемент Use Case Сеанс Банкомата

include (Идентификация клиента)	//включение
include (Проверка счета)	//включение
(диалог возможен)	//первая точка расширения
напечатать заголовок квитанции	
(выдача квитанции)	//вторая точка расширения
конец сеанса	

Расширяющий элемент Use Case Состояние

сегмент	//начало первого сегмента
принять запрос состояния	//условие расширения
отобразить информацию о состоянии счета	
сегмент	//начало второго сегмента
напечатать информацию о состоянии счета	

Расширяющий элемент Use Case Снять

сегмент	//начало первого сегмента
принять запрос снятия	//условие расширения
определить сумму	
(проверка снятия)	//точка расширения
сегмент	//начало второго сегмента
напечатать снимаемую сумму	
выдать наличные деньги	

Расширяющий элемент Use Case Захват карты

сегмент	//начало единственного сегмента
принять список подозрений	//условие расширения
проглотить карту	
конец сеанса	

Включаемый элемент Use Case Идентификация клиента

получить имя клиента	
include (Проверить достоверность)	//включение
получить номер счета клиента	

Включаемый элемент Use Case Проверка счета

установить соединение с Базой данных счетов	
получить состояние и ограничения счета	

Включаемый элемент Use Case Проверить достоверность

установить соединение с Базой данных клиентов	
получить параметры клиента	
(проверка)	//точка расширения

Опишем возможное поведение модели, задаваемое этой диаграммой.

Актер Клиент инициирует действия базового элемента Use Case Сеанс Банкомата. На первом шаге запускается включаемый элемент Use Case Идентификация клиента. Этот элемент Use Case получает имя клиента и запускает элемент Use Case Проверить достоверность, в результате чего устанавливается соединение с базой данных клиентов и получаются параметры клиента.

Если к этому моменту исполняется условие расширения список подозрений, то «срабатывает» расширяющий элемент Use Case Захват карты, карта арестовывается и работа системы прекращается.

В противном случае происходит возврат к элементу Use Case Идентификация клиента, который получает номер счета клиента и возвращает управление базовому элементу Use Case.

Базовый элемент Use Case переходит ко второму шагу работы — вызывает включаемый элемент Use Case Проверка счета, который устанавливает соединение с базой данных счетов и получает состояние и ограничения счета.

Управление опять возвращается к базовому элементу Use Case. Базовый элемент Use Case переходит к первой точке расширения диалог возможен. В этой точке возможно подключение одного из двух расширяющих элементов Use Case.

Положим, что к этому моменту выполняется условие расширения запрос состояния, поэтому запускается первый сегмент элемента Use Case Состояние. В результате отображается информация о состоянии счета и управление передается базовому элементу Use Case. В базовом элементе Use Case печатается заголовок квитанции и обеспечивается переход ко второй точке расширения выдача квитанции.

Поскольку в активном состоянии продолжает находиться расширяющий элемент Use Case Состояние, запускается его второй сегмент — в квитанции печатается информация о состоянии счета.

В последний раз управление возвращается к базовому элементу Use Case — завершается сеанс работы банкомата.

ПРИМЕЧАНИЕ

По правилам языка UML полную диаграмму Use Case можно помещать в рамку с пятиугольником в левом верхнем углу. Имя диаграммы записывается в пятиугольнике вслед за пометкой use case. Рамка считается дополнительной презентационной возможностью, то есть не является обязательным элементом диаграммы.

Аспекты банкомата

Заметим, что по сути каждый элемент Use Case задает некоторый набор требований и соответствует некоторому набору понятий. В дальнейшем каждый элемент Use Case будет реализован определенным набором классов. Если некий элемент Use Case представлен как расширяющий, то это значит, что задаваемый им набор понятий пересекает набор понятий другого, базового элемента Use Case. Из этого следует: реализация расширяющего элемента Use Case будет пересекать реализацию базового элемента Use Case. Рассмотрим элемент Use Case банкомата по имени Захват карты. Он расширяет, то есть пересекает два элемента: Проверить достоверность и Снять.

Это значит, что на этапе проектирования его следует реализовать в виде аспекта. Напомним, что аспект — это отдельный программный элемент, для которого определяются три элемента:

- Совет (advice) — программный код аспекта, реализующий некоторое пересекающее требование заказчика;
- Точка соединения (join point) — место в процессе работы банкомата, куда *может* вплетаться совет;
- Срез (pointcut) — инструкция аспекта, указывающая, в какие точки соединения и при каких условиях *должен* вплетаться совет.

Для элемента Use Case Захват карты:

Совет — «проглотить карту».

Точка соединения — «список подозрений».

Срез — «(проверка достоверности & список подозрений) или (проверка снятия & список подозрений)».

Построение модели требований

Напомним, что основное назначение диаграмм Use Case — формирование требований заказчика к будущему программному приложению. Обсудим разработку ПО для машины утилизации, которая принимает использованные бутылки, банки, ящики. Для определения элементов Use Case, которые должны выполняться в системе, вначале определяют актеров.

Выбор актеров

Поиск актеров — большая работа. Сначала выделяют первичных актеров, использующих систему по прямому назначению. Каждый из первичных актеров участвует в выполнении одной или нескольких главных задач системы. В нашем примере первичным актером является Потребитель. Потребитель кладет в машину бутылки, получает квитанцию от машины.

Кроме первичных, существуют и вторичные актеры. Они наблюдают и обслуживают систему. Вторичные актеры существуют только для того, чтобы первичные актеры могли использовать систему. В нашем примере вторичным актером является Оператор. Оператор обслуживает машину и получает дневные отчеты о ее работе. Мы не будем нуждаться в операторе, если не будет потребителей.

Таким образом, внешняя среда машины утилизации имеет вид, представленный на рис. 9.11.

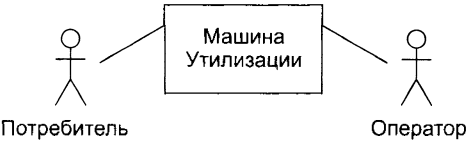


Рис. 9.11. Внешняя среда машины утилизации

Деление актеров на первичных и вторичных облегчает выбор системной архитектуры в терминах основного функционального назначения. Системную структуру определяют в основном первичные актеры. Именно от них в систему приходят главные изменения. Поэтому полное выделение первичных актеров гарантирует, что архитектура системы будет настроена на большинство важных пользователей.

Определение элементов Use Case

После выбора внешней среды можно выявить внутренние функциональные возможности системы. Для этого определяются элементы Use Case.

Каждый элемент Use Case задает некоторый путь использования системы, выполнение некоторой части функциональных возможностей. Полная совокупность элементов Use Case определяет все существующие пути использования системы.

Элемент Use Case — это последовательность взаимодействий в диалоге, выполняемом актером и системой. Запускается элемент Use Case актером, поэтому удобно выявлять элементы Use Case с помощью актеров.

Рассматривая каждого актера, мы решаем, какие элементы Use Case он может выполнять. Для этого изучается описание системы (с точки зрения актера) или проводится обсуждение с теми, кто будет действовать как актер.

Перейдем к примеру. Потребитель — первичный актер, поэтому начнем с этой роли. Этот актер должен выполнять возврат утилизируемых элементов. Так формируется элемент Use Case Возврат Элемента. Приведем его текстовое описание:

```
Начинается, когда потребитель начинает возвращать банки, бутылки, ящики. Для каждого элемента, помещенного в машину утилизации, система увеличивает количество элементов, принятых от Потребителя, и общее количество элементов этого типа за день. После сдачи всех элементов Потребитель нажимает кнопку квитанции, чтобы получить квитанцию, на которой напечатаны названия возвращенных элементов и общая сумма возврата.
```

Следующий актер — Оператор. Он получает дневной отчет об элементах, сданных за день. Это образует элемент Use Case Создание Дневного Отчета. Его описание:

```
Начинается оператором, когда он хочет получить информацию об элементах, сданных за день. Система печатает количество элементов каждого типа и общее количество элементов, полученных за день.
Для подготовки к созданию нового дневного отчета сбрасывается в ноль параметр Общее количество.
```

Кроме того, актер Оператор может изменять параметры сдаваемых элементов. Назовем соответствующий элемент Use Case Изменение Элемента. Его описание:

```
Могут изменяться цена и размер каждого возвращаемого элемента. Могут добавляться новые типы элементов.
```

После выявления всех элементов диаграмма Use Case системы принимает вид, показанный на рис. 9.12.

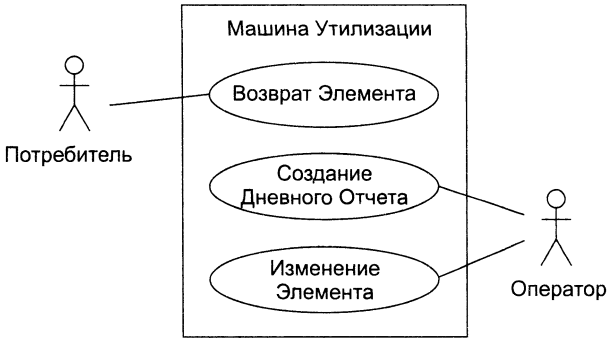


Рис. 9.12. Диаграмма Use Case для машины утилизации

Чаще всего полные описания элементов Use Case формируются за несколько итераций. На каждом шаге в описание вводятся дополнительные детали. Например окончательное описание Возврата Элемента может иметь вид:

```
Когда потребитель возвращает сдаваемый элемент, элемент измеряется системой. Измерения позволяют определить тип элемента. Если тип допустим, то увеличивается количество элементов этого типа, принятых от Потребителя, и общее количество элементов этого типа за день.
Если тип недопустим, то на панели машины высвечивается «недействительно».
Когда Потребитель нажимает кнопку квитанции, принтер печатает дату. Производятся вычисления. По каждому типу принятых элементов печатается информация: название, принятое количество, цена, итого для типа. В конце печатается сумма, которую должен получить потребитель.
```

Не всегда очевидно, как распределить функциональные возможности по отдельным элементам Use Case и что является вариантом одного и того же элемента Use Case. Основной критерий выбора — сложность элемента Use Case. При анализе вариантов поведения рассматривают их различия. Если различия малы, варианты встраивают в один элемент Use Case. Если различия велики, то варианты описываются как отдельные элементы Use Case.

Обычно элемент Use Case задает одну основную и несколько альтернативных последовательностей событий.

Каждый элемент Use Case выделяет частный аспект функциональных возможностей системы. Поэтому элементы Use Case обеспечивают инкрементную схему анализа функций системы. Можно независимо разрабатывать элементы Use Case для разных функциональных областей, а позднее соединить их вместе (для формирования полной модели требований).

Вывод: на основе элементов Use Case в каждый момент времени можно концентрировать внимание на одной частной проблеме, что позволяет вести параллельную разработку.

Расширение функциональных возможностей

Для добавления в элемент Use Case новых действий удобно применять отношение расширения. С его помощью базовый элемент Use Case может быть расширен новым элементом Use Case.

В нашем примере поведение системы не определено для случая, когда сдаваемый элемент застрял. Введем элемент Use Case Элемент Застрял, который будет расширять базовый элемент Use Case Возврат Элемента (рис. 9.13).

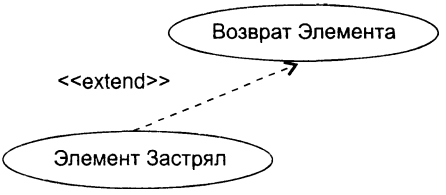


Рис. 9.13. Расширение элемента Use Case Возврат Элемента

Описание элемента UseCase Элемент Застрял может иметь вид:

```
Если элемент застрял, для вызова Оператора вырабатывается сигнал тревоги. После
удаления застрявшего элемента Оператор сбрасывает сигнал тревоги. В результате По-
требитель может продолжить сдачу элементов. Величина ИТОГО сохраняет правильное
значение. Цена застрявшего элемента не засчитывается.
```

Таким образом, описание базового элемента остается прежним, простым. Еще один пример приведен на рис. 9.14.

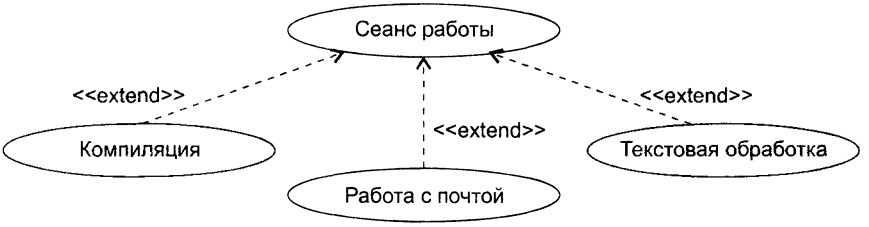


Рис. 9.14. Применение отношения расширения

Здесь мы видим только один базовый элемент Use Case Сеанс работы. Все остальные элементы Use Case могут добавляться как расширения. Базовый элемент Use Case при этом остается почти без изменений.

Отношение расширения определяет прерывание базового элемента Use Case, которое происходит для вставки другого элемента Use Case. Базовый элемент Use Case не знает, будет выполняться прерывание или нет. Вычисление условий прерывания находится вне компетенции базового элемента Use Case.

В расширяющем элементе Use Case указывается ссылка на то место базового элемента Use Case, куда он будет вставляться (при прерывании). После выполнения расширяющего элемента Use Case продолжается выполнение базового элемента Use Case.

Обычно расширения используют:

- для моделирования вариантных частей элементов Use Case;
- для моделирования сложных и редко выполняемых альтернативных последовательностей;
- для моделирования подчиненных последовательностей, которые выполняются только в определенных случаях;
- для моделирования систем с выбором на основе меню.

Главное, что следует помнить: решение о выборе, подключении варианта на основе расширения принимается вне базового элемента Use Case. Если же вы вводите в базовый элемент Use Case условную конструкцию, конструкцию выбора, то придется применять отношение включения. Это случай, когда «штурвал управления» находится в руках базового элемента Use Case.

Уточнение модели требований

Уточнение модели сводится к выявлению одинаковых частей в элементах Use Case и извлечению этих частей. Любые изменения в такой части, выделенной в отдельный элемент Use Case, будут автоматически влиять на все элементы Use Case, которые используют ее совместно.

Извлеченные элементы Use Case называют абстрактными. Они не могут быть конкретизированы сами по себе, применяются для описания одинаковых частей в других, конкретных элементах Use Case. Таким образом, описания абстрактных элементов Use Case используются в описаниях конкретных элементов Use Case. Говорят, что конкретный элемент Use Case находится в отношении «включает» с абстрактным элементом Use Case.

Вернемся к нашему примеру. В этом примере два конкретных элемента Use Case Возврат Элемента и Создание Дневного Отчета имеют общую часть — действия, обеспечивающие печать квитанции. Поэтому, как показано на рис. 9.15, можно выделить абстрактный элемент Use Case Печать. Этот элемент Use Case будет специализироваться на выполнении распечаток.

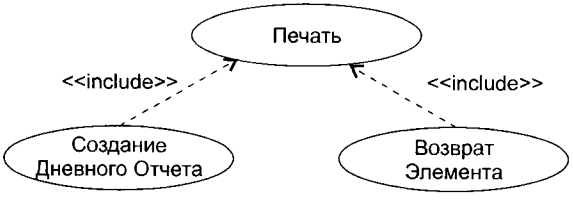


Рис. 9.15. Применение отношения включения

В свою очередь, абстрактные элементы Use Case могут использоваться другими абстрактными элементами Use Case. Так образуется иерархия. При построении иерархии абстрактных элементов Use Case руководствуются правилом: выделение элементов Use Case прекращается при достижении уровня отдельных операций над объектами.

Выделение абстрактных элементов Use Case можно упростить с помощью абстрактных актеров.

Абстрактный актер — это общий фрагмент роли в нескольких конкретных актерах. Абстрактный актер выражает подобия в элементах Use Case. Конкретные актеры находятся в отношении наследования с абстрактным актером. Так, в машине утилизации конкретные актеры имеют одно общее поведение: они могут получать квитанцию. Поэтому можно определить одного абстрактного актера — Получателя Квитанции. Как показано на рис. 9.16, наследниками этого актера являются Потребитель и Оператор.

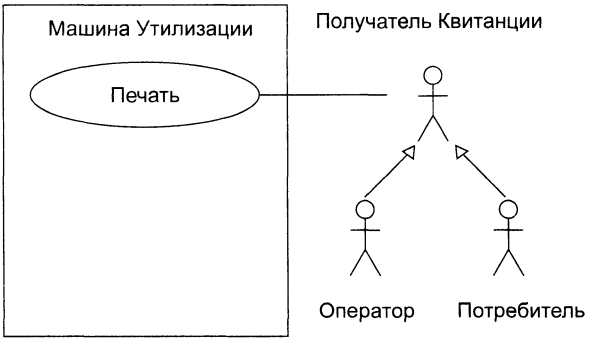


Рис. 9.16. Выделение абстрактного актера

Выводы

1. Абстрактные элементы Use Case находят извлечением общих последовательностей из различных элементов Use Case.
2. Отношение «включает» применяется, если несколько элементов Use Case имеют общее поведение. Цель: устранить повторения, ликвидировать избыточность. Кроме того, это отношение часто используют для ограничения сложности большого элемента Use Case.
3. Отношение «расширяет» применяется, когда описывается вариация, дополняющая нормальное поведение.

Оценка программного проекта на основе диаграммы Use Case

Исследование модели требований заказчика, представляемой диаграммой Use Case, позволяет выделить следующие факторы, влияющие на ресурсы программного проекта:

- Количество шагов для выполнения элемента Use Case.
- Количество и сложность актеров.
- Техническая сложность программного проекта (параллелизм, безопасность, производительность и т. д.).
- Уровень квалификации команды разработчиков (опыт, знания предметной области и т. д.).

Многочисленные источники утверждают, что учет перечисленных факторов дает возможность получить оценку с погрешностью в 20% относительно реальных затрат [23, 42, 43].

Основателем метода, ориентированного на применение так называемых *указателей Use Case* (*UCP – Use Case Points*), является Густав Карнер [65]. В основное уравнение UCP-метода входят три переменные:

1. Исходное количество указателей Use Case (*UUCP – Unadjusted Use Case Point*).
2. Фактор технической сложности (*TCF – The Technical Complexity Factor*).
3. Фактор квалификации разработчиков (*ECF – The Environment Complexity Factor*).

Каждая переменная определяется и вычисляется отдельно, с использованием измерений, весовых коэффициентов и ограничивающих констант. Коэффициенты и константы основаны на статистике многочисленных проектов, выполненных по технологии А. Якобсона. Измерения выполняются командой разработчиков, которые исходят из собственных представлений о технической сложности проекта и возможностях команды.

При добавлении фактора производительности (*PF – Productivity Factor*) основное уравнение позволяет прогнозировать затраты на проект в человеко-часах:

$$ЗАТРАТЫ = UCP \times PF = (UUCP \times TCF \times ECF) \times PF \text{ [чел.-час.]},$$

где *PF* определяет количество человеко-часов для реализации одного указателя Use Case.

В свою очередь, исходное количество указателей Use Case равно сумме двух параметров:

- количеству взвешенных элементов Use Case (*UUCW – The Unadjusted Use Case Weight*),
- количеству взвешенных актеров (*UAW – The Unadjusted Actor Weight*).

Вес элемента Use Case пропорционален общему количеству действий (шагов), содержащихся во всех его сценариях, а вес актера — сложности его описания (роли). По весу все элементы Use Case делятся на три категории: простой, средний, сложный (табл. 9.1), а актеры принадлежат к одному из трех типов (табл. 9.2) с такими же названиями.

Таблица 9.1. Категории элементов Use Case

Категория элемента Use Case	Описание	Вес
Простой	Простой пользовательский интерфейс. Взаимодействует только с одной сущностью БД. В сценарии предусмотрено до трех шагов. Реализуется менее чем пятью классами	5
Средний	Средний по сложности пользовательский интерфейс. Взаимодействует с двумя и более сущностями БД. В сценариях предусмотрено 4–7 шагов. Реализуется 5–10 классами	10

Категория элемента Use Case	Описание	Вес
Сложный	Сложный пользовательский интерфейс. Взаимодействует с тремя и более сущностями БД. В сценариях предусмотрено более семи шагов. Реализуется более чем десятью классами	15

Таблица 9.2. Классификация актеров

Тип актера	Описание	Вес
Простой	Актер представляет внешнюю систему с точно определенным программным интерфейсом	1
Средний	Актер представляет внешнюю систему, которая взаимодействует по протоколу, подобному TCP/IP	2
Сложный	Актер представляет личность, пользующуюся графическим интерфейсом	3

Количество взвешенных элементов Use Case *UUCW* вычисляется подсчетом количества элементов каждой категории, умножением этого количества на соответствующий вес и суммированием результатов по всем категориям. В табл. 9.3 приведены данные по расчету *UUCW* для машины утилизации, описанной в предыдущем подразделе.

Таблица 9.3. Вычисление количества взвешенных элементов Use Case (*UUCW*) для машины утилизации

Категория элемента Use Case	Имена элементов Use Case	Вес	Количество	Результат
Простой	Создание Дневного Отчета Изменение Элемента	5	2	10
Средний	Возврат Элемента Элемент Застрял	10	2	20
Сложный	Нет	15	0	0
UUCW =				30

Количество взвешенных актеров *UAW* вычисляется подсчетом количества актеров каждого типа, умножением этого количества на соответствующий вес и суммированием результатов по всем категориям. В табл. 9.4 приведены данные по расчету *UAW* для машины утилизации, описанной в предыдущем подразделе.

Таблица 9.4. Вычисление количества взвешенных актеров (*UAW*) для машины утилизации

Тип актера	Имена актеров	Вес	Количество	Результат
Простой	Нет	1	0	0
Средний	Нет	2	0	0
Сложный	Потребитель Оператор	3	2	6
UAW =				6

Таким образом, для машины утилизации исходное количество указателей Use Case равно

$$UUCP = UUCW + UAW = 30 + 6 = 36.$$

Теперь нужно вычислить факторы *TCF* и *ECF*.

Фактор технической сложности проекта *TCF* определяется с учетом 13 показателей технической сложности (табл. 9.5). Каждому показателю команда разработчиков назначает значение в диапазоне от 0 до 5. При выборе значения исходят из влияния показателя на сложность конкретного проекта: нуль означает, что влияния нет, 3 — показатель оказывает среднее влияние, 5 — показатель оказывает сильное влияние. Обычно, если влияние неизвестно, выбирают среднее значение — 3.

Таблица 9.5. *Показатели технической сложности проекта*

Показатель технической сложности T_i	**Описание**	**Вес** W_i
T_1	Распределенная система	2
T_2	Высокая производительность	1
T_3	Эффективность работы конечных пользователей	1
T_4	Сложная обработка данных	1
T_5	Повторное использование кода	1
T_6	Простота установки	0,5
T_7	Простота использования	0,5
T_8	Переносимость	2
T_9	Простота внесения изменений	1
T_{10}	Параллелизм	1
T_{11}	Специальные требования к безопасности	1
T_{12}	Непосредственный доступ к системе со стороны третьих лиц (внешних пользователей)	1
T_{13}	Специальные требования к обучению пользователей	1

Значение фактора технической сложности проекта рассчитывают по формуле:

$$TCF = C_1 + C_2 \times \sum_{i=1}^{13} W_i \times T_i,$$

где $C_1 = 0{,}6$, $C_2 = 0{,}01$, W_i — вес i-го показателя технической сложности, T_i — значение i-го показателя технической сложности.

Константы ограничивают воздействие фактора *TCF* на указатель Use Case в диапазоне от 0,6 (все показатели равны нулю) до 1,3 (все показатели равны 5). Таким образом, фактор технической сложности может оказывать как положительное влияние на затраты (уменьшая их на 40%), так и отрицательное влияние (увеличивая их на 30%).

Пример вычисления фактора *TCF* для машины утилизации приведен в табл. 9.6.

Таблица 9.6. Вычисление фактора технической сложности проекта *TCF* для машины утилизации

Показатель технической сложности T_i	Описание	Вес W_i	Значение	Результат
T_1	Распределенная система	2	0	0
T_2	Высокая производительность	1	3	3
T_3	Эффективность работы конечных пользователей	1	5	5
T_4	Сложная обработка данных	1	1	1
T_5	Повторное использование кода	1	5	5
T_6	Простота установки	0,5	5	2,5
T_7	Простота использования	0,5	5	2,5
T_8	Переносимость	2	5	10
T_9	Простота внесения изменений	1	5	5
T_{10}	Параллелизм	1	0	0
T_{11}	Специальные требования к безопасности	1	3	3
T_{12}	Непосредственный доступ к системе со стороны третьих лиц (внешних пользователей)	1	0	0
T_{13}	Специальные требования к обучению пользователей	1	0	0
$TCF = 0{,}6 + 0{,}01 \times$ Результат = 0,97				37

Фактор квалификации разработчиков *ECF* определяется с учетом 8 показателей квалификации (табл. 9.7). Каждому показателю команда разработчиков назначает значение в диапазоне от 0 до 5. При выборе значения исходят из влияния показателя на успех конкретного проекта: нуль означает, что влияния нет, 1 — показатель оказывает сильное негативное влияние, 3 — показатель оказывает среднее влияние, 5 — показатель оказывает сильное позитивное влияние. Например, члены команды со слабой мотивацией оказывают сильное негативное влияние (1), а работники со значительным объектно-ориентированным опытом (5) — сильное позитивное влияние на успех проекта.

ПРИМЕЧАНИЕ

Два показателя надо пояснить особо, поскольку рост их значений является негативной тенденцией. Для показателя E_2 нуль означает отсутствие работников с частичной занятостью, 5 — все работники с частичной занятостью. Для показателя E_7 нуль означает простоту использования языка программирования, 5 — высокую сложность использования языка программирования. Влияние их значений учитывается отрицательным знаком соответствующих весовых коэффициентов.

Таблица 9.7. Показатели квалификации разработчиков

Показатель квалификации E_i	**Описание**	**Вес** W_i
E_1	Знакомство с UML	1,5
E_2	Работники с частичной занятостью	–1
E_3	Квалификация аналитика	0,5
E_4	Опыт работы с прикладной областью	0,5
E_5	Опыт использования объектно-ориентированного подхода	1
E_6	Мотивация	1
E_7	Сложность языка программирования	–1
E_8	Стабильность требований	2

Значение фактора квалификации разработчиков рассчитывают по формуле:

$$ECF = C_1 - C_2 \times \sum_{i=1}^{8} W_i \times E_i,$$

где $C_1 = 1{,}4$, $C_2 = 0{,}03$, W_i — вес i-го показателя квалификации, E_i — значение i-го показателя квалификации.

Пример вычисления фактора *ECF* для машины утилизации приведен в табл. 9.8. Как видим, слабым местом команды является отсутствие опыта в данной прикладной области. В целом же команда демонстрирует хороший уровень квалификации и снижает размер объективных затрат на 38%.

Таблица 9.8. Вычисление фактора квалификации разработчиков *ECF* для машины утилизации

Показатель квалификации E_i	**Описание**	**Вес** W_i	**Значение**	**Результат**
E_1	Знакомство с UML	1,5	5	7,5
E_2	Работники с частичной занятостью	–1	0	0
E_3	Квалификация аналитика	0,5	5	2,5
E_4	Опыт работы с прикладной областью	0,5	0	0
E_5	Опыт использования объектно-ориентированного подхода	1	5	5
E_6	Мотивация	1	5	5
E_7	Сложность языка программирования	–1	0	0
E_8	Стабильность требований	2	3	6
$ECF = 1{,}4 - 0{,}03 \times$ Результат $= 0{,}62$				26

Подводя итог оценке нашего примера, вычислим, что затраты на разработку машины утилизации (в указателях Use Case) равны

$$UCP = UUCP \times TCF \times ECF = 36 \times 0.97 \times 0.62 = 21,65.$$

Для перехода к человеко-часам надо определить фактор производительности *PF*.

Здесь есть две возможности:

1. Взять данные по фактору производительности из архива фирмы, по уже выполненному проекту-аналогу.
2. Выбрать значение из диапазона от 15 до 30 человеко-часов на один указатель Use Case. При выборе учитывается квалификация команды. Для новой команды советуют значение 20.

Воспользовавшись советом, определим, что затраты на разработку машины утилизации равны

$$ЗАТРАТЫ = 21,65 \times PF = 21,65 \times 20 = 433 \text{ [чел.-час.]}.$$

Формирование требований с помощью диаграммы деятельности

Диаграмма деятельности представляет желаемое поведение системы в виде последовательно и параллельно выполняемых шагов, соединяемых потоками управления. Шагами являются конкретные действия. Иногда здесь показывают и потоки данных. Словом, диаграммы деятельности очень похожи на блок-схемы алгоритмов (хотя официально они основаны на свободной интерпретации сетей Петри).

Основной вершиной в диаграмме деятельности является узел действия (рис. 9.17), которое изображается как прямоугольник с закругленными углами.

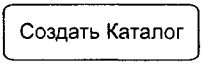


Рис. 9.17. Узел действия

Действие считается атомарным (действие нельзя прервать) и выполняется за один квант времени, его нельзя подвергнуть декомпозиции. Если нужно представить сложное действие, которое можно подвергнуть дальнейшей декомпозиции (разбить на ряд более простых действий), то используют вызов другой деятельности. Изображение вызова деятельности содержит пиктограмму трезубца в правом нижнем углу (рис. 9.18).

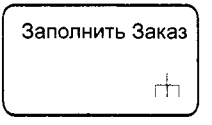


Рис. 9.18. Узел вызова деятельности

Фактически в данную вершину вписывается имя другой диаграммы деятельности, имеющей внутреннюю структуру.

Переходы между вершинами — узлами действий — изображаются в виде стрелок. Переходы выполняются по окончании действий.

Кроме того, в диаграммах деятельности используются узлы управления и узлы объектов:

- разветвление (decision — ромбик с одной входящей и несколькими исходящими стрелками);
- объединение (merge — ромбик с несколькими входящими и одной исходящей стрелкой);
- линейка синхронизации — разделение (fork — жирная горизонтальная линия с одной входящей и несколькими исходящими стрелками);
- линейка синхронизации — слияние (join — жирная горизонтальная линия с несколькими входящими и одной исходящей стрелкой);
- начальный узел деятельности (черный кружок);
- конечный узел деятельности (незакрашенный кружок, в котором размещен черный кружок меньшего размера);
- конечный узел потока (незакрашенный кружок, в котором размещен черный крест X);
- объект (обычный прямоугольник, имя которого подчеркивается).

Вершина «разветвление» позволяет отобразить разветвление вычислительного процесса, исходящие из него стрелки помечаются сторожевыми условиями ветвления.

Вершина «объединение» отмечает точку объединения альтернативных потоков действий.

Линейки синхронизации позволяют показать параллельные потоки действий, отмечая точки их синхронизации при запуске (момент разделения) и при завершении (момент слияния).

ПРИМЕЧАНИЕ

С точки зрения топологии стрелок узлы decision и fork схожи (один входной поток и несколько выходных). Разница между ними в параллельности: на выходе decision-узла появляется только один поток (из нескольких возможных), а на выходе fork-узла — несколько параллельных потоков. То же самое можно сказать о паре merge-join: merge-узел пропускает на выход любой из входных потоков, а join-узел ждет появления всех входных потоков и лишь после этого запускает выходной поток. Иными словами, законы «ромбиков» отличаются от законов «черточек». Прямо как в жизни: несмотря на внешнее сходство, один может оказаться бандитом, а другой — праведником.

Диаграмма деятельности помещается в рамку с пятиугольником в левом верхнем углу. Имя диаграммы записывается вслед за пометкой activity. Пример диаграммы деятельности приведен на рис. 9.19, где показан порядок обработки заказа в интернет-кассе филармонии. Здесь представлена точка разветвления — для принятия решения о покупке отдельного билета или абонемента на весь сезон. Для обеспечения покупки абонемента введены две линейки сихронизации: верхняя отражает разделение на два параллельных процесса, а нижняя — их слияние. Раз-

деление инициирует параллельные действия, которые логически осуществляются в одно и то же время. Ниже слияния показана точка объединения обоих вариантов заказа. В любом варианте деятельность завершается отсылкой уведомления по электронной почте.

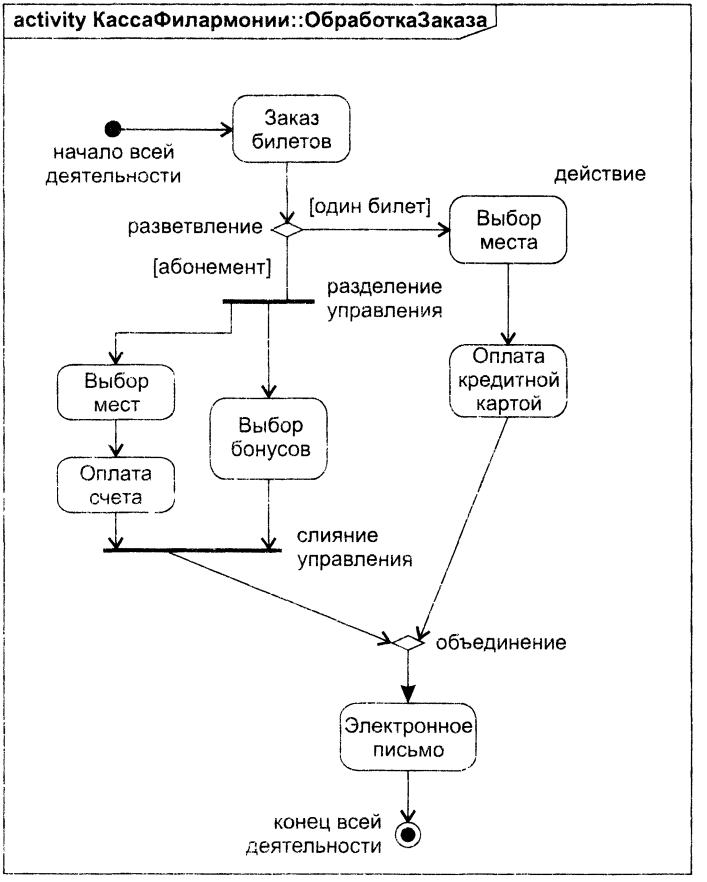


Рис. 9.19. Диаграмма деятельности для покупки билета в филармонию

Элементы диаграммы деятельности можно сгруппировать в разделы, которые иногда называют «плавательными дорожками» из-за способа изображения на диаграмме. Цель такой группировки — обозначить границы ответственности за выполнение конкретных действий. В корпоративных системах разделами могут быть филиалы и отделы. В простых системах в качестве разделов могут выступать подсистемы или крупные объекты.

Наконец, еще одна возможность диаграмм деятельности — отображение информационных потоков с помощью объектов.

На рис. 9.20 вся совокупность действий распределена по трем разделам, каждый из которых имеет свое имя и соответствует отдельному заинтересованному лицу. Разделы отделены друг от друга вертикальными линиями. Кроме того, здесь

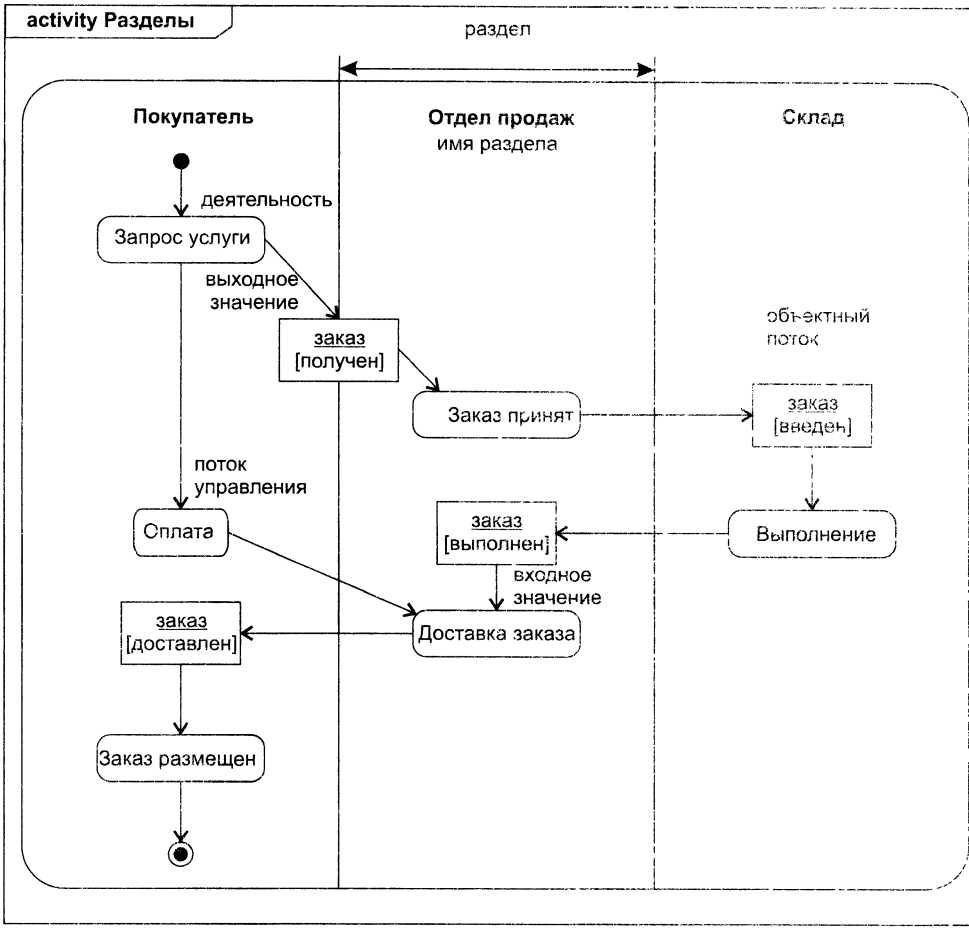


Рис. 9.20. Диаграмма деятельности с «плавательными дорожками»

показаны изменения состояния объекта заказ (по мере того, как он проходит через различные элементы деятельности):

- запись заказ[получен] означает, что заказ обработан действием Запрос услуги;
- запись заказ[введен] свидетельствует, что заказ обработан действием Заказ принят;
- запись заказ[выполнен] говорит о прохождении заказа через действие Выполнение;
- запись заказ[доставлен] фиксирует применение к заказу действия Доставка заказа.

Таким образом, объектный поток заказа отражает траекторию его жизни во времени. Стрелки объектного потока рисуются от действия к объектному узлу, являющемуся результатом данного действия, а также от объектного узла к действию, воспринимающему объект как источник данных.

Анализ требований с помощью диаграмм взаимодействия

Взаимодействия определяют поведение системы в виде коммуникаций между его частями (объектами), представляя систему как сообщество совместно работающих объектов. Именно поэтому взаимодействия считают основным аппаратом для анализа, то есть создания детальных требований к программной системе. В отличие от обобщенных требований заказчика, в которых реализация программной системы не видна, они прямо опираются на элементы программной реализации. Исходными данными для создания взаимодействий являются диаграммы, отражающие требования заказчика (диаграммы Use Case и диаграммы деятельности).

В программной системе объекты должны взаимодействовать друг с другом, посылая сообщения. Взаимодействие — это такое поведение на основе обмена сообщениями между набором объектов, которое обеспечивает реализацию требований к системе.

Взаимодействия моделируют динамику сообщества объектов, играющих конкретные роли и работающих совместно для достижения синергетического эффекта. Фактически взаимодействия используются для моделирования потока управления между участниками системы. Задача взаимодействий — статическими средствами отразить все этапы поведения в жизненном цикле системы. Взаимодействия могут выделять последовательность сообщений во времени или же определять эту последовательность в контексте структурной организации объектов. Хорошо структурированные взаимодействия, как и хорошо структурированные алгоритмы, должны демонстрировать высокую эффективность, быть понятными и адаптируемыми.

Объекты и роли

Участники взаимодействия могут быть либо конкретными, либо обобщенными сущностями. Как конкретные сущности они выражают вполне определенные объекты. Например, первокурсник, экземпляр класса Студент, может описывать конкретное лицо (Олю Кулик). С другой стороны, как обобщенная сущность первокурсник может представлять любой экземпляр класса Студент.

При разработке моделей детальных требований удобнее использовать обобщенные объекты, поскольку они повышают степень универсальности моделей.

Обобщенный объект называется ролью и обозначается так:

имяРоли:ИмяКласса.

В отличие от имени роли имя конкретного объекта всегда подчеркивается:

имяОбъекта:ИмяКласса.

Пример 9.1. На рис. 9.21, *а* показаны классы Институт и Студент, между которыми существует ассоциация «многие-ко-многим», на рис. 9.21, *б* — пересылка обобщенного сообщения между соответствующими ролями а:Институт и первокурсник:Студент, на рис. 9.21. *в* — пересылка конкретного сообщения между конкретными объектами тси:Институт и оляКулик:Студент (прости нас, Оля, за имя с маленькой буквы, но таковы уж правила именования объектов).

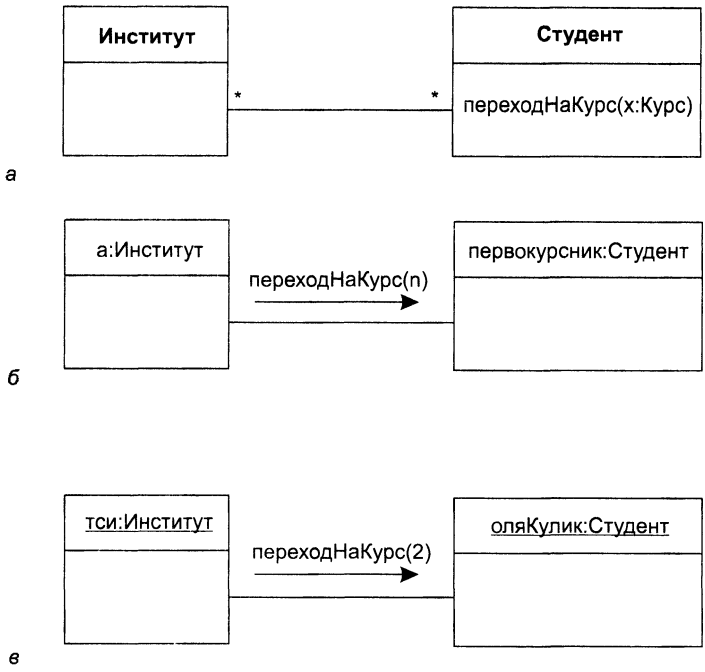


Рис. 9.21. Классы, роли и объекты: *а* — классы; *б* — роли; *в* — конкретные объекты

ПРИМЕЧАНИЕ

Существует тонкое различие между конкретными объектами и ролями. Роль — это не объект, а описание участника в какой-либо среде взаимодействия (контексте). Роль, как и атрибут объекта, соответствует многим возможным значениям. Конкретные объекты применяют в специфических примерах (на диаграммах объектов, компонентов и развертывания). Роли появляются в обобщенных описаниях (на диаграммах взаимодействия и деятельности).

Диаграммы взаимодействия

Диаграммы взаимодействия предназначены для моделирования динамических возможностей системы. Диаграмма взаимодействия показывает взаимодействие, включающее набор участников (ролей, обобщенных объектов) и их отношений, а также пересылаемые между участниками сообщения. Существует две разновидности диаграммы взаимодействия — диаграмма последовательности и диаграмма коммуникации. Диаграмма последовательности — это диаграмма взаимодействия, которая выделяет упорядочение сообщений по времени. Диаграмма коммуникации — это диаграмма взаимодействия, которая выделяет структурную организацию обобщенных объектов, посылающих и принимающих сообщения. Элементами диаграмм взаимодействия являются участники взаимодействия (обобщенные объекты, роли), связи, сообщения.

Диаграммы коммуникации

Диаграммы коммуникации отображают взаимодействие ролей или объектов в процессе функционирования системы. Такие диаграммы моделируют сценарии поведения системы.

Обозначение объекта показано на рис. 9.22.

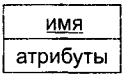


Рис. 9.22. Обозначение объекта

Имя объекта подчеркивается и указывается всегда, атрибуты указываются выборочно. И имя объекта, и имена атрибутов записываются с маленькой буквы.

Синтаксис представления имени имеет вид: имяОбъекта : ИмяКласса

Примеры записи имени.

адам : Человек	Имя объекта и класса
: Пользователь	Только имя класса (анонимный объект)
мойКомпьютер	Только имя объекта (подразумевается, что имя класса известно)
агент :	Объект — сирота (подразумевается, что имя класса неизвестно)

Синтаксис представления атрибута имеет вид: имя : Тип = Значение

Примеры записи атрибута.

номерТелефона = '7350-120'	Имя, тип, значение
активен = True	Имя и значение

Объекты взаимодействуют друг с другом с помощью связей — каналов для передачи сообщений. Связь между парой объектов рассматривается как экземпляр ассоциации между их классами. Иными словами, связь между двумя объектами существует только тогда, когда имеется ассоциация между их классами. Неявно все классы имеют ассоциацию сами с собой, следовательно, объект может послать сообщение самому себе.

Итак, связь – это путь для пересылки сообщения. Сообщение — это спецификация передачи информации между объектами в ожидании того, что будет обеспечена требуемая деятельность. Прием сообщения рассматривается как событие.

Результатом обработки сообщения обычно является действие. В языке UML моделируются следующие разновидности действий.

вызов (call)	В объекте запускается операция
возврат (return)	Возврат значения в вызывающий объект
посылка (send)	В объект посылается сигнал
создание	Создание объекта, выполняется по стандартному сообщению <<create>>
уничтожение	Уничтожение объекта, выполняется по стандартному сообщению <<destroy>>

Для записи сообщений в языке UML принят следующий синтаксис:

имяАтрибута = имяСообщения (Аргументы):ВозвращаемоеЗначение,

где имяАтрибута задает атрибут, куда помещается возвращаемое значение.

Примеры записи сообщений.

координ = текущПоложение(самолетТ1)	Вызов операции, возврат значения
оповещение()	Посылка сигнала
установитьМаршрут(х)	Вызов операции с действительным параметром
<<create>>	Стандартное сообщение для создания объекта

Когда объект посылает сообщение в другой объект (делегируя некоторое действие получателю), объект-получатель, в свою очередь, может послать сообщение в третий объект и т. д. Так формируется поток сообщений — последовательность управления. Очевидно, что сообщения в последовательности должны быть пронумерованы. Номера записываются перед именами сообщений, направления сообщений указываются стрелками (размещаются над линиями связей).

Наиболее общую форму управления задает процедурный поток, иначе называемый потоком с вложениями, — поток синхронных сообщений. Как показано на рис. 9.23, процедурный поток рисуется стрелками с закрашенными наконечниками.

Рис. 9.23. Поток синхронных сообщений

Здесь сообщение 2.1 : напиток = изготовить(Смесь№3) определено как первое сообщение, вложенное во второе сообщение 2 : заказать(Смесь№3) последовательности, а сообщение 2.2 : принести(Напиток) — как второе вложенное сообщение. Все сообщения процедурной последовательности считаются синхронными. Работа с синхронным сообщением подчиняется следующему правилу: передатчик ждет до тех пор, пока получатель не примет и не обработает сообщение. В нашем примере это означает, что третье сообщение будет послано только после обработки сообщений 2.1 и 2.2. Отметим, что степень вложенности сообщений может быть любой. Главное, чтобы соблюдалось правило — последовательность сообщений внешнего уровня возобновляется только после завершения вложенной последовательности.

Менее общую форму управления задает асинхронный поток управления. Как показано на рис. 9.24, асинхронный поток рисуется обычными стрелками. Здесь все сообщения считаются асинхронными, при которых передатчик не ждет реакции от получателя сообщения. Такой вид коммуникации имеет семантику почтового ящика — получатель принимает сообщение по мере готовности. Иными словами,

передатчик и получатель не синхронизируют свою работу, скорее один объект «избавляется» от сообщения для другого объекта. В нашем примере сообщение читатьПисьмо() определено как второе сообщение в потоке управления.

Заметим, что участниками взаимодействий на рис. 9.23–9.24 объявлены роли, то есть обобщенные объекты. Для ссылок на обычные объекты все имена в вершинах нужно было бы подчеркнуть.

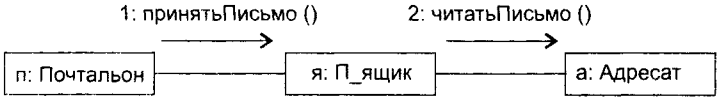


Рис. 9.24. Поток асинхронных сообщений

Помимо рассмотренных линейных потоков управления можно моделировать и более сложные формы — итерации и ветвления.

Итерация представляет повторяющуюся последовательность сообщений. После номера сообщения итерации добавляется выражение:

```
*[i := 1 .. n]
```

Оно означает, что сообщение итерации будет повторяться заданное количество раз. Например, четырехкратное повторение первого сообщения рисоватьСторонуПрямоугольника можно задать выражением:

```
1*[i := 1 .. 4] : рисоватьСторонуПрямоугольника(i)
```

Для моделирования ветвления после номера сообщения добавляется выражение условия, например: [x>0]. Сообщение альтернативной ветви помечается таким же номером, но с другим условием: [x<=0]. Пример итерационного и разветвляющегося потока сообщений приведен на рис. 9.25.

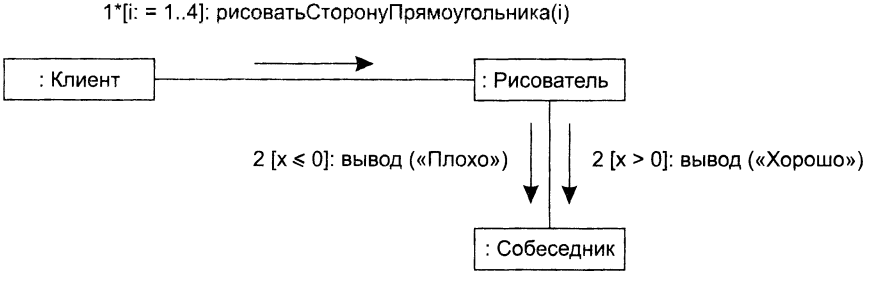


Рис. 9.25. Итерация и ветвление

Здесь первое сообщение повторяется 4 раза, а в качестве второго выбирается одно из двух сообщений (в зависимости от значения переменной x). В итоге обобщенный объект рисователя нарисует на экране прямоугольное окно, а обобщенный объект собеседника выведет в него соответствующее донесение.

Таким образом, для формирования диаграммы коммуникации выполняются следующие действия:

1) отображаются участники взаимодействия;

2) рисуются связи, соединяющие этих участников;

3) связи помечаются сообщениями, которые посылают и получают определенные участники.

В итоге формируется ясное визуальное представление потока управления (в контексте структурной организации сотрудничающих объектов).

Диаграмма коммуникации помещается в рамку. Имя диаграммы указывается вслед за пометкой comm в пятиугольнике, размещаемом в левом верхнем углу рамки. В качестве примера на рис. 9.26 приведена диаграмма коммуникации системы управления полетом летательного аппарата.

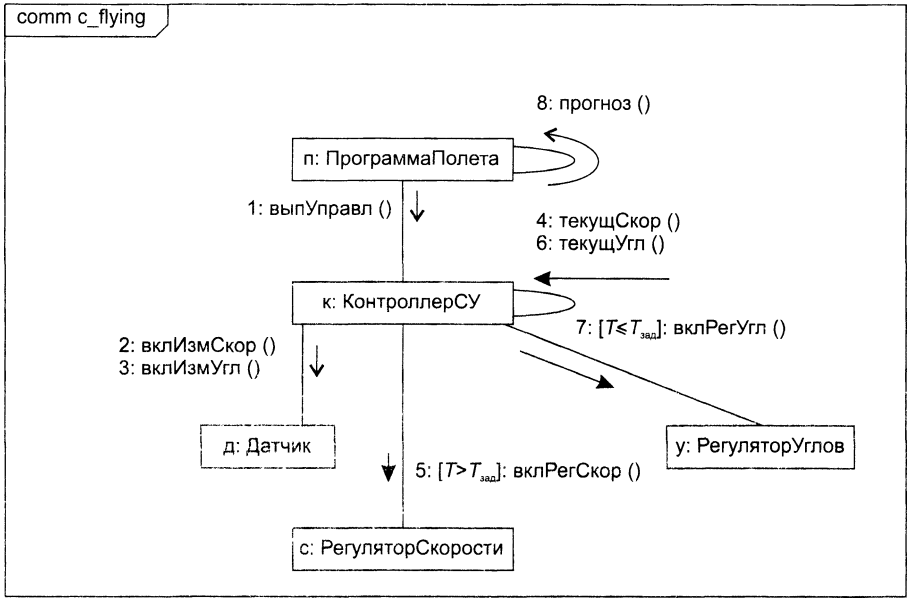


Рис. 9.26. Диаграмма коммуникации системы управления полетом

На данной диаграмме представлены пять обобщенных объектов системы. Поток управления в системе включает восемь сообщений: четыре асинхронных и четыре синхронных сообщения. Обобщенный экземпляр Контроллера СУ ждет приема и обработки сообщений:

- вклРегСкор();
- вклРегУгл();
- текущСкор();
- текущУгл().

Порядок следования сообщений задан их номерами. Для пятого и седьмого сообщений указаны условия:

- включение Регулятора Скорости происходит, если относительное время полета T больше заданного периода $T_{зад}$;
- включение Регулятора Углов обеспечивается, если относительное время полета меньше или равно заданному периоду.

Диаграммы последовательности

Диаграмма последовательности — вторая разновидность диаграмм взаимодействия. Отражая сценарий поведения в системе, эта диаграмма обеспечивает более наглядное представление порядка передачи сообщений. Правда, она не позволяет показать такие детали, которые видны на диаграмме коммуникации (структурные характеристики объектов и связей).

Графически диаграмма последовательности — двумерная схема, которая показывает обобщенные объекты (роли), размещенные вдоль горизонтальной оси, и сообщения, упорядоченные по времени, вдоль вертикальной оси.

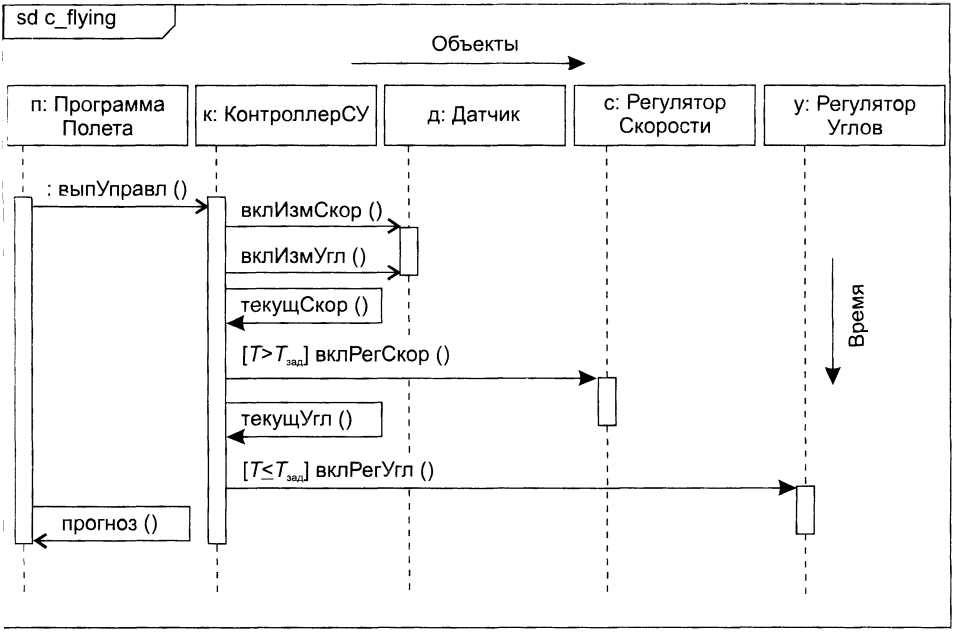


Рис. 9.27. Диаграмма последовательности системы управления полетом

Как показано на рис. 9.27, участники взаимодействия (роли или объекты) помещаются на вершине диаграммы, вдоль горизонтальной оси. Обычно слева размещается участник, инициирующий взаимодействие, а справа — участники по возрастанию подчиненности. Сообщения, посылаемые и принимаемые участниками, показываются вдоль вертикальной оси, в порядке возрастания времени от вершины к основанию диаграммы. Используется тот же синтаксис и обозначения синхронизации, что и в диаграммах коммуникации. Таким образом, обеспечивается простое визуальное представление потока управления во времени.

От диаграмм коммуникации диаграммы последовательности отличают две важные характеристики.

Первая характеристика — *линия жизни* участника взаимодействия.

Линия жизни участника взаимодействия — это вертикальная пунктирная линия, которая обозначает период существования участника взаимодействия.

ПРИМЕЧАНИЕ

Сообщения выглядят на диаграмме как горизонтальные стрелки, идущие от линии жизни одного участника (отправителя) к линии жизни другого участника (получателя). Хотя участник может послать сообщение и самому себе. Если сообщению требуется некоторое время на пересылку, то стрелка сообщения изображается с наклоном вниз — это обозначает, что получение сообщения происходит с задержкой. При этом у обоих концов стрелки могут располагаться метки с указанием времени отправки и получения сообщения.

Большинство обобщенных объектов существуют на протяжении всего взаимодействия, их линии жизни тянутся от вершины до основания диаграммы. Впрочем, обобщенные объекты могут создаваться в ходе взаимодействия. Их линии жизни начинаются с момента приема сообщения <<create>>. Кроме того, обобщенные объекты могут уничтожаться в ходе взаимодействия. Их линии жизни заканчиваются с момента приема сообщения <<destroy>>. Как представлено на рис. 9.28, уничтожение линии жизни отмечается пометкой X в конце линии.

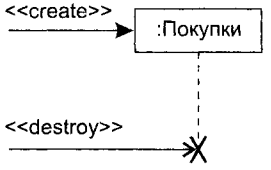


Рис. 9.28. Создание и уничтожение обобщенного объекта

Вторая характеристика — *спецификация выполнения* (execution specification), или *активация* (activation).

Спецификация выполнения (активация) — это высокий, тонкий прямоугольник, отображающий период времени, в течение которого обобщенный объект выполняет действие (свою операцию). Вершина прямоугольника отмечает начало действия (к ней подходит стрелка сообщения, инициирующего это действие), а основание — его завершение. Момент завершения может маркироваться сообщением возврата, которое показывается пунктирной стрелкой (она направлена к линии жизни вызывавшего объекта). В процедурном потоке управления стрелки возврата можно опустить, поскольку их наличие подразумевается, но указание их на диаграмме придает ей большую ясность. Можно показать вложение активации (например, рекурсивный вызов собственной операции). Для этого вторая активация рисуется немного правее первой (рис. 9.29).

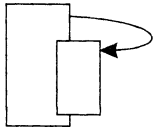


Рис. 9.29. Вложение спецификаций выполнения (активаций)

Конечно, основное назначение диаграммы последовательности — изображать линейный поток управления, но иногда возникает необходимость в использова-

ний более сложных потоков управления. Для этой цели в диаграмму внедряют комбинированные фрагменты. *Комбинированный фрагмент* (combined fragment) состоит из ключевого слова, подмножества линий жизни и одного или нескольких субфрагментов (называемых операндами взаимодействия). Количество и значение субфрагментов зависит от ключевого слова. Фрагмент помещается в рамку.

Наиболее популярны следующие комбинированные фрагменты:

- *Использование взаимодействия* (interaction use) — это ссылка на другое взаимодействие, которое описывается отдельной диаграммой последовательности. Оно помечается ключевым словом ref.
- *Цикл* (ключевое слово loop) — имеет один субфрагмент со сторожевым условием. Сторожевое условие может содержать минимальное и максимальное количество повторений, а также логическое условие. Субфрагмент повторяется до тех пор, пока сторожевое условие истинно, но не менее, чем указанное минимальное количество раз, и не более, чем указанное максимальное количество раз. Если сторожевое условие отсутствует, оно считается истинным, и выполнение цикла полностью определяется указанным количеством повторений.
- *Условный фрагмент* (ключевое слово alt) — включает два или более субфрагмента, каждый из которых имеет сторожевое условие. Субфрагменты отделяются друг от друга при помощи горизонтальных пунктирных линий. Когда поток управления достигает условного фрагмента, выполняется тот из его субфрагментов, сторожевое условие которого является истинным. Если сторожевое условие истинно более чем у одного субфрагмента, выбор одного из таких субфрагментов осуществляется случайным образом. Если ни одно из сторожевых условий не является истинным, то ни один субфрагмент не выполняется.
- *Необязательный фрагмент* (ключевое слово opt) — является частным случаем условного фрагмента. В него входит один субфрагмент, который выполняется в случае, если его сторожевое условие истинно, и не выполняется, если оно ложно.
- *Параллельный фрагмент* (ключевое слово par) — имеет два или более субфрагментов. Когда поток управления достигает параллельного фрагмента, то все его субфрагменты выполняются параллельно. Относительный порядок следования сообщений в параллельных субфрагментах не определен, и порядок выполнения отдельных элементов может быть любым. Когда выполнение всех субфрагментов завершается, поток управления заново сливается воедино.

Кроме того, в языке UML определены еще восемь специальных комбинированных фрагментов.

В качестве примера на рис. 9.30 показана диаграмма последовательности для обработки заказа билетов в филармонию. В эту диаграмму вложено «использование взаимодействия», определяющее параметры заказчика, и фрагмент-цикл, причем в цикл, в свою очередь, вложен условный фрагмент с двумя субфрагментами. В каждой итерации цикла обрабатывается один элемент заказа. В ходе обработки запрашивается база данных билетов. Параметрами запроса являются дата концерта и необходимое количество билетов. При положительном исходе (истинным является сторожевое условие доступно первого условного субфрагмента) в билетах указываются места. При отрицательном исходе (истинным является сторожевое

условие **недоступно** второго условного субфрагмента) в билетах отказывают. Цикл завершается после обработки всех элементов заказа. По итогам обработки со счета клиента снимается определенная сумма.

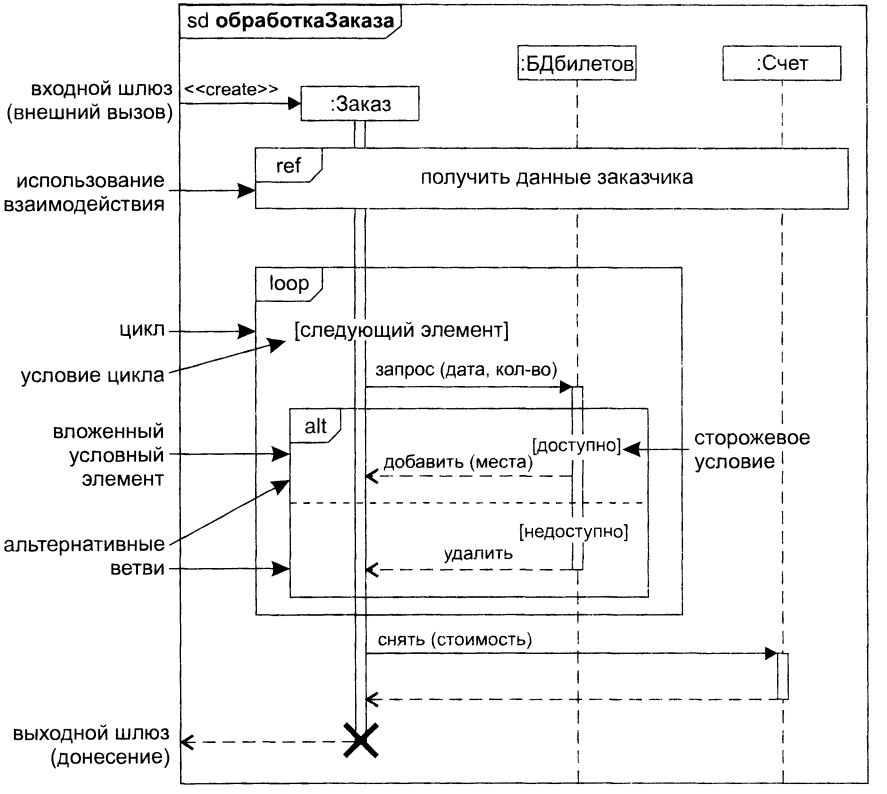


Рис. 9.30. Диаграмма последовательности с вложенными фрагментами

Ограничения на количество итераций фрагмента-цикла указываются в скобках после ключевого слова loop:

```
loop                минимум = 0, максимум неограничен
loop(количество)    минимум = максимум = количество
loop(мин, макс)     явное задание минимальной и максимальной границ.
```

Помимо границ на линии жизни фрагмента может быть записано логическое выражение сторожевого условия. Цикл продолжается, пока условие истинно, но выполняться он будет не менее минимального количества раз и не более максимального, независимо от сторожевого условия.

На рис. 9.31 показан цикл с границами и сторожевым условием.

Диаграмма последовательности позволяет показать состояния модели. Состояние (или условие) изображается символом состояния (прямоугольник со скругленными углами), который помещается на линию жизни. Это состояние (условие) сохраняется в модели до момента наступления следующего состояния

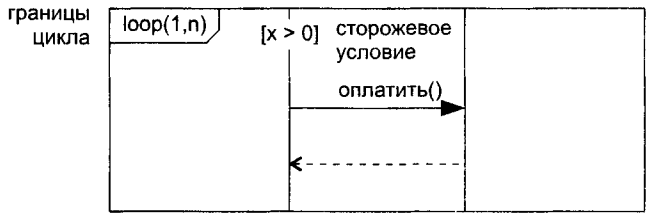


Рис. 9.31. Границы числа итераций

на линии жизни. На рис. 9.32 показаны состояния в жизненном цикле билета в филармонию. Когда линия жизни прерывается символом состояния, состояние объекта меняется. К символу состояния обычно примыкает стрелка сообщения, вызвавшего перемену состояния.

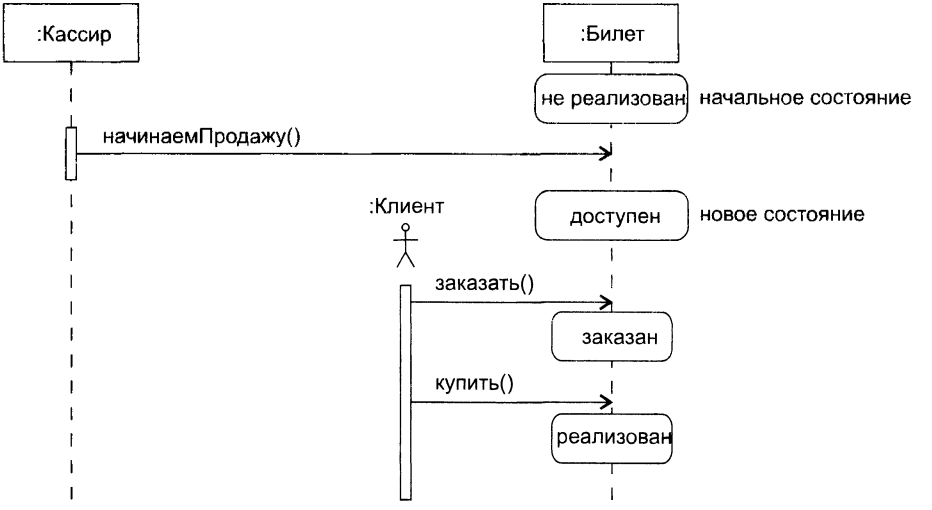


Рис. 9.32. Состояние объекта на диаграмме последовательности

Диаграмма последовательности может взаимодействовать с другими диаграммами. В этом случае в ней предусматриваются входные и выходные шлюзы. Входной шлюз — это точка входа внешних сообщений в диаграмму последовательности (см. рис. 9.30). Соответственно выходной шлюз обеспечивает выдачу сообщений во внешнюю среду.

Диаграммы последовательности считают самым популярным средством представления детальных требований на этапе анализа. Исходными данными для их создания являются вербальные описания требований заказчика, содержащиеся в спецификациях элементов Use Case (сценариях). Эти описания и преобразуются в диаграммы последовательности. Цель преобразования — повысить уровень точности, формализовать требования, привести их к виду, необходимому для решения задач проектирования.

Для построения диаграмм последовательности проводится грамматический разбор каждого сценария элемента Use Case: значащие существительные превращаются в обобщенные объекты и их атрибуты, а значащие глаголы — в сообщения, пересылаемые между объектами.

Моделирование поведения с помощью диаграмм конечных автоматов

При помощи взаимодействий удобно моделировать поведение совокупности совместно работающих объектов. Конечный автомат ориентирован на представление поведения отдельного объекта.

Конечный автомат (State machine) описывает поведение в терминах последовательности состояний, через которые проходит объект в течение своей жизни. Эта последовательность рассматривается как ответ на события и включает реакции на эти события. Автомат задает поведение системы как цельной, единой сущности, он моделирует жизненный цикл единого объекта. В силу этого автоматный подход удобно применять для формализации динамики отдельного, трудного для понимания блока системы.

Диаграмма конечного автомата

Диаграмма конечного автомата отображает конечный автомат, выделяя поток управления, следующий от состояния к состоянию. Конечный автомат — поведение, которое определяет последовательность состояний в ходе существования объекта.

Диаграмма конечного автомата показывает:

1) набор состояний системы;
2) события, которые вызывают переход из одного состояния в другое;
3) действия, которые происходят в результате изменения состояния.

В языке UML состоянием называют период в жизни объекта, на протяжении которого он удовлетворяет какому-то условию, выполняет определенную деятельность или ожидает некоторого события. Как представлено на рис. 9.33, состояние изображается как прямоугольник с закругленными углами, обычно включающий его имя и подсостояния (если они есть).

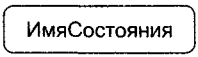


Рис. 9.33. Обозначение состояния

Переходы между состояниями отображаются помеченными стрелками (рис. 9.34)

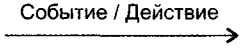


Рис. 9.34. Переходы между состояниями

На рис. 9.34 обозначено: Событие — происшествие, вызывающее изменение состояния, Действие — набор операций, запускаемых событием.

Иначе говоря, события вызывают переходы, а действия являются реакциями на события и переходы.

ПРИМЕЧАНИЕ

Событие, запускающее переход, называют переключающим событием. В языке UML определено, что реакция на события проявляется в виде эффектов. Когда происходит событие, в зависимости от текущего состояния объекта наблюдается некоторый эффект. Эффект задает поведение, реализуемое внутри автомата. В конечном счете, эффекты проявляют себя в выполнении действий, изменяющих состояние объекта или возвращающих какое-либо значение.

Примеры событий.

баланс < 0	Изменение в состоянии
помехи	Сигнал (объект с именем)
уменьшить(Давление)	Вызов действия
after (5 seconds)	Истечение периода времени
when (time = 16:30)	Наступление абсолютного момента времени

Примеры действий.

кассир.прекратитьВыплаты()	Вызов одной операции
flt = new(Фильтр); flt.убратьПомехи()	Вызов двух операций
send ник.привет	Посылка сигнала в объект ник

ПРИМЕЧАНИЕ

Для отображения посылки сигнала используют специальное обозначение — перед именем сигнала указывают служебное слово send.

Для отображения перехода в начальное состояние принято обозначение, показанное на рис. 9.35. Заметим, что по сути начальное состояние является псевдосостоянием, отмечающим точку старта.

Рис. 9.35. Переход в начальное состояние

ПРИМЕЧАНИЕ

Псевдосостояние — это временное состояние, формирующее внутреннюю структуру перехода. При активизации псевдосостояния конечный автомат еще не закончил выполнение непрерывного шага перехода, поэтому не может обрабатывать события. Псевдосостояния используются для связывания фрагментов (сегментов) перехода. Переход к одному из псевдосостояний подразумевает, что дальнейший переход к следующему состоянию будет сделан автоматически, без помощи события.

Соответственно, обозначение перехода в конечное состояние имеет вид, представленный на рис. 9.36. Конечное состояние не является псевдосостоянием. Это особое состояние, которое остается активным после завершения конечным автоматом непрерывного шага перехода.

Рис. 9.36. Переход в конечное состояние

В качестве примера на рис. 9.37 показана диаграмма конечного автомата для системы охранной сигнализации.

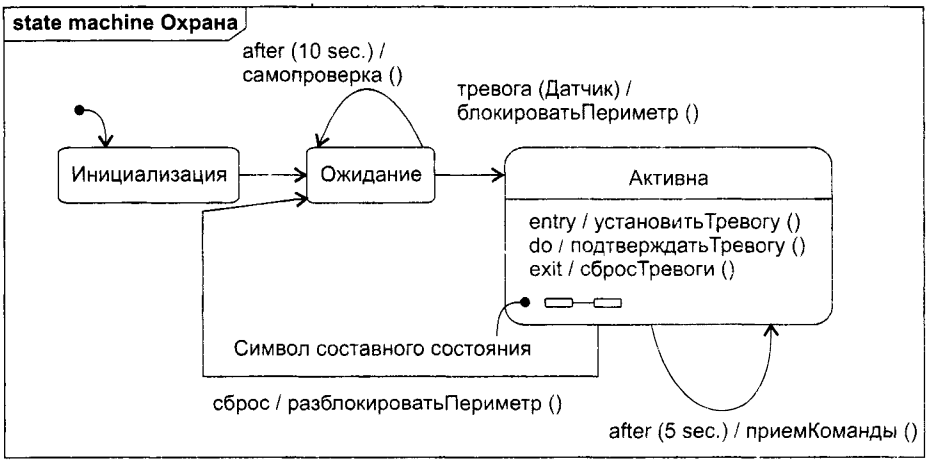


Рис. 9.37. Диаграмма конечного автомата для системы охранной сигнализации

Из рисунка видно, что система начинает свою жизнь в состоянии Инициализация, затем переходит в состояние Ожидание. В этом состоянии через каждые 10 секунд (по событию after(10 sec.)) выполняется самопроверка системы (операция самопроверка()). При наступлении события тревога(Датчик) реализуются действия, связанные с блокировкой периметра охраняемого объекта – исполняется операция блокироватьПериметр(), и осуществляется переход в состояние Активна. В активном состоянии через каждые 5 секунд по событию after(5 sec.) запускается операция приемКоманды(). Если команда получена (наступило событие сброс), система возвращается в состояние Ожидание. В процессе возврата разблокируется периметр охраняемого объекта (операция разблокироватьПериметр()).

Действия в состояниях

Для указания действий, выполняемых при входе в состояние и при выходе из состояния, используются метки entry и exit соответственно.

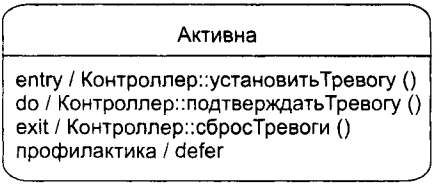


Рис. 9.38. Входные и выходные действия и деятельность в состоянии Активна

Например, как показано на рис. 9.38, при входе в состояние Активна выполняется операция установитьТревогу() из класса Контроллер, а при выходе из состояния — операция сбросТревоги().

Действие, которое должно выполняться, когда система находится в данном состоянии, указывается после метки do. Считается, что такое действие начинается при входе в состояние и заканчивается при выходе из него. Например, в состоянии Активна это действие подтверждатьТревогу().

Возможны события, осуществление которых в данном состоянии должно быть отложено. Такие события остаются отложенными до тех пор, пока система не перейдет в другое состояние, где они уже не будут отложенными. Зарезервированное имя действия defer указывает на то, что событие отложено в данном состоянии и его подсостояниях:

```
имя-события / defer
```

В состоянии Активна (см. рис. 9.38) отложенным является событие профилактика.

Условные переходы

Между состояниями возможны различные типы переходов. Обычно переход инициируется переключающим событием. Допускаются переходы и без событий — переходы по завершению (они запускаются по завершении деятельности в текущем состоянии). Наконец, разрешены условные или охраняемые переходы.

Правила пометки стрелок условных переходов иллюстрирует рис. 9.39.

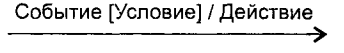


Рис. 9.39. Обозначение условного перехода

Порядок выполнения условного перехода:

1) происходит событие (или завершается деятельность в текущем состоянии);
2) вычисляется сторожевое условие УсловиеПерехода;
3) при УсловиеПерехода=true запускается переход и активизируется действие, в противном случае переход не выполняется.

Пример условного перехода между состояниями Инициализация и Ожидание приведен на рис. 9.40. Он происходит по событию питаниеПодано, но только в том случае, если достигнут боевой режим лазера.

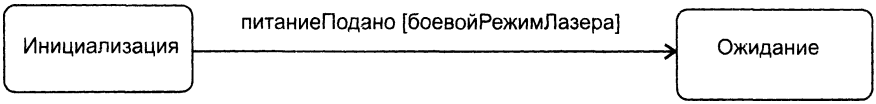


Рис. 9.40. Условный переход между состояниями

Композитные состояния

Одной из наиболее важных характеристик конечных автоматов в UML является подсостояние. Подсостояние позволяет значительно упростить моделирование сложного поведения. Подсостояние — это состояние, вложенное в другое состояние. На рис. 9.41 показано композитное состояние, содержащее в себе два подсостояния.

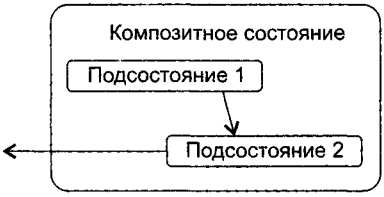


Рис. 9.41. Обозначение подсостояний

На рис. 9.42 приведена внутренняя структура композитного состояния Активна.

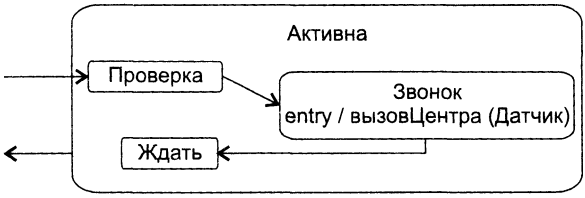


Рис. 9.42. Переходы в состоянии Активна

Семантика вложенности такова: если система находится в состоянии Активна, то она должна быть точно в одном из подсостояний: Проверка, Звонок, Ждать. В свою очередь, в подсостояние могут вкладываться другие подсостояния. Степень вложенности подсостояний не ограничивается. Данная семантика соответствует случаю неортогональных (последовательных) подсостояний.

Иногда при возврате в композитное состояние возникает необходимость попасть в то его подсостояние, которое в прошлый раз было последним. Такое подсостояние называют историческим. Информация об историческом состоянии запоминается. Как показано на рис. 9.43, подобная семантика переходов отображается значком истории — буквой H внутри кружка.

При первом посещении состояния Активна автомат не имеет истории, поэтому происходит простой переход в подсостояние Проверка. Предположим, что в подсостоянии Звонок произошло событие запрос. Средства управления заставляют автомат покинуть подсостояние Звонок (и состояние Активна) и вернуться в состояние

Команды. Когда работа в состоянии Команды завершается, выполняется возврат в историческое подсостояние состояния Активна. Поскольку теперь автомат запомнил историю, он переходит прямо в подсостояние Звонок (минуя подсостояние Проверка).

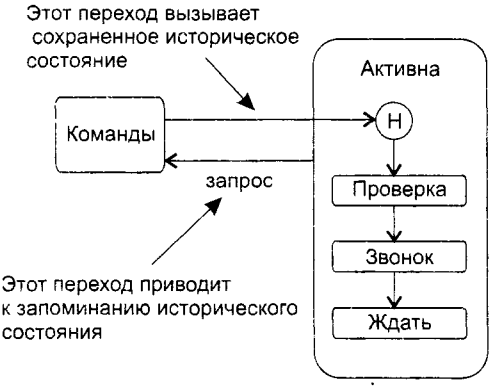


Рис. 9.43. Историческое состояние

Как показано на рис. 9.44, для обозначения композитного состояния, имеющего внутри себя скрытые (не показанные на диаграмме) подсостояния, используется символ «очки».

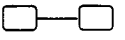


Рис. 9.44. Символ состояния со скрытыми подсостояниями

Возможны также ортогональные (параллельные) подсостояния — они выполняются параллельно внутри композитного состояния. В этом случае композитное состояние разделяется на две или более области. Его области называются ортогональными областями. Если такое состояние активно, то в каждой его ортогональной области активно ровно одно из подсостояний. Степень распараллеливания независимых действий равна количеству ортогональных областей. Графически ортогональные области отделяются друг от друга пунктирными линиями. Секция имени отделяется от областей сплошной линией. Альтернативно можно указать имя композитного состояния на небольшом прямоугольнике, присоединенном к символу этого состояния. В этом случае секция с именем не выглядит как еще одна область.

На рис. 9.45 изображен конечный автомат с композитным состоянием Разработка, у которого есть три ортогональные области. Каждая из них, в свою очередь, делится на подсостояния. Кроме этого, имеются простые состояния Завершен и Остановлен. Автомат моделирует процесс выполнения программного проекта.

При переходе в композитное состояние Разработка становятся активными начальные подсостояния каждой области. Прохождение по каждой из этих трех ортогональных областей осуществляется параллельно. Если одна ортогональная область переходит в конечное состояние раньше других, управление в ней ожидает, пока

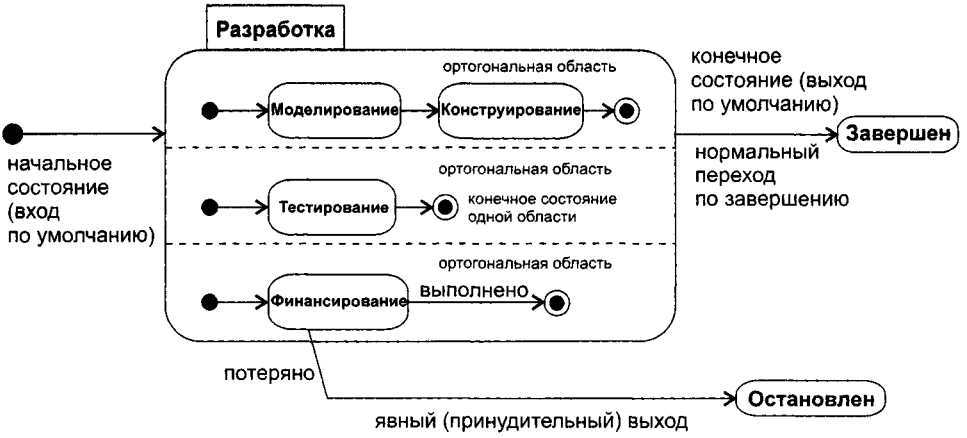


Рис. 9.45. Конечный автомат с ортогональным композитным состоянием

другие области не достигнут своего конечного состояния. Когда все три подобласти достигают своих конечных состояний, управление из них сливается обратно в общий поток, и для внешнего композитного состояния Разработка запускается переход по завершению, после чего активным становится состояние Завершен. В случае если в тот период, когда активно состояние Разработка, происходит событие потеряно (потеряно финансирование), все три параллельные области прекращаются, и активным становится состояние Остановлен.

Когда имеется переход в композитное состояние с ортогональными областями, управление всегда разделяется на столько же параллельных потоков, сколько существует таких областей. Аналогично, при наличии перехода из композитного состояния с ортогональными областями параллельные потоки управления сливаются воедино. Это происходит всегда. Если все ортогональные области достигают своего конечного состояния или же появился явный переход из объемлющего композитного состояния, управление сливается в один поток.

Псевдосостояния управления

В диаграммах конечных автоматов для управления потоком переходов применяются фактически те же узлы управления, что и в диграммах деятельности:

- *выбор* (choice — ромбик с одной входящей и несколькими исходящими стрелками);
- *разделение* (fork — жирная горизонтальная линия с одной входящей и несколькими исходящими стрелками);
- *слияние* (join — жирная горизонтальная линия с несколькими входящими и одной исходящей стрелкой).

В конечном автомате эти узлы образуют псевдосостояния. Псевдосостояние «выбор» позволяет отобразить разветвление переходов, исходящие из него стрелки помечаются сторожевыми условиями ветвления. В зависимости от значения условия псевдосостояние обеспечивает выбор одного из многих переходов.

Псевдосостояния «разделение» и «слияние» позволяют показать параллельные потоки подсостояний, отмечая точки их синхронизации при запуске (момент разделения) и при завершении (момент слияния).

На рис. 9.46 представлен вариант предыдущего примера с явными переходами разделения и слияния.

Разделение обеспечивает переход от одного состояния Начало к трем ортогональным состояниям М, Т и Ф сразу. Графически это выражает толстая черная линия с одной входящей стрелкой и тремя исходящими, каждая из которых указывает на ортогональное состояние.

В свою очередь, слияние реализует переход от трех ортогональных состояний К, Т и Ф к единственному состоянию Завершен. Здесь три входящие стрелки и одна исходящая. Слияние происходит, если завершены все три ортогональных состояния.

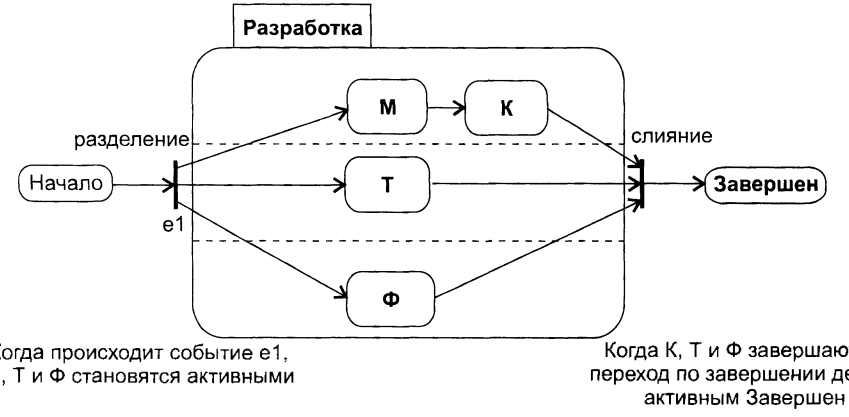


Рис. 9.46. Разделение и слияние

ПРИМЕЧАНИЕ

Не обязательно, чтобы механизм разделения-слияния охватывал все ортогональные области композитного состояния. Например, какая-то область может стартовать или финализироваться автономно.

Еще одним мощным средством управления переходами является псевдосостояние «переходное состояние» (junction).

Переходное состояние используется для создания общего перехода из нескольких сегментов. Оно может иметь один или несколько сегментов входящего и исходящего переходов. Все сегменты, следующие за первым, не имеют переключающего события. Переключающее событие может быть только у первого из этих сегментов, а сторожевые условия у всех. Фактическое сторожевое условие всего перехода образуется логическим умножением сторожевых условий его сегментов.

Одно переходное состояние может покрывать несколько переходов, у каждого из которых свое собственное переключающее событие. Каждый маршрут, проходящий через множество переходных сегментов, является отдельным переходом.

К входящему и исходящему переходу может также прикрепляться действие. Исходящий переход запустится немедленно после запуска входящего перехода. При этом будет выполняться прикрепленное к нему действие. Выполнение входящего и исходящего переходов представляет собой атомарный непрерывный шаг.

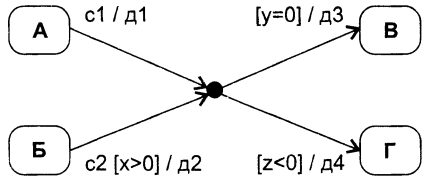
Переходное состояние изображается в виде маленького черного кружка, к которому и от которого ведут стрелки входящих и исходящих переходов. Имени у переходного состояния нет.

На рис. 9.47 показан пример переходного состояния. Два входных сегмента и два выходных сегмента позволяют создать четыре полных перехода между состояниями А, Б, В, и Г.

Запись маршрута Б c2 [x>0 and y=0] / д2; д3 В означает следующее:

- Переход выполняется из состояния Б в состояние В.
- Переключающим событием перехода является событие c2 нижнего входящего перехода.
- Сторожевое условие x>0 and y=0 образуется конъюнкцией сторожевого условия x>0 нижнего входящего перехода и сторожевого условия y=0 верхнего исходящего перехода.
- При переходе выполняются действие д2 нижнего входящего перехода и действие д3 верхнего исходящего перехода.

В каждом маршруте происходит проверка всех сторожевых условий перед запуском перехода или выполнением каких-либо действий. Если значения сторожевых условий изменяются в процессе выполнения действий, никакого влияния на переходы это не оказывает.



Маршруты полных переходов:

А c1 [y=0] / д1; д3 В
А c1 [z<0] / д1; д4 Г
Б c2 [x>0 and y=0] / д2; д3 В
Б c2 [x>0 and z<0] / д2; д4 Г

Рис. 9.47. Переходное псевдосостояние с несколькими маршрутами

Часто бывает удобно определять именованные точки входа и выхода автомата, соединенные с его внутренними состояниями. На рис. 9.48 приведен конечный автомат Авторизация.

Автомат выполняет идентификацию клиента. Он имеет стандартный вход и стандартный выход. В стандартном режиме данные пользователя считываются с кредитной карты. Возможен и необычный, ручной ввод данных. В этом режиме пользователь вводит свои данные вручную. Для ручного режима задан свой вход — именованная точка ручной набор (обозначается на границе автомата небольшим

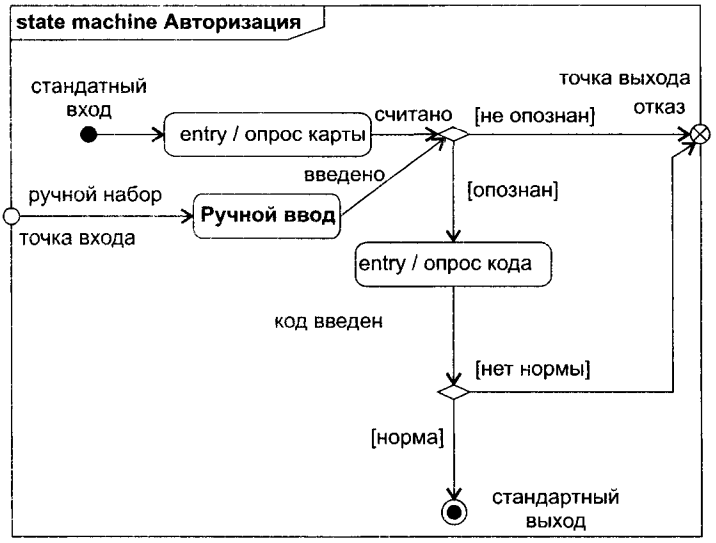


Рис. 9.48. Конечный автомат с именованной точкой входа и выхода

кружком), которая напрямую соединяется с композитным состоянием Ручной ввод. При успешной авторизации автомат прекращает работу в своем конечном состоянии, на стандартном выходе. В противном случае он переходит в состояние, соединенное с именованной точкой выхода отказ. Точка выхода изображается на границе автомата небольшим кружком с крестиком.

Применение диаграмм конечных автоматов

Обычно диаграммы конечных автоматов применяются для моделирования поведения событийно-управляемых объектов. Событийно-управляемым назовем объект, поведение которого определяется его реакцией на внешние события. Говорят, что этот объект управляется событиями. Его работа укладывается в следующий сценарий:

1. Ожидание наступления внешнего события.
2. При появлении события объект реагирует на него. Реакция зависит от его состояния, определяемого прошлыми событиями.
3. Возвращение в состояние ожидания события, то есть переход к пункту 1.

Сценарий применяется на протяжении всего жизненного цикла объекта.

Очевидно, что этому определению соответствуют экземпляры классов, элементы Use Case, да и программные системы в целом. Главное, чтобы объект моделирования управлялся событиями и рассматривался как единое целое.

При моделировании поведения событийно-управляемого объекта всегда определяют следующие моменты:

- формируют состояния объекта (устанавливают начальное и конечное состояния, остальные состояния упорядочивают по времени так, чтобы они покрыли весь жизненный цикл объекта);

- устанавливают перечень событий, вызывающих смену состояний;
- обдумывают действия, выполняемые при смене состояний;
- анализируют возможности упрощения автомата (за счет композитных состояний и подсостояний, применения механизмов выбора, разделения, слияния, переходных и исторических состояний).

ПРИМЕЧАНИЕ

Понятие конечного автомата UML базируется на понятиях абстрактных конечных автоматов Мура и Мили. Напомним, что в терминах UML автомат Мили описывается зависимостями: состояние(t) = f(состояние(t – 1), событие(t)), действие(t) = φ(состояние(t – 1), событие (t)). Автомат Мура реализует зависимости: состояние(t) = f(состояние(t – 1), событие(t)), действие(t) = φ(состояние(t)). То есть в автомате Мили все действия привязаны к состояниям и событиям, а в автомате Мура — только к состояниям. Доказано, что автомат Мура является частным случаем автомата Мили. Диаграммы автоматов UML используют комбинацию автоматов Мура и Мили.

Контрольные вопросы и упражнения

1. Из каких элементов состоит диаграмма Use Case?
2. Какие отношения разрешены между элементами диаграммы Use Case?
3. Для чего применяют диаграммы Use Case?
4. Чем отличаются друг от друга отношения включения и расширения с точки зрения управления?
5. Каково назначение спецификации элемента Use Case и как она оформляется?
6. Что такое сценарий элемента Use Case?
7. Как документируется отношение включения?
8. Как документируется отношение расширения?
9. Каков порядок построения модели требований?
10. С помощью диаграммы Use Case создайте модель желаемого поведения системы управления летательного аппарата. В качестве исходных данных используйте требования заказчика из примера 4.1 четвертой главы.
11. С помощью диаграммы Use Case создайте модель желаемого поведения системы управления роботом, описанной в примере 5.2 пятой главы.
12. Усовершенствуйте модель Банкомата, описанную в данной главе. Добавьте возможность внесения денег в банкомат.
13. Поясните два подхода к моделированию поведения системы. Объясните достоинства и недостатки каждого из этих подходов.
14. Охарактеризуйте средства и возможности диаграммы деятельности.
15. Когда не следует применять диаграмму деятельности?

16. Какие средства диаграммы деятельности позволяют отобразить параллельные действия?
17. Зачем в диаграмму деятельности введены плавательные дорожки?
18. Как представляется имя обычного объекта, обобщенного объекта и роли в диаграмме коммуникации? Чем они отличаются друг от друга с точки зрения содержания?
19. Поясните синтаксис представления атрибута в диаграмме коммуникации.
20. В какой форме записываются сообщения в языке UML? Поясните смысл сообщения.
21. Как связаны сообщения и действия? Перечислите разновидности действий.
22. Чем отличается процедурный поток от асинхронного потока сообщений?
23. Как указывается повторение сообщений на диаграмме коммуникации?
24. Как показать ветвление сообщений на диаграмме коммуникации?
25. Что общего в диаграмме последовательности и диаграмме коммуникации? Чем они отличаются друг от друга?
26. Как отображается порядок передачи сообщений в диаграмме последовательности?
27. Когда удобнее применять диаграммы последовательности?
28. Какое назначение имеют комбинированные фрагменты в диаграмме последовательности? Какие комбинированные фрагменты вы знаете? Приведите примеры диаграмм с их использованием.
29. Составьте диаграммы последовательности для диаграммы Use Case банкомата, описанной в данной главе.
30. Даны три объекта:
    - Объект класса A (включает атрибуты pa1, pa2 и операции opA1(), opA2 ());
    - Объект класса B (включает атрибут pb1 и операции opB1(), opB2 ());
    - Объект класса C (включает атрибуты pc1, pc2 и операции opC1(), opC2 ()).

    Первым начинает работу объект класса A. Он выполняет операцию opA1(). Из операции opA1() вызывается операция opB1() объекта класса B. В свою очередь, из операции opB1() вызывается операция opC1() объекта класса C. Нарисовать диаграмму коммуникации.
31. Даны три объекта:
    - Объект класса A (включает операции opA1(), opA2 ());
    - Объект класса B (включает операцию opB1());
    - Объект класса C (включает операцию opC1()).

    Первым начинает работу объект класса A. Он выполняет операцию opA1(). Из операции opA1() вызывается операция opB1() объекта класса B. В свою очередь, из операции opB1() вызывается операция opC1() объекта класса C. Далее из операции opC1() вызывается операция opA2() объекта класса A. Нарисовать диаграмму последовательности.

32. Охарактеризуйте вершины и дуги диаграммы конечного автомата. В чем состоит назначение этой диаграммы?
33. Как отображаются действия в состояниях диаграммы конечного автомата?
34. Как показываются условные переходы между состояниями?
35. Как задаются вложенные состояния в диаграмме конечного автомата?
36. Поясните понятие исторического подсостояния.
37. Поясните понятие ортогонального композитного состояния.
38. Что такое псевдосостояние конечного автомата? Какие псевдосостояния управления вы знаете? Приведите примеры их применения.

Глава 10

Объектно-ориентированное проектирование и реализация

Десятая глава освещает широкий круг вопросов проектирования в объектно-ориентированном стиле: принципы детального проектирования, инструментарий языка UML, поддерживающий архитектурное и детальное проектирование; паттерны проектирования как надежное средство обеспечения быстрых и качественных решений; рекомендации по проведению паттерн-ориентированного проектирования. Здесь иллюстрируются средства и возможности промышленного подхода к компонентной упаковке программного кода от фирмы Microsoft, обсуждаются базовые идеи аспектно-ориентированного подхода, поясняются принципы проектирования пользовательского интерфейса и метрики практичности интерфейса, вопросы размещения проектных артефактов на аппаратных средствах компьютерных систем.

Архитектурное проектирование

Формирование архитектуры — первый и основополагающий шаг в решении задачи проектирования, закладывающий фундамент представления программной системы, способной выполнять весь спектр детальных требований.

Основные понятия архитектуры мы начали обсуждать в первой главе. В шестой главе это обсуждение было продолжено. Здесь мы сконцентрировали внимание на структурной организации обобщенных архитектурных решений и вопросах управления структурой.

Изложенный материал доказывал, что обобщенно архитектура индифферентна к специфике детального проектирования, то есть к формированию структуры подсистем, определению модульной организации. В общем случае это справедливо. Но если переходить в практическую плоскость, не разумно полностью игнорировать влияние техники декомпозиции на архитектурные решения.

Например, язык UML поддерживает технику объектно-ориентированной декомпозиции и включает специальные средства для записи структур архитектурного уровня, которые гармонизированы как со средствами для фиксации требований,

так и со средствами для обеспечения детального проектирования. Рассмотрим эти средства.

Диаграммы пакетов

Диаграмма пакетов — это структурная диаграмма, в которой основными элементами являются пакеты и зависимости между ними.

Пакет — хранилище элементов, изображается в виде прямоугольника с закладкой в одном из углов (обычно в левом верхнем). Если содержимое пакета не показывается, имя пакета указывают внутри прямоугольника (рис. 10.1).

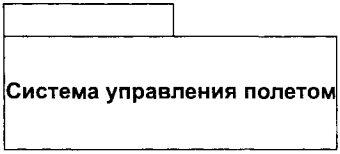


Рис. 10.1. Именование пакета со скрытым содержимым

При отображении содержимого пакета в прямоугольнике имя пакета указывают внутри закладки (рис. 10.2).

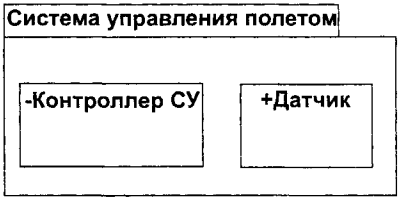


Рис. 10.2. Именование пакета с отображением его содержания

Содержимое пакета можно показать и иным способом, с помощью отношения владения (рис. 10.3). Отношение владения (containment) обозначают линией с небольшим крестиком в кружке, примыкающем к контуру пакета. В данном случае пакет Система управления полетом владеет двумя классами: Контроллер СУ и Датчик.

Рис. 10.3. Внешнее обозначение содержимого пакета

Пакет является средством группировки, внутри пакета могут находиться вложенные пакеты, классы и т. д. Пакет задает видимость находящихся в нем элементов. Видимость характеризует доступность элементов (или их содержимого) другим элементам модели. Видимость элементов пакета может быть определена как приватная или публичная. Публичные элементы доступны другим элементам пакета-владельца или вложенных в него пакетов, а также пакетам, импортирующим данный пакет. Приватные элементы недоступны вне пакета-владельца.

Видимость элемента пакета указывается с помощью символа видимости, который ставится перед именем элемента:

- Плюс (+) обозначает публичную видимость.
- Минус (–) соответствует приватной видимости.

Например, на рис. 10.2 класс Контроллер СУ является приватным (к нему нет доступа вне пакета Система управления полетом), а класс Датчик — публичным элементом (он доступен для всех).

ПРИМЕЧАНИЕ

Содержимое таких элементов, как атрибуты или операции классов, может также иметь защищенную видимость или пакетную видимость. Защищенные элементы доступны только потомкам класса-владельца. Элементы с пакетной видимостью доступны всем элементам того же пакета.

Возможность доступа к какому-то элементу изображается с помощью отношения зависимости. Это отношение показывает, что некий элемент зависит от другого элемента.

Зависимости между элементами пакетов и пакетами показываются с помощью пунктирных стрелок. Хвост стрелки примыкает к зависимому элементу (клиенту), а острие указывает на элемент, от которого зависят (сервер) (рис. 10.4).

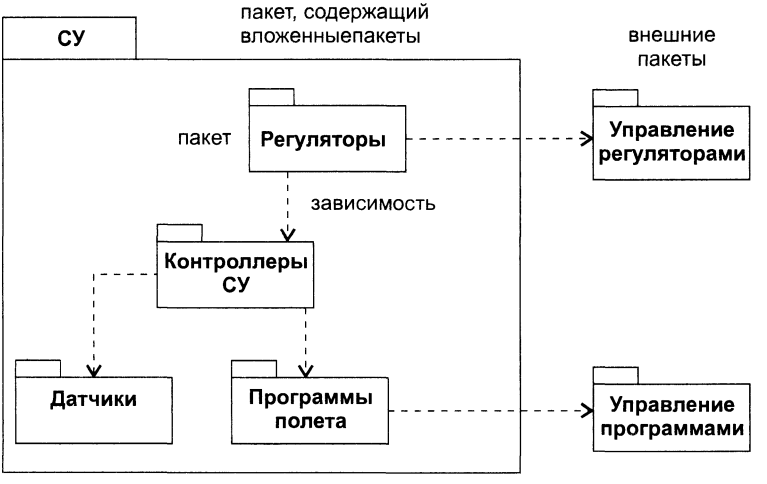


Рис. 10.4. Зависимости между пакетами системы управления летательным аппаратом

ПРИМЕЧАНИЕ

Диаграмма пакетов может помещаться в рамку с пятиугольником в левом верхнем углу. Имя диаграммы записывается вслед за пометкой package.

Имена элементов в каждом пакете образуют собственное пространство имен, в пределах которого работает схема прямой адресации, по простым адресам. На элементы из других пакетов можно ссылаться по их квалифицированным именам, имеющим формат имя пакета: :имя элемента.

Для упрощения ссылок на чужие имена следует применять механизм импортирования. Пакет (клиент), импортирующий элемент из другого пакета (поставщика), обращается к импортируемым элементам по их простым именам. Импортировать можно только публичные (видимые) элементы. Приватные элементы пакетов импортировать в другие пакеты нельзя.

Возможна как приватная, так и публичная видимость импорта внутри импортирующего пакета. Если она публична, импортированный элемент виден всем элементам, которым виден весь импортирующий пакет в целом. Если же выбрана приватная видимость, импортированный элемент не будет виден извне импортирующего пакета.

В общем случае работают следующие правила видимости элементов:

- Определенный в пакете элемент виден внутри этого пакета.
- Если элемент виден внутри пакета, он виден и всем пакетам, вложенным в данный.
- Если пакет импортирует другой пакет с публичными элементами, то все элементы импортированного пакета оказываются видны внутри импортирующего пакета.
- Пакет не видит элементы собственных вложенных пакетов, пока он не импортирует их (при условии, что эти элементы публичны).

Зависимость импортирования отображается пунктирной стрелкой, хвост которой находится на пакете-клиенте, а острие — на пакете-поставщике. В качестве метки на стрелке ставится стереотип «import» при публичном импорте и стереотип «access» — при приватном импорте.

Рассмотрим пример публичного импорта пакетов (рис. 10.5). Пакет П1 импортирует пакет П2. Классы К1 и К2 пакета П1 могут применить простое имя К3 для обращения к публичному классу К3 пакета П2, но приватный класс К4 для них невидим. Классы К3 и К4 пакета П2 могут использовать квалифицированное имя П1::К1 для обращения к публичному классу К1 пакета П1, а приватный класс К2 из пакета П2 невидим.

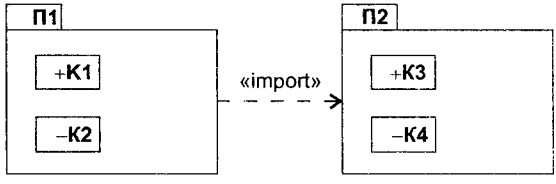


Рис. 10.5. Публичный импорт пакетов

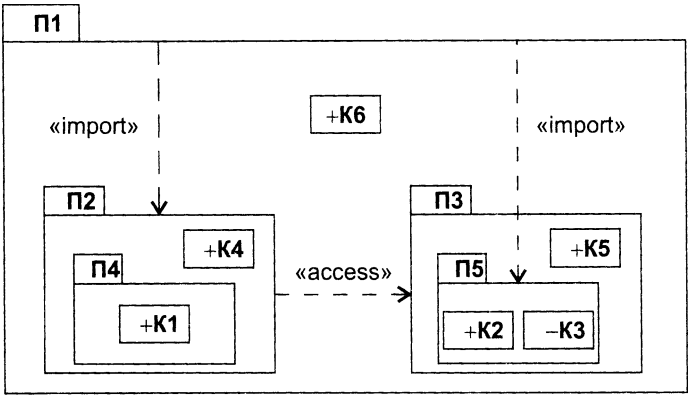


Рис. 10.6. Приватный импорт пакетов

На рис. 10.6 представлены четыре уровня вложенности: на первом (внешнем) уровне находятся пакет П1 и публичный класс К6, на втором — пакеты П2 и П3, на третьем уровне — пакеты П4, П5 и публичные классы К4, К5, на четвертом (внутреннем) уровне — публичные классы К1, К2 и приватный класс К3. Пакет П1 использует публичное импортирование вложенного пакета П2 и пакета П5, вложенного в пакет П3. Пакет П2 применяет приватное импортирование пакета П3. Подробное обсуждение видимости классов для этой структуры приведено в табл. 10.1.

Таблица 10.1. Видимость классов в иерархической структуре пакетов на рис. 10.6

Характеристика видимости классов
Класс К1 видит классы К4 и К6 и использует их простые имена, потому что они находятся внутри пакетов П2 и П1
Классы К1 и К4 видят класс К5 и могут обращаться к нему по простому имени, потому что пакет П2 импортирует пакет П3. Класс К1 вложен в пакет П4, который вложен в пакет П2, а потому видит все, что видит П2
Классы К1, К4 и К6 видят К2 и обращаются к нему по простому имени, потому что они вложены в пакет П1, импортирующий пакет П5 (содержащий К2). Они не видят К3, потому что тот приватен внутри П5. Поэтому класс К3 невидим извне пакета П5
Класс К6 видит К5, но должен обращаться к нему по квалифицированному имени П3::К5, потому что К5 находится в пакете П3, который не был импортирован пакетом П1. Импорт пакета П3 пакетом П2 выполняется с приватной видимостью, а потому импорт пакета П2 пакетом П1 не дает последнему видеть элементы пакета П3
Класс К4 и класс К6 видят К1, но должны обращаться к нему по квалифицированному имени П4::К1, потому что класс К1 находится в пакете П4, который никем не импортировался. Пакет П2 импортировался пакетом П1, поэтому классу К6 не требуется включать в квалифицированное имя пакет П2
Классы К2 и К3 видят классы К5 и К6 и обращаются к ним по простым именам, потому что те классы находятся в пакетах П3 и П1, содержащих их пакет П5. Они также видят класс К4 и обращаются к нему по простому имени, потому что К4 находится в пакете П2, который импортируется пакетом П1, содержащим К2 и К3. Класс К3 приватен, но это не мешает ему видеть другие классы (хотя они, конечно, не могут видеть его)
Классы К2 и К3 видят друг друга и обращаются напрямую, потому что они находятся в одном пакете. Класс К3 приватен для классов из внешних пакетов, но не для классов собственного пакета

Механизм импорта может быть задан как на уровне пакета, так и на уровне отдельных элементов пакета.

Например, в случае, показанном на рис. 10.7, в пакет П1 импортируются оба класса К3 и К4 пакета П2.

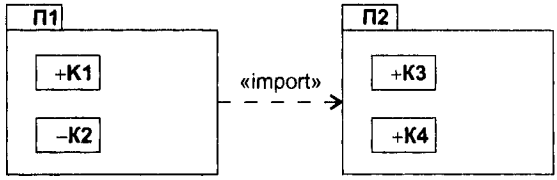


Рис. 10.7. Публичный импорт на уровне пакета

В свою очередь, на рис. 10.8 представлен вариант, когда в пакет П1 импортируется только один класс К3 из пакета П2. В данном варианте к классу К4 придется обращаться по квалифицированному имени П2::К4.

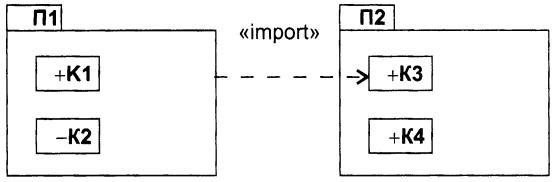


Рис. 10.8. Публичный импорт на уровне элемента пакета

Конечно, пакеты как средства группировки родственных элементов можно применять на многих этапах разработки ПО. Главное, чтобы пакеты обладали высоким уровнем связности внутренних элементов и были слабо сцеплены друг с другом. В контексте архитектурного проектирования мы полагаем пакет удобным средством упаковки подсистемы, которая является частью архитектурной организации, а диаграмму пакетов — моделью структуры архитектурного уровня.

Диаграммы компонентов

Еще одним строительным блоком для создания архитектуры объектно-ориентированной системы считается компонент. Диаграмма компонентов показывает определения, внутреннюю структуру и зависимости набора компонентов.

Компоненты

Компонент — модульная и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов. Реализация компонента всегда скрыта. Компоненты системы с одинаковыми наборами интерфейсов являются взаимозаменяемыми.

Интерфейс — очень важная часть понятия «компонент», его мы обсудим в следующем подразделе.

Графически компонент изображается как прямоугольник со стереотипом «component». Вместо указания стереотипа (или в дополнение к нему) можно поместить

в правом верхнем углу прямоугольника пиктограмму компонента. Эта пиктограмма имеет вид прямоугольника, в одну из сторон которого врезаны два прямоугольника поменьше. Имя компонента указывается во внешнем прямоугольнике (рис. 10.9). Компонент, подобно классу или типу, является обобщенным описанием, поэтому существуют экземпляры компонента. Имя экземпляра компонента подчеркивается, формат его имени похож на формат имени для объекта.

Рис. 10.9. Обозначения компонента

Интерфейсы

Интерфейс – список операций, которые определяют услуги компонента (или класса). Образно говоря, интерфейс — это разъем, который торчит из ящичка компонента. С помощью интерфейсных разъемов компоненты стыкуются друг с другом, объединяясь в систему.

Еще одна аналогия. Интерфейс подобен абстрактному классу, у которого отсутствуют атрибуты и работающие операции, а есть только абстрактные операции (не имеющие тел). Если хотите, интерфейс похож на улыбку чеширского кота из правдивой истории об Алисе, где кот отдельно и улыбка отдельно. Все операции интерфейса открыты и видимы клиенту (в противном случае они потеряли бы всякий смысл). Итак, операции интерфейса только именуют услуги, не более того.

Интерфейсы могут быть двух видов: обеспеченные и требуемые.

Обеспеченный интерфейс описывает услуги, исполнение которых компонент предлагает (другим компонентам). Каждой операции такого интерфейса должен соответствовать элемент реализации внутри компонента. Иными словами, обеспеченный интерфейс именует набор услуг, предоставляемых компонентом (или классом).

Требуемый интерфейс описывает услуги, которые поставляются другими компонентами. Требуемый интерфейс декларирует, что данному компоненту (или классу) для работы требуется прибегать к услугам другого элемента, предоставляющего этот интерфейс.

В прямоугольнике компонента можно показать секцию со списком обеспеченных интерфейсов (начинается со стереотипа «provided») и списком требуемых интерфейсов (начинается со стереотипа «required»). Пример представлен на рис. 10.10.

Рис. 10.10. Описание интерфейсов в прямоугольнике компонента

Чаще всего интерфейсы изображаются в форме пиктограмм, соединенных линиями с границей прямоугольника компонента. Обеспеченный интерфейс рисуется как кружок, а требуемый интерфейс — как полуокружность (рис. 10.11).

Рис. 10.11. Представление интерфейса в форме пиктограммы

Развернутый способ представления интерфейса иллюстрирует рис. 10.12. Здесь интерфейс изображается в виде прямоугольника со стереотипом «interface». В прямоугольнике интерфейса показываются его операции. Компонент, который реализует обеспеченный интерфейс, подключается к нему отношением реализации. Компонент, который получает доступ к услугам другого компонента через требуемый интерфейс, подключается к интерфейсу отношением зависимости со стереотипом «use».

Рис. 10.12. Развернутая форма представления интерфейса

У одного компонента может быть несколько требуемых и несколько обеспеченных интерфейсов.

Тот факт, что между двумя компонентами всегда находится интерфейс, устраняет их прямую зависимость. Компонент, использующий обеспеченный интерфейс, будет функционировать правильно вне зависимости от того, какой компонент реализует этот интерфейс. Это очень важно и обеспечивает гибкую замену компонентов в интересах развития системы.

Порты компонентов

Порт является окном в капсулу компонента. Через это окно происходит взаимодействие внешней среды с закрытым содержимым компонента.

Порт изображается как маленький квадрат на границе символа компонента (рис. 10.13). К символу порта может быть присоединен как обеспеченный, так и требуемый интерфейс. Соединение показывается сплошной линией.

Обеспеченный интерфейс изображает услуги, которые внешний клиент может запросить через данный порт, а требуемый интерфейс — услуги, получаемые от какого-либо другого компонента через порт. С одним портом могут быть связаны несколько интерфейсов.

Каждый порт может иметь имя, задающее его индивидуальность. Имя компонента вместе с именем порта поддерживает адресное взаимодействие компонентов.

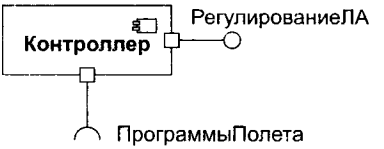


Рис. 10.13. Порты компонента

ПРИМЕЧАНИЕ

С помощью портов моделируется стандартный способ коммуникации между компонентами. В дальнейшем, на этапе программирования, порты могут поддерживаться программным кодом.

Внутренняя структура компонента

Крупный компонент создается из малых компонентов (субкомпонентов), которые применяют в качестве строительных блоков. Внутренняя структура компонента содержит соединенные между собой части, формирующие его реализацию.

Часть — это единица реализации компонента. Часть является обобщенной сущностью, описывающей какую-то роль в компоненте. Часть характеризуется именем, типом и множественностью, записываемыми в формате:

```
Имя_роли: Тип [множественность]
```

Если множественность части равна единице, она опускается.

Примеры представления ролей (частей).

сотрудник:Организация	Именованная роль, множественность равна 1
сотрудник:Организация[*]	Именованная роль, множественность равна «много»
:Организация	Анонимная роль, множественность равна 1
сотрудник	Именованная роль, множественность равна 1

Когда части являются субкомпонентами с портами, их можно соединить между собой через эти порты[1]. Два порта можно соединить друг с другом, если один из них обеспечивает какой-то интерфейс, а другой требует его. Подключение портов подразумевает, что для получения сервиса требующий порт вызовет обеспечивающий. Главное здесь — совместимость интерфейсов. Все остальное неважно. Программный «провод» между двумя портами называют соединителем.

Различают сборочные и делегирующие соединители.

Сборочные соединители связывают между собой порты и интерфейсы различных субкомпонентов. Сборочные соединители обеспечивают сборку из субкомпонентов крупного компонента.

[1] Забегая вперед, отметим, что это справедливо и для классов. Классы тоже могут иметь порты.

ПРИМЕЧАНИЕ

Возможны два варианта сборочных соединителей: прямые соединители и соединители «шарик-и-гнездо». Первый вариант — это простая линия между квадратиками двух портов. Второй вариант выглядит как «вложение» кружка, изображающего обеспеченный интерфейс, в полуокружность требуемого интерфейса.

Делегирующий соединитель связывает внешний порт компонента с портом одного из его внутренних субкомпонентов. Делегирующие соединители поддерживают реализацию сложных операций с помощью внутренних субкомпонентов. Делегирующие соединители рисуются в виде сплошных стрелок с обычными наконечниками, связывающих порт на границе внешнего компонента с символом интерфейса или портом субкомпонента. Стрелку делегирования можно помечать стереотипом <<delegate>>.

Делегирующие соединители должны связывать элементы одной полярности: требуемые элементы с требуемыми, а обеспеченные — с обеспеченными. Правила выбора направления делегирующего соединителя:

- от внешнего обеспеченного порта к внутреннему обеспеченному элементу;
- от внутреннего требуемого порта или элемента к внешнему требуемому порту;
- если порт поддерживает сложный интерфейс, включающий как требуемые, так и обеспеченные интерфейсы, направление соединителя не указывается.

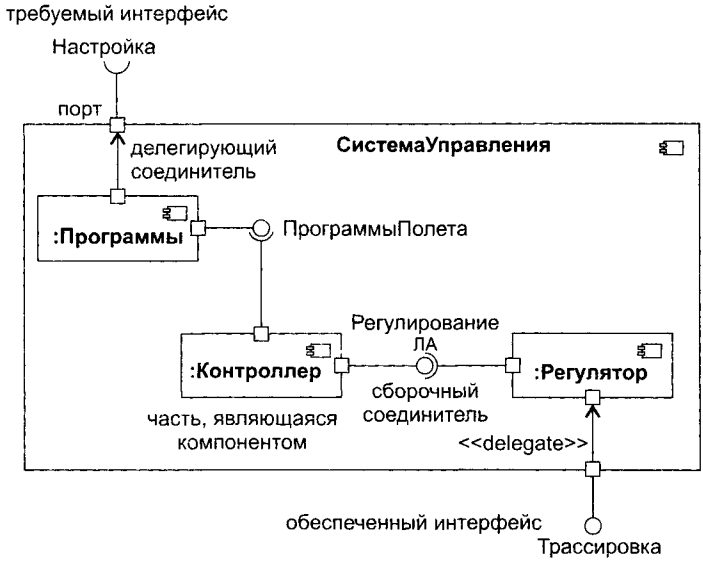


Рис. 10.14. Внутренняя структура компонента

На рис. 10.14 изображена внутренняя структура компонента для системы управления (СУ) летательным аппаратом (ЛА). Имеются два внешних интерфейса: для службы, настраивающей СУ на конкретный полет (Настройка), и для службы, отслеживающей выработку управляющих воздействий на исполнительные органы ЛА (Трассировка). Внутреннюю структуру образуют три субкомпонента: компонент

задания программ полета :Программы, компонент отработки программ управления :Контроллер и компонент :Регулятор, распределяющий управляющие воздействия между исполнительными органами летательного аппарата. Для сочленения компонентов использованы два сборочных соединителя (в варианте «шарик-и-гнездо»), а для связи с внешней средой – два делегирующих соединителя.

Пример диаграммы компонентов

Диаграмма компонентов показывает компоненты в системе, то есть программные подсистемы, из которых создается программная архитектура, а также зависимости между компонентами.

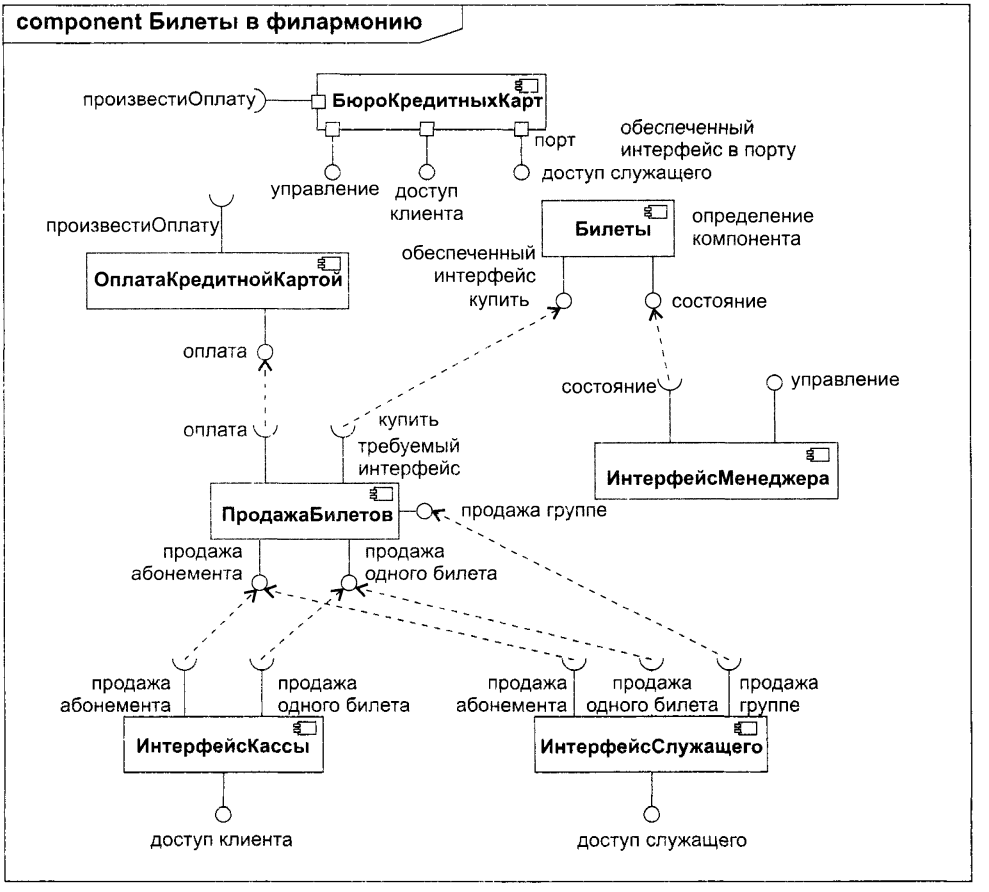


Рис. 10.15. Диаграмма компонентов

На рис. 10.15 приведена диаграмма компонентов для системы продажи билетов в филармонию. Как видим, можно купить один билет на концерт, произвести групповую покупку (например, все студенты курса идут на концерт Дениса Мацуева), приобрести абонемент на весь сезон. Покупку можно выполнить в интернет-кассе филармонии или в ее офисе. Правда, групповая покупка осуществляется только

в офисе. Оплата производится с помощью кредитной карты. Пунктирными линиями здесь показаны стрелки зависимости требуемых интерфейсов от обеспеченных интерфейсов. Впрочем, от стрелок можно отказаться при использовании нотации «шарик-и-гнездо».

Компоновка системы

За последние полвека разработчики аппаратуры прошли путь от компьютеров размером с комнату до крошечных «ноутбуков», обеспечивших возросшие функциональные возможности. За те же полвека разработчики программного обеспечения прошли путь от больших систем на Ассемблере и Фортране до еще больших систем на C++ и Java. Увы, но программный инструментарий развивается медленнее, чем аппаратный инструментарий. В чем главный секрет аппаратчиков? — спросят у аппаратчика-мальчишка программеры-буржуины.

Этот секрет — компоненты. Разработчик аппаратуры создает систему из готовых аппаратных компонентов (микросхем), выполняющих определенные функции и предоставляющих набор услуг через ясные интерфейсы. Задача инженеров упрощается за счет повторного использования результатов, полученных другими.

Повторное использование — магистральный путь развития программного инструментария. Создание нового ПО из существующих, работоспособных программных компонентов приводит к более надежному и дешевому коду. При этом сроки разработки существенно сокращаются.

Основная цель программных компонентов — допускать сборку системы из двоичных заменяемых частей. Они должны обеспечить начальное создание системы из компонентов, а затем и ее развитие — добавление новых компонентов и замену некоторых старых компонентов без перестройки системы в целом. Ключ к воплощению такой возможности — интерфейсы. После того как интерфейс определен, к выполняемой системе можно подключить любой компонент, который удовлетворяет ему или обеспечивает этот интерфейс. Для расширения системы производятся компоненты, которые обеспечивают дополнительные услуги через новые интерфейсы.

Такой подход основывается на особенностях компонента, перечисленных в табл. 10.2.

Таблица 10.2. Особенности компонента

Компонент универсален. Он живет не только в мире логических понятий, но и в физическом мире, мире битов, и не зависит от языка программирования
Компонент — заменяемый элемент. Свойство заменяемости позволяет заменить один компонент другим компонентом, который удовлетворяет тем же интерфейсам. Механизм замены оговорен современными компонентными моделями (COM, COM+, CORBA, Java Beans), требующими незначительных преобразований или предоставляющими утилиты, которые автоматизируют механизм
Компонент является частью системы, он редко автономен. Чаще компонент сотрудничает с другими компонентами и существует в архитектурной или технологической среде, предназначенной для его использования. Компонент связан и физически, и логически, он обозначает фрагмент большой системы
Компонент соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов

Вывод: компоненты — базисные строительные блоки, из которых может проектироваться и составляться система. Компонент может появляться на различных уровнях иерархии представления сложной системы. Система на одном уровне абстракции может стать простым компонентом на более высоком уровне абстракции.

Детальное проектирование

Детальное проектирование — это вторая ступень проектирования, которая следует за созданием архитектуры. В ходе этой деятельности ориентируются на максимальную подготовку к кодированию программной системы. Программисты должны получить детальные проектные решения, которые обеспечат их полной информацией для создания программного кода.

Напомним взаимосвязи между элементами всей предшествующей цепочки разработки: формирование требований — анализ требований — архитектурное проектирование.

При формировании требований создаются элементы Use Case, документирующие пожелания заказчика, эти пожелания детализируются и формализуются на этапе анализа, превращаясь в диаграммы последовательности. На этапе анализа уже приходится опираться на архитектурные черты будущей системы. Можно сказать, что параллельно с анализом начинается архитектурное проектирование. Архитектура и детальные требования питают фазу детализации проектирования. Архитектурный скелет обрастает деталями — классами, способными реализовать сценарии, описанные диаграммами последовательности. Завершается детальное проектирование в момент получения полного плана для этапа программирования.

Итак, основным строительным блоком детального проектирования являются классы. Создание структуры классов в языке UML поддерживается диаграммой классов. Перейдем к ее рассмотрению.

Диаграммы классов

Диаграммы классов считают основным средством для представления структуры систем в терминах базовых строительных блоков и отношений между ними [22, 38, 77, 91]. Вершины диаграмм классов нагружены классами, а дуги (ребра) — отношениями между ними. Диаграммы используются:

- в ходе анализа — для указания ролей и обязанностей сущностей, которые обеспечивают поведение системы;
- в ходе проектирования — для фиксации структуры классов, которые детализируют системную архитектуру.

Вершины в диаграммах классов

Вершина в диаграмме классов — это класс. Обозначение класса показано на рис. 10.16.

Имя класса указывается всегда, атрибуты и операции — выборочно. Предусмотрено задание области действия атрибута (операции). Если атрибут (операция) подчеркивается — его областью действия является класс, в противном случае — областью действия является экземпляр (рис. 10.17).

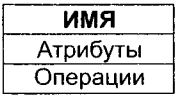


Рис. 10.16. Обозначение класса

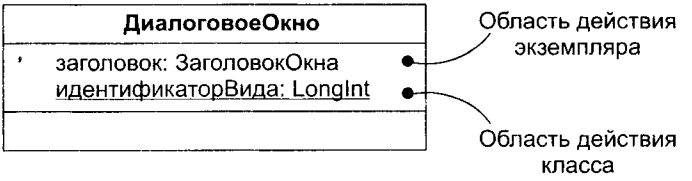


Рис. 10.17. Атрибуты уровней класса и экземпляра

Что это значит? Если областью действия атрибута является класс, то все его экземпляры (объекты) используют общее значение этого атрибута, в противном случае — у каждого экземпляра свое значение атрибута.

Атрибуты

Общий синтаксис представления атрибута имеет вид:

```
Видимость Имя : Тип [Множественность] = НачальнЗначение {Свойства}
```

Рассмотрим видимость и свойства атрибутов.

В языке UML определены четыре уровня видимости.

public	Любой клиент класса может использовать атрибут (операцию), обозначается символом +
package	Любой клиент класса, объявленный в том же пакете, может использовать атрибут (операцию), обозначается символом ~
protected	Любой наследник класса может использовать атрибут (операцию), обозначается символом #
private	Атрибут (операция) может использоваться только самим классом, обозначается символом -

ПРИМЕЧАНИЕ

Если видимость не указана, считают, что атрибут объявлен с публичной видимостью.

Свойств у атрибутов достаточно много, приведем лишь некоторые.

readOnly	После инициализации объекта значение атрибута не изменяется (по умолчанию для атрибута установлено свойство changeable — нет ограничений на модификацию значения)
ordered	Для атрибутов с множественностью, большей единицы; значения атрибута упорядочены
unique	Для атрибутов с множественностью, большей единицы; наличие дубликатов значений запрещено

Примеры объявления атрибутов.

начало	Только имя
+ начало	Видимость и имя
начало : Координаты	Имя и тип
имяФамилия: String [0..1]	Имя, тип, множественность
левыйУгол : Координаты=(0, 10)	Имя, тип, начальное значение
сумма : Integer [3]{readOnly, ordered}	Имя, тип, множественность и свойства

Операции

Общий синтаксис представления операции имеет вид:

```
Видимость Имя (Список Параметров): ВозвращаемыйТип {Свойства}
```

Примеры объявления операций.

записать	Только имя
+ записать	Видимость и имя
зарегистрировать(и: Имя, ф: Фамилия)	Имя и параметры
балансСчета ( ) : Integer	Имя и возвращаемый тип
нагревать ( ) {guarded}	Имя и свойство

В сигнатуре операции можно указать поль или более параметров, форма представления параметра имеет следующий синтаксис:

```
Направление Имя : Тип = ЗначениеПоУмолчанию
```

Элемент Направление может принимать одно из следующих значений.

in	Входной параметр, не может модифицироваться
out	Выходной параметр, может модифицироваться для передачи информации в вызывающий объект
inout	Входной параметр, может модифицироваться

Примеры свойств операций.

leaf	Конечная операция, операция не может быть полиморфной и не может переопределяться (в цепочке наследования)
isQuery	Выполнение операции не изменяет состояния объекта
sequential	В каждый момент времени в объект поступает только один вызов операций. Как следствие, в каждый момент времени выполняется только одна операция объекта. Другими словами, допустим только один поток вызовов (поток управления)
guarded	Допускается одновременное поступление в объект нескольких вызовов, но в каждый момент времени обрабатывается только один вызов охраняемой операции. Иначе говоря, параллельные потоки управления исполняются последовательно (за счет постановки вызовов в очередь)
concurrent	В объект поступает несколько потоков вызовов операций (из параллельных потоков управления). Разрешается параллельное (и множественное) выполнение операции. Подразумевается, что такие операции являются атомарными

Организация атрибутов и операций

Известно, что пиктограмма класса включает три секции (для имени, для атрибутов и для операций). Пустота секции не означает, что у класса отсутствуют атрибуты или операции, просто в данный момент они не показываются. Можно явно определить наличие у класса большего количества операций или атрибутов. Для этого в конце показанного списка проставляются три точки. Как показано на рис. 10.18, в длинных списках атрибутов и операций разрешается группировка — каждая группа начинается со своего стереотипа.

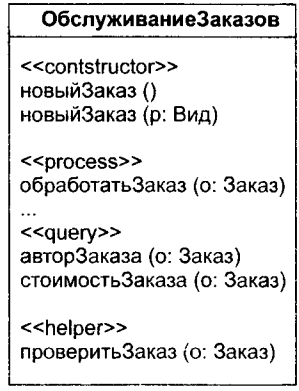


Рис. 10.18. Стереотипы для операций класса

Множественность

Иногда бывает необходимо ограничить количество экземпляров класса:

- задать ноль экземпляров (в этом случае класс превращается в утилиту, которая предлагает свои атрибуты и операции);
- задать один экземпляр (класс-singleton);
- задать конкретное количество экземпляров;
- не ограничивать количество экземпляров (это случай, предполагаемый по умолчанию).

Количество экземпляров класса называется его множественностью. Выражение множественности записывается в правом верхнем углу значка класса. Например, как показано на рис. 10.19, КонтроллерУглов — это класс-singleton, а для класса ДатчикУгла разрешены три экземпляра.

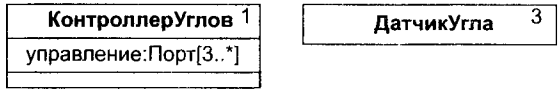


Рис. 10.19. Множественность

Множественность применима не только к классам, но и к атрибутам. Множественность атрибута задается выражением в квадратных скобках, записанным после

его типа. Например, на рисунке заданы три и более экземпляра атрибута управление (в экземпляре класса КонтроллерУглов).

Отношения в диаграммах классов

Отношения, используемые в диаграммах классов, показаны на рис. 10.20.

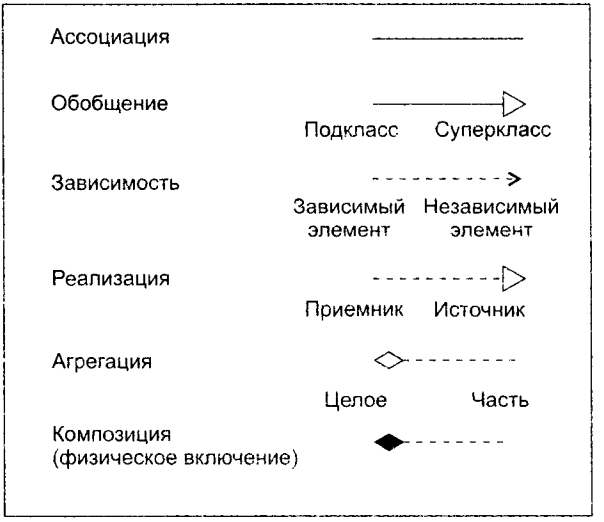


Рис. 10.20. Отношения в диаграммах классов

Ассоциации отображают структурные отношения между экземплярами классов, то есть соединения между объектами. Каждая ассоциация может иметь метку — *имя*, которое описывает природу отношения. Как показано на рис. 10.21, имени можно придать направление — достаточно добавить треугольник направления, который указывает направление, заданное для чтения имени.

Рис. 10.21. Имена ассоциаций

Когда класс участвует в ассоциации, он играет в этом отношении определенную роль. Как показано на рис. 10.22, *роль ассоциации* определяет, каким представляется класс на одном полюсе ассоциации для класса на противоположном полюсе ассоциации.

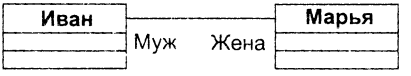


Рис. 10.22. Роли

Один и тот же класс в разных ассоциациях может играть разные роли.

Часто важно знать, как много объектов может соединяться через экземпляр ассоциации. Это количество называется *мощностью* роли в ассоциации, записывается в виде выражения, задающего диапазон величин или одну величину (рис. 10.23). Запись мощности на одном полюсе ассоциации определяет количество объектов, соединяемых с каждым объектом на противоположном полюсе ассоциации. Например, можно задать следующие варианты мощности:

- 5 — точно пять;
- * — неограниченное количество;
- 0..* — ноль или более;
- 1..* — один или более;
- 3..7 — определенный диапазон;
- 1..3,7 — определенный диапазон или число.

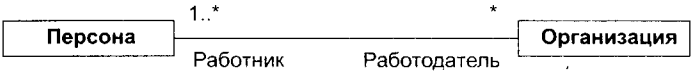


Рис. 10.23. Мощность

Достаточно часто возникает следующая проблема: как для объекта на одном полюсе ассоциации выделить набор объектов на противоположном полюсе? Например, рассмотрим взаимодействие между банком и клиентом — вкладчиком. Как показано на рис. 10.24, мы устанавливаем ассоциацию между классом Банк и классом Клиент. В контексте Банка мы имеем номерСчета, который позволяет идентифицировать конкретного Клиента. В этом смысле НомерСчета является атрибутом ассоциации. Он не является характеристикой Клиента, так как Клиенту не обязательно знать служебные параметры его счета. Теперь для данного экземпляра Банка и данного значения номераСчета можно выявить ноль или один экземпляр Клиента. В UML для решения этой проблемы вводится *квалификатор* — атрибут ассоциации, чьи значения выделяют набор объектов, связанных с объектом через ассоциацию. Квалификатор изображается маленьким прямоугольником, присоединенным к концу ассоциации. В прямоугольник вписывается атрибут — атрибут ассоциации.

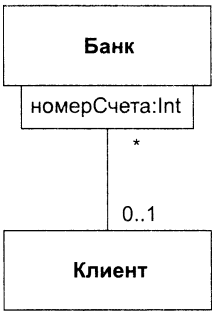


Рис. 10.24. Квалификация

Кроме того, роли в ассоциациях могут иметь пометки *видимости*. Например, на рис. 10.25 показаны ассоциации между Начальником и Женщиной, а также между Женщиной и Загадкой. Для данного экземпляра Начальника можно определить соответствующие экземпляры Женщины. С другой стороны, Загадка приватна для Женщины, поэтому она не доступна извне. Как показано на рисунке, из объекта Начальника можно перемещаться к экземплярам Женщины (и наоборот), но нельзя видеть экземпляры Загадки для объектов Женщины.

Рис. 10.25. Видимость в ассоциации

На конце ассоциации можно задать четыре уровня видимости, добавляя символ видимости к имени роли:

- по умолчанию для роли задается публичная видимость;
- пакетная видимость означает, что объекты на данном полюсе доступны объектам текущего пакета;
- приватная видимость указывает, что объекты на данном полюсе недоступны любым объектам вне ассоциации;
- защищенная видимость (protected) указывает, что объекты на данном полюсе недоступны любым объектам вне ассоциации, за исключением потомков того класса, который указан на противоположном полюсе ассоциации.

В языке UML ассоциации могут иметь атрибуты. Как показано на рис. 10.26, такие возможности отображаются с помощью классов-ассоциаций. Эти классы присоединяются к линии ассоциации пунктирной линией и рассматриваются как классы с атрибутами ассоциаций, или как ассоциации с атрибутами классов.

Атрибуты класса-ассоциации характеризуют не один, а пару объектов, в данном случае — пару экземпляров Профессора и Университета.

Отношения агрегации и композиции в языке UML считаются разновидностями ассоциации, применяемыми для отображения структурных отношений между «целым» (агрегатом) и его «частями». *Агрегация* показывает отношение по ссылке (в агрегат включены только указатели на части), *композиция* — отношение физического включения (в агрегат включены сами части).

Зависимость является отношением использования между клиентом (зависимым элементом) и поставщиком (независимым элементом). Обычно операции клиента:

- вызывают операции поставщика;
- имеют сигнатуры, в которых возвращаемое значение или аргументы принадлежат классу поставщика.

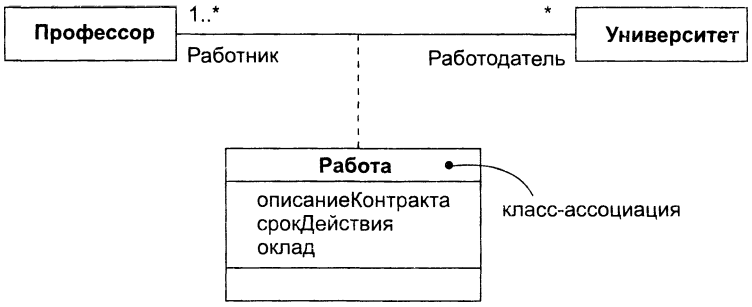


Рис. 10.26. Класс-ассоциация

Например, на рис. 10.27 показана зависимость класса Заказ от класса Книга, так как Книга используется в операциях проверкаДоступности(), добавить() и удалить() класса Заказ.

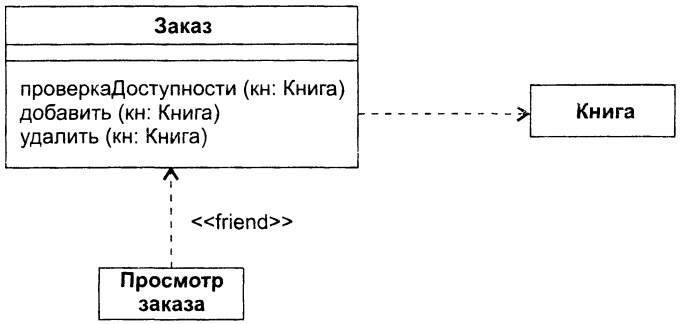


Рис. 10.27. Отношения зависимости

На этом рисунке изображена еще одна зависимость, которая показывает, что класс Просмотр Заказа использует класс Заказ. Причем Заказ ничего не знает о Просмотре Заказа. Данная зависимость помечена стереотипом <<friend>>, который расширяет простую зависимость, определенную в языке. Отметим, что отношение зависимости очень разнообразно — в настоящее время язык предусматривает 19 разновидностей зависимости, различаемых по стереотипам.

Обобщение — отношение между общим предметом (суперклассом) и специализированной разновидностью этого предмета (подклассом). Подкласс может иметь одного родителя (один суперкласс) или несколько родителей (несколько суперклассов). Во втором случае говорят о множественном наследовании.

Как показано на рис. 10.28, подкласс Летающий шкаф является наследником суперклассов Летающий предмет и Хранилище вещей. Этому подклассу достаются в наследство все атрибуты и операции двух классов-родителей.

Множественное наследование достаточно сложно и коварно, имеет много «подводных камней». Например, подкласс Яблочный_Пирог не следует производить от суперклассов Пирог и Яблоко. Это типичное неправильное использование множественного наследования: потомок наследует все атрибуты от его родителя

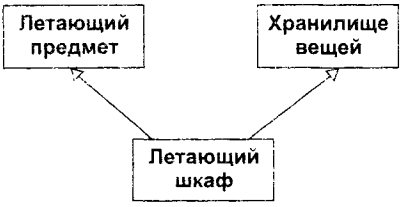


Рис. 10.28. Множественное наследование

хотя обычно не все атрибуты применимы к потомку. Очевидно, что Яблочный_Пирог является Пирогом, но не является Яблоком, так как пироги не растут на деревьях.

Еще более сложные проблемы возникают при наследовании от двух классов, имеющих общего родителя. Говорят, что в результате образуется ромбовидная решетка наследования (рис. 10.29).

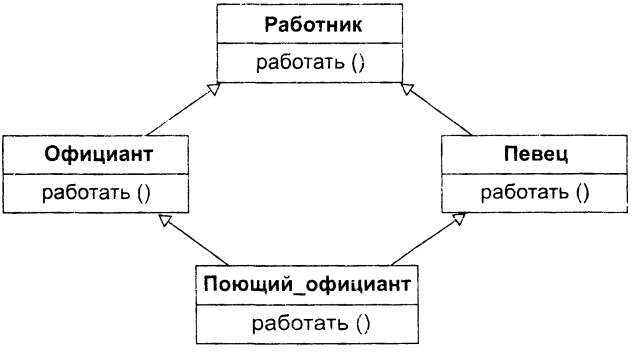


Рис. 10.29. Ромбовидная решетка наследования

Полагаем, что в подклассах Официант и Певец операция работать() суперкласса Работник переопределена в соответствии с обязанностью подкласса (работа официанта состоит в обслуживании едой, а певца — в пении). Возникает вопрос: какую версию операции работать() унаследует Поющий_официант? А что делать с атрибутами, доставшимися в наследство от родителей и общего прародителя? Хотим ли мы иметь несколько копий атрибута или только одну?

Все эти проблемы увеличивают сложность реализации, приводят к введению многочисленных правил для обработки особых случаев.

Реализация — семантическое отношение между классами, в котором класс-приемник выполняет реализацию операций интерфейса класса-источника. Например, на рис. 10.30 показано, что класс Каталог должен реализовать интерфейс Обработчик каталога, то есть Обработчик каталога рассматривается как источник, а Каталог — как приемник.

Интерфейс Обработчик каталога позволяет клиентам взаимодействовать с объектами класса Каталог без знания той дисциплины доступа, которая здесь реализована (LIFO — последний вошел, первый вышел; FIFO — первый вошел, первый вышел и т. д.).

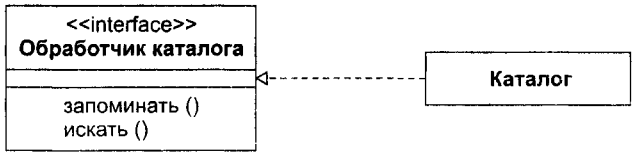


Рис. 10.30. Реализация интерфейса

Деревья наследования

При использовании отношений обобщения строится иерархия классов. Некоторые классы в этой иерархии могут быть абстрактными. *Абстрактным* называют класс, который не может иметь экземпляров. Имена абстрактных классов записываются курсивом. Например, на рис. 10.31 показаны абстрактные классы *Млекопитающие*, *Собаки*, *Кошки*.

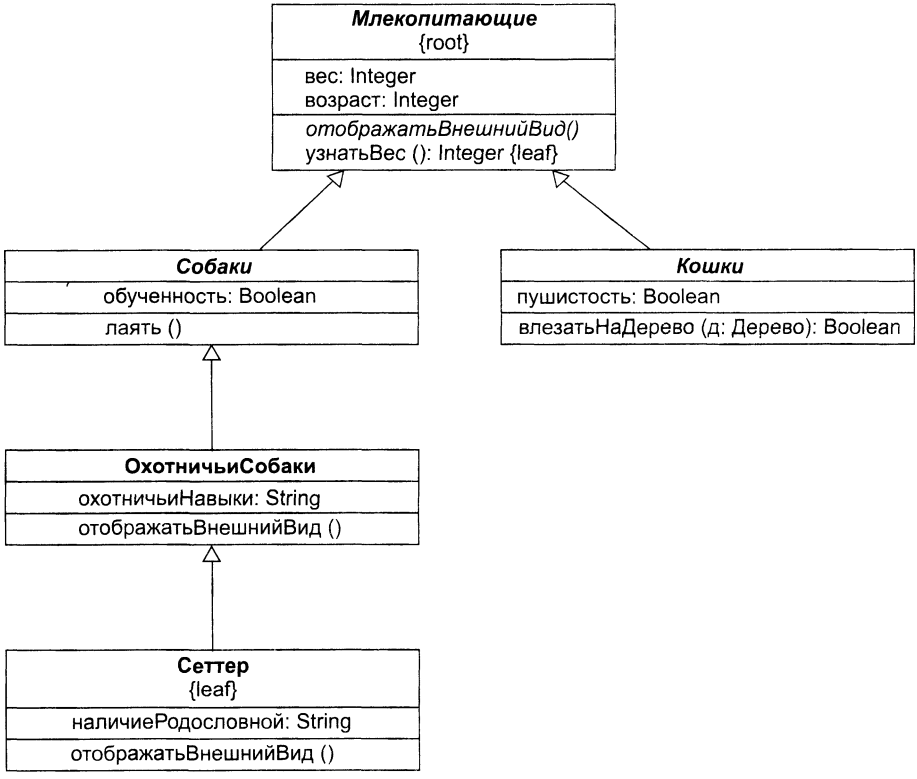


Рис. 10.31. Абстрактность и полиморфизм

Кроме того, здесь имеются конкретные классы ОхотничьиСобаки, Сеттер, каждый из которых может иметь экземпляры.

Обычно класс наследует какие-то характеристики класса-родителя и передает свои характеристики классу-потомку. Иногда требуется определить *конечный*

класс, который не может иметь детей. Такие классы помечаются теговой величиной (характеристикой) leaf, записываемой за именем класса. Например, на рисунке показан конечный класс Сеттер.

Иногда полезно отметить *корневой* класс, который не может иметь родителей. Такой класс помечается теговой величиной (характеристикой) root, записываемой за именем класса. Например, на рисунке показан корневой класс *Млекопитающие*.

Аналогичные возможности имеют и операции. Обычно операция является полиморфной, это значит, что в различных точках иерархии можно определять операции с похожей сигнатурой. Такие операции из дочерних классов переопределяют поведение соответствующих операций из родительских классов. При обработке сообщения (в период выполнения) производится полиморфный выбор одной из операций иерархии в соответствии с типом объекта. Например, отображатьВнешнийВид () и влезатьНаДерево (дуб) — полиморфные операции. К тому же операция *Млекопитающие::отображатьВнешнийВид ()* является абстрактной, то есть неполной и требующей для своей реализации потомка. Имя абстрактной операции записывается курсивом (как и имя класса). С другой стороны, Млекопитающие::узнатьВес () — конечная операция, что отмечается характеристикой leaf. Это значит, что операция не полиморфна и не может перекрываться.

Примеры диаграмм классов

В качестве первого примера на рис. 10.32 показана диаграмма классов системы управления полетом летательного аппарата.

Здесь представлен класс ПрограммаПолета, который имеет атрибут траекторияПолета, операцию-модификатор выполнятьПрограмму () и операцию-селектор прогнозОкончУправления (). Имеется ассоциация между этим классом и классом Контроллер СУ — экземпляры программы задают параметры движения, которые должны обеспечивать экземпляры контроллера.

Класс Контроллер СУ — агрегат, чьи экземпляры включают по одному экземпляру классов Регулятор скорости и Регулятор углов, а также по шесть экземпляров класса Датчик. Экземпляры Регулятора скорости и Регулятора углов включены в агрегат физически (с помощью отношения *композиция*), а экземпляры Датчика — по ссылке, то есть экземпляр Контроллера СУ включает лишь указатели на объекты-датчики. Регулятор скорости и Регулятор углов — это подклассы абстрактного суперкласса *Регулятор*, который передает им в наследство абстрактные операции *включить ()* и *выключить ()*. В свою очередь, класс *Регулятор* использует конкретный класс Порт.

Как видим, ассоциация имеет имя (Определяет полет), роли участников ассоциации явно указаны (Сервер, Клиент). Отношения композиции также имеют имена (Включать), причем на эти отношения наложено ограничение — контроллер не может включать Регулятор скорости и Регулятор углов одновременно.

Для класса Контроллер СУ задано ограничение на множественность — допускается не более трех экземпляров этого класса. Класс Регулятор скорости имеет ограничение другого типа — повторное включение его экземпляра разрешается не раньше, чем через 64 мс.

В качестве второго примера на рис. 10.33 приведена диаграмма классов для информационной системы театра. Эту систему образует 6 классов.

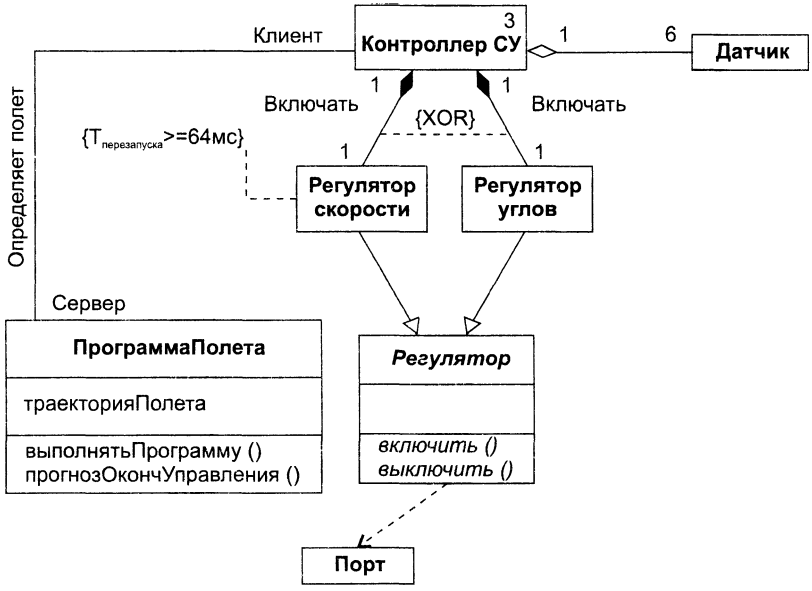


Рис. 10.32. Диаграмма классов системы управления полетом

Классы-агрегаты Театр и Труппа имеют операции добавления и удаления своих частей, которые включаются в агрегаты по ссылке. Частями Театра являются Зрители и Труппы, а частями Труппы — Актеры. Отношения агрегации между классом Театр и классами Труппа и Зритель слегка отличны. Театр может состоять из одной или нескольких трупп, но каждая труппа находится в одном и только одном театре. С другой стороны, в театр может ходить любое количество зрителей (включая нулевое количество). причем зритель может посещать один или несколько театров.

Между классами Труппа и Актер существует два отношения — агрегация и ассоциация. Агрегация показывает, что каждый актер работает в одной или нескольких труппах, а в каждой труппе должен быть хотя бы один актер. Ассоциация отображает, что каждой труппой управляет только один актер — художественный руководитель, а некоторые актеры не являются руководителями.

Ассоциация между классами Спектакль и Актер фиксирует, что в спектакле должен быть занят хотя бы один актер. впрочем, актер может играть в любом количестве спектаклей (или вообще может ничего не играть).

Между классами Спектакль и Зритель тоже определена ассоциация. Она поясняет, что зритель может смотреть любое число спектаклей, а на каждом спектакле может быть любое число зрителей.

И наконец, на диаграмме отображены два отношения наследования, утверждающие, что и в зрителях, и в актерах есть человеческое начало.

Создание начальной диаграммы классов

Работа по созданию начальной диаграммы классов требует изучения содержания всех диаграмм последовательности, являющихся результатом этапа анализа. Проводится она в три этапа.

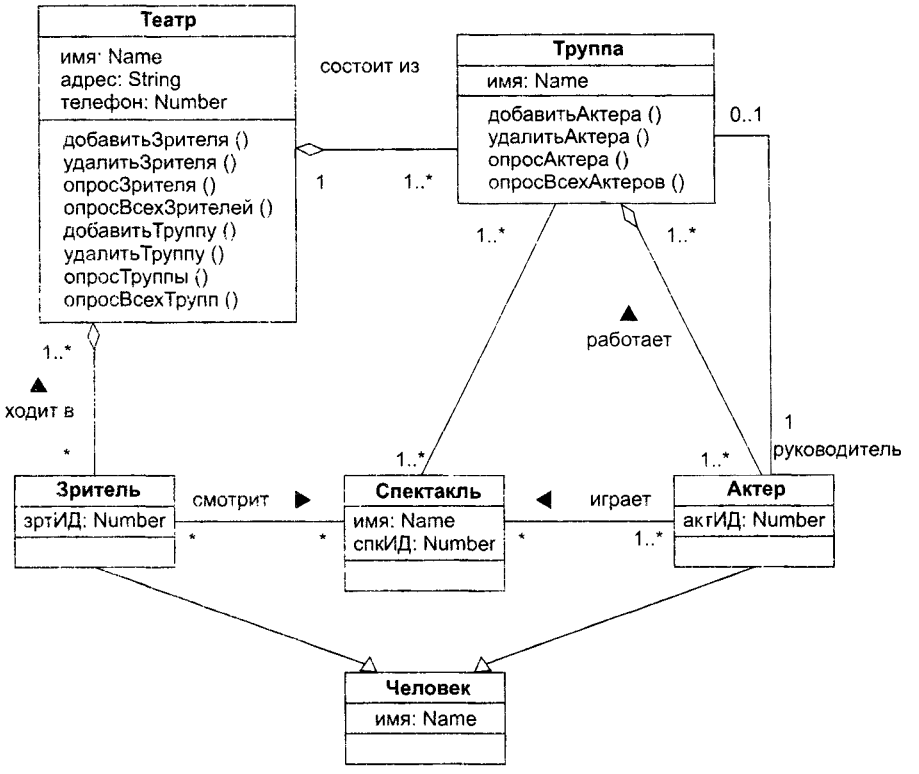


Рис. 10.33. Диаграмма классов информационной системы театра

На первом этапе выявляются и именуются классы. Для этого просматривается каждая диаграмма последовательности. Любая роль в этой диаграмме должна принадлежать конкретному классу, для которого надо придумать имя. Например, резонно предположить, что роли контроллер должен соответствовать класс Controller, поэтому класс Controller следует ввести в диаграмму. Конечно, если в другой диаграмме последовательности опять появится подобная роль, то дополнительный класс не образуется.

На втором этапе выявляются операции классов. На диаграмме последовательности такая операция соответствует стрелке (и имени) сообщения, указывающей на линию жизни участника взаимодействия. Например, если к линии жизни роли контроллер подходит стрелка сообщения регулировать, то в класс Controller нужно ввести операцию регулировать().

На третьем этапе определяются отношения ассоциации между классами — они обеспечивают пересылки сообщений между соответствующими экземплярами. Эти отношения просто воспроизводят стрелки передачи сообщений на диаграммах последовательности.

Конечно, начальная диаграмма классов далека от совершенства, но она обладает несомненным достоинством: реализует набор детальных требований, заданных диаграммами последовательности.

Дальнейшие шаги проектировщика направлены на определение атрибутов классов и оптимизацию всей диаграммы классов. Например, в классах ищутся общие черты (атрибуты и операции). Если они найдутся, то их выделяют в дополнительный суперкласс, который связывается с родственными классами отношениями наследования. Исследуется также возможность подчиненности классов, которая приводит к появлению отношений агрегации.

При выполнении этих действий нужно придерживаться общепризнанных принципов объектно-ориентированного проектирования на детальном уровне. Рассмотрим эти принципы.

Основные принципы детального проектирования

К детальному проектированию применимы четыре основных принципа объектно-ориентированного подхода. Они повышают устойчивость проектных решений к изменениям, которым эти решения наверняка будут подвергаться в жизненном цикле разработки и использования программной системы [9].

Принцип открытия–закрытия Бертрана Мейера (OCP — The Open-Closed Principle) [9]. Программный модуль должен быть открыт для расширения, но в то же время закрыт для модификации. Это утверждение кажется противоречивым, хотя представляет важнейший показатель хорошего проектирования на уровне классов. Просто следует определить класс так, чтобы его можно было расширять (в пределах обеспечиваемой функциональности) без необходимости вводить внутренние (на уровне кода или логики) модификации в существующую реализацию. Чтобы достичь этого, достаточно создать абстракцию, которая служит буфером между расширяемыми функциональными возможностями и собственно классом.

Например, предположим, что класс Контроллер опрашивает класс ДатчикУгла (рис. 10.34). Весьма вероятно, что количество типов датчиков со временем будет расти. Если внутренняя логика обработки реализована как последовательность структур if-then-else, где каждая из этих структур относится к своему типу датчиков, то добавление нового типа датчиков потребует дополнительной внутренней логики обработки (еще одного if-then-else). Это нарушение OCP.

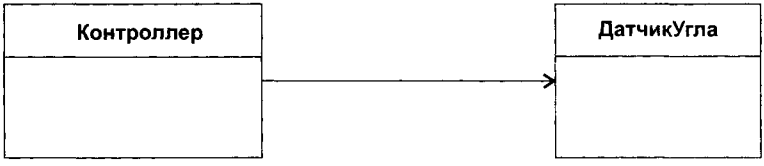


Рис. 10.34. Класс Контроллер не открыт и не закрыт

Один из способов достичь OCP для класса Контроллер иллюстрирует рис. 10.35. Интерфейс датчика обеспечивает единообразное представление разных датчиков классу Контроллер. При добавлении нового типа датчика изменять класс Контроллер не нужно. OCP соблюдается.

Принцип подстановки Барбары Лисков (LSP — Liskov Substitution Principle) [68]. Авторская трактовка этого принципа достаточно замысловата: «Если каждому объекту O1 типа S соответствует объект O2 типа T, причем таким образом, что для всех программ P, определенных на основе T, поведение P не меняется при подста-

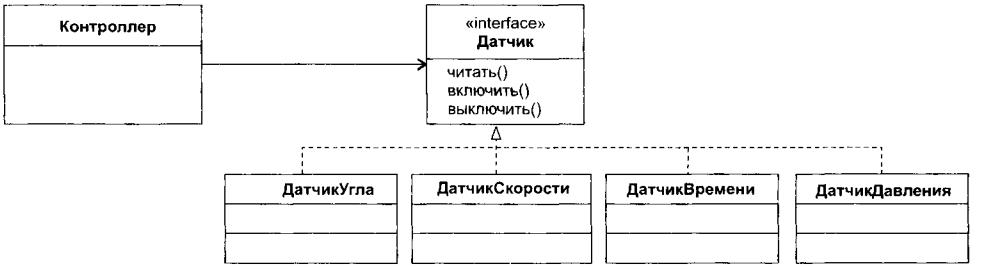


Рис. 10.35. Класс Контроллер открыт и закрыт

новке O1 вместо O2, то S является подтипом T». Суть принципа в том, что должна обеспечиваться корректность любой подстановки экземпляра дочернего класса вместо экземпляра родительского класса. Иными словами, дочерний класс обязан уметь делать все то, что умеет родительский класс.

LSP требует, чтобы любой производный класс соблюдал контракт между родительским классом и программными компонентами, которые его используют. Контракт образуют: 1) *предусловие*, которое должно соблюдаться перед использованием компонентом родительского класса; 2) *постусловие*, которое должно соблюдаться после использования компонентом родительского класса. При создании производных классов следует убедиться, что они соответствуют пред- и постусловиям контракта.

ПРИМЕЧАНИЕ

Программирование по контракту разработано Бертраном Мейером и реализовано им в языке Eiffel.

Нарушение LSP приводит, в свою очередь, к нарушению OCP. Эвристическим правилом для обнаружения класса, нарушающего LSP, является вопрос: *удаляет ли этот класс какую-либо функциональность родительского класса*?

Принцип инверсии зависимостей Роберта Мартина (DIP — Dependency Inversion Principle) [9]. Принцип провозглашает: «Будьте зависимы от абстракций. Будьте независимы от конкретики». Как мы видели при обсуждении OCP, абстракция — это место, где проектное решение может быть легко расширено. Чем больше класс зависит от других конкретных классов (а не от абстракций, таких как интерфейс), тем труднее его расширить.

На рис. 10.36 представлены зависимости в стандартной трехслойной архитектуре.

На этой диаграмме классы пакета высокого уровня Стратегия зависят от классов из пакета низкого уровня Механизм, которые, в свою очередь, зависят от классов пакета Утилиты. Следовательно, в этой структуре пакет Стратегия очень чувствителен к любым изменениям пакета Утилиты.

Принцип инверсии управления позволяет изменить направление стрелок отношений (рис. 10.37).

Здесь классы каждого пакета объявляют интерфейс для необходимых им операций. Затем на основе этих интерфейсов реализуются классы пакетов нижних

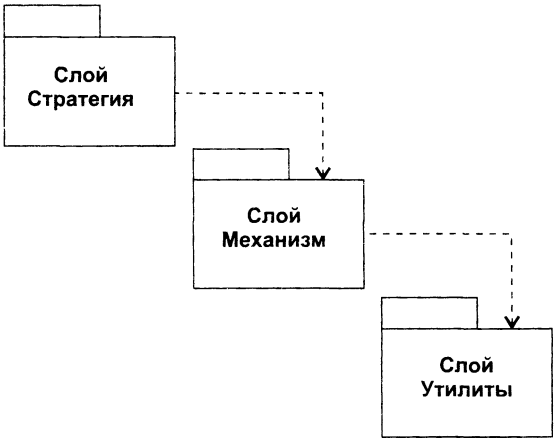


Рис. 10.36. Зависимости в стандартной трехслойной архитектуре

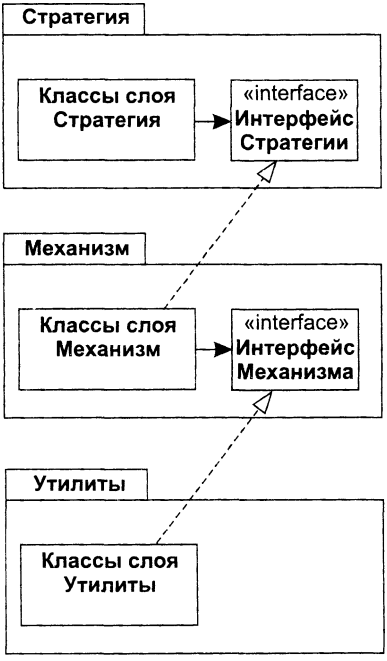


Рис. 10.37. Инвертирование отношений в трехслойной архитектуре

уровней. Теперь каждый класс пакета более высокого уровня использует следующий под ним уровень через интерфейс (интерфейс обеспечивает «развязку» и инвертирование отношений — зависимости заменяются реализациями, направленными в противоположную сторону). Таким образом, пакеты верхнего уровня не зависят от пакетов низкого уровня. Вместо этого нижние пакеты зависят от служебных интерфейсов, объявленных в верхних пакетах.

Принцип отделения интерфейса (ISP — Interface Segregation Principle). Принцип утверждает: «Одному универсальному интерфейсу для клиентов следует предпочесть набор специализированных интерфейсов»[9]. Во многих случаях разнообразные классы клиентов используют один набор операций, обеспечиваемый классом сервера. ISP предполагает, что для обслуживания каждой категории клиентов надо создать специализированный интерфейс. В интерфейсе для клиента должны быть только те операции, которые действительно необходимы данной категории. Если различные клиенты нуждаются в однотипных операциях, операции следует включать в каждый специализированный интерфейс.

Например, набор готовности летательного аппарата к полету производится многими членами экипажа. Типичными операциями здесь являются вводПараметров(), оценкаГотовности(), проверкаИсправности() и т. д. Эти операции, равно как и узкоспециальные операции, должны быть включены и в интерфейс пилота, и в интерфейс борт-инженера, и в интерфейс радиста, и в интерфейс командира корабля.

Принципы упаковки классов в архитектурные подсистемы

Достаточно часто возникает потребность группировки классов в подсистемы, пакеты и компоненты. Эта задача смыкается с задачей архитектурного проектирования и ориентирована на минимизацию сцепления пакетов и компонентов. Обсудим три принципа, способствующих ее решению.

Принцип эквивалентности повторного применения (REP — Release Reuse Equivalency Principle). *Уровень детализации повторного применения (reuse) должен соответствовать желаемому уровню детализации новой версии.* Речь идет об определении уровня повторного применения — ведь оно может обеспечиваться на уровне класса, группы классов, пакетов, компонентов, подсистем. Чем выше уровень, тем дороже обеспечивать возможность reuse, тем меньше универсальность модуля. В конце концов, все зависит от разработчиков новой версии (новой системы): из модулей какого размера они хотят собирать новый продукт? Размер повторно применяемого модуля определяет соглашение между текущими разработчиками и лицами, заинтересованными в повторном применении.

Принцип общего закрытия (CCP — Common Closure Principle). *Классы, входящие в состав пакета, должны быть закрыты по отношению к одним и тем же изменениям. Изменение, влияющее на пакет, оказывает воздействие на все классы этого пакета, не затрагивая другие пакеты.* Классы в пакете должны иметь высокую связность. Иными словами, классы в таком пакете реализуют требования, ориентированные на одну и ту же функциональность. Когда требования к функциональности меняются, изменения затрагивают только содержимое пакета. Такая упаковка обеспечивает более эффективный контроль изменений и управление версиями.

Принцип общего повторного применения (CRP — Common Reuse Principle). *Классы, входящие в состав пакета, повторно применяются совместно. Если вы повторно применяете один из классов пакета, вы повторно применяете их все.* Отсюда следует, что в составе пакета не должно быть лишних, посторонних классов (из-за которых от пакета может отказаться лицо, заинтересованное в повторном применении). Классы в пакете тоже должны демонстрировать высокую связность. При изменении любого класса меняется версия пакета. Изменение версии пакета

инициирует большой объем работы: тотальное обновление и тестирование всех зависимых пакетов. И это второй аргумент в пользу крайне осторожного включения классов в подобный пакет.

Документирование процесса проектирования

Стандарт IEEE Std 1016-2009 «Systems Design — Software Design Descriptions» предлагает документировать весь этап проектирования с помощью описания программного проектирования (SDD — software design description).

Этот стандарт задает, что SDD должно быть организовано в виде нескольких проектных представлений. Каждое представление отражает конкретный набор понятий заинтересованных лиц. Каждое проектное представление описывается с некоторой точки зрения на проектирование. Точка зрения определяет набор проектных понятий, отражаемых в представлении, а также используемый язык проектирования.

Стандарт определяет следующие точки зрения:

- *Контекстная точка зрения.* Отражает услуги, оказываемые пользователям и другим заинтересованным лицам, взаимодействующим с системой. Система рассматривается как черный ящик.
- *Композитная точка зрения.* Описывает систему как структуру, состоящую из нескольких частей, и определяет роль каждой части.
- *Логическая точка зрения.* Выявляет существующие и проектируемые типы, а также их реализацию (классы и интерфейсы, их структурные статические отношения). Могут использоваться экземпляры типов.
- *Точка зрения зависимостей.* Определяются отношения взаимодействия и доступа отдельных сущностей. Эти отношения описывают разделяемую информацию, порядок выполнения, параметры интерфейсов.
- *Информационная точка зрения.* Точка зрения применима, если ожидается присутствие персистентных (устойчивых) данных.
- *Точка зрения с использованием паттернов.* Рассматривается сотрудничество паттернов, их абстрактные роли и соединения.
- *Интерфейсная точка зрения.* Обеспечивает проектировщиков, программистов и тестировщиков информацией о правильном использовании сервисов системы. Здесь описываются детали внешних и внутренних интерфейсов, не заданные в спецификации требований. Предлагается набор интерфейсных описаний для каждой сущности.
- *Структурная точка зрения.* Используется для документирования внутренних составляющих частей и организации системы в терминах элементов.
- *Точка зрения взаимодействий.* Определяет стратегии для взаимодействия сущностей, выделяя почему, где, как и зачем происходят действия.
- *Точка зрения динамики состояний.* Применима к событийно-управляемым системам.
- *Алгоритмическая точка зрения.* Детальное описание операций (методов, функций), внутренние детали и логика каждой сущности.

- *Ресурсная точка зрения.* Моделируются характеристики и использование ресурсов системы.

Оглавление SDD имеет следующий вид.

1. Аннотация
   - Дата создания и статус
   - Выпускающая организация
   - Авторство
   - Перечень изменений
2. Введение
   - Цель
   - Область действия
   - Контекст
   - Выводы
3. Ссылки
4. Словарь
5. Основная часть
   - Заинтересованные лица и проектные понятия
   - Проектная точка зрения 1
   - Проектное представление 1
   - ...
   - Проектная точка зрения N
   - Проектное представление N
   - Обоснование проектирования.

Кооперации и паттерны

Кооперации (сотрудничества) являются средством представления комплексных решений в разработке ПО на высшем, архитектурном уровне. С одной стороны, кооперации обеспечивают компактность цельной спецификации программного продукта, с другой — несут в себе реализации потоков управления и данных, а также структур данных.

В терминологии фирмы IBM Rational (вдохновителя и организатора побед языка UML) кооперации называют реализациями элементов Use Case, да и обозначения их весьма схожи (рис. 10.38).

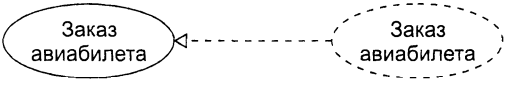


Рис. 10.38. Элемент Use Case и его реализация

Обратите внимание, что и связаны эти элементы отношением реализации: кооперация реализует конкретный элемент Use Case.

Кооперации содержат две составляющие — статическую (структурную) и динамическую (поведенческую).

Статическая составляющая кооперации задает структуру совместно работающих классов и других элементов (интерфейсов, компонентов, узлов). Чаще всего для этого используют одну или несколько диаграмм классов. Динамическая составляющая кооперации определяет поведение совместно работающих элементов. Обычно для определения применяют одну или несколько диаграмм последовательности.

Таким образом, если заглянуть под «обложку» кооперации, мы увидим набор разнообразных диаграмм. Например, требования к информационной системе авиакассы задаются множеством элементов Use Case, каждый из которых реализуется отдельной кооперацией. Все эти кооперации применяют одни и те же классы, но все же имеют разную функциональную организацию. В частности, поведение кооперации для заказа авиабилета может описываться диаграммой последовательности, показанной на рис. 10.39.

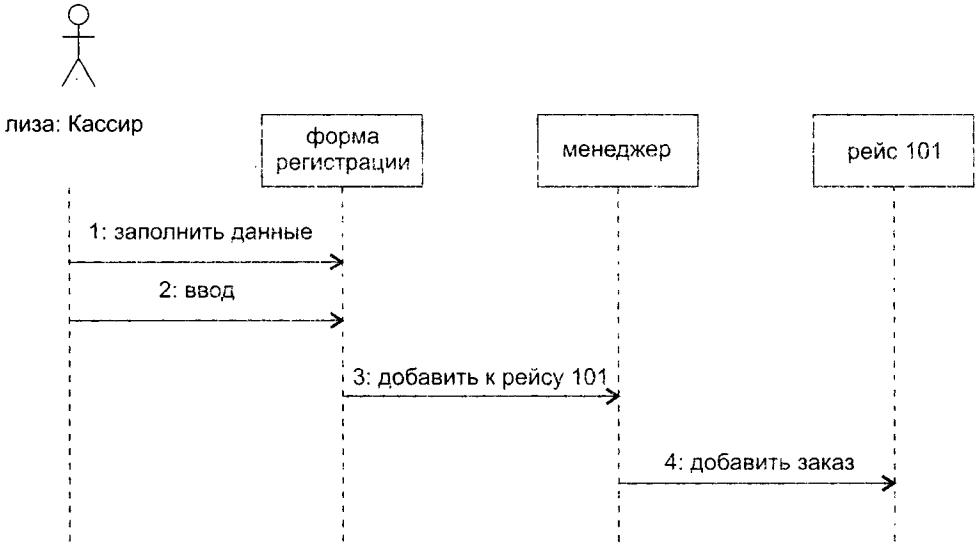


Рис. 10.39. Динамическая составляющая кооперации Заказ авиабилета

Соответственно, структура кооперации для заказа авиабилета может иметь вид, представленный на рис. 10.40.

Важно понимать, что кооперации отражают понятийный аспект архитектуры системы. Один и тот же элемент может участвовать в различных кооперациях. Ведь речь здесь идет не о владении элементом, а только о его применении.

Параметризованные, то есть настраиваемые кооперации называют паттернами (образцами). Паттерн является решением типичной проблемы в определенном контексте. Обозначение паттерна имеет вид, представленный на рис. 10.41.

На место параметров настройки паттерна подставляются различные фактические параметры, в результате создаются разные кооперации.

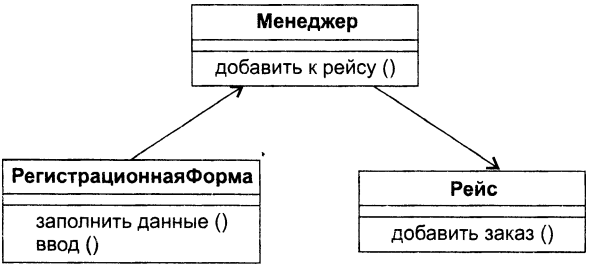


Рис. 10.40. Статическая составляющая кооперации Заказ авиабилета

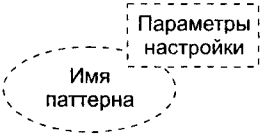


Рис. 10.41. Обозначение паттерна

Паттерны рассматриваются как крупные строительные блоки. Их использование приводит к существенному сокращению затрат на анализ и проектирование ПО, повышению качества и правильности разработки на логическом уровне, ведь паттерны создаются опытными профессионалами и отражают проверенные и оптимизированные решения [46, 51, 92].

Итак, паттерны — это наборы готовых решений, рецепты, предлагающие к повторному использованию самое ценное для разработчика — сплав мирового опыта по созданию ПО.

Наиболее распространенные паттерны формализуют и сводят в единые каталоги. Самым известным каталогом проектных паттернов, обеспечивающих этап проектирования ПО, считают каталог «Команды Четырех» (Э. Гамма и др.). Он включает в себя 23 паттерна, разделенных на три категории [51]. Как показано в табл. 10.3, по мнению «Команды Четырех», описание паттерна должно состоять из четырех основных частей.

Обсудим применение нескольких паттернов из каталога «Команды Четырех».

Таблица 10.3. Описание паттерна

Раздел	Описание
Имя	Выразительное имя паттерна дает возможность указать проблему проектирования, ее решение и последствия ее решения. Использование имен паттернов повышает уровень абстракции проектирования
Проблема	Формулируется проблема проектирования (и ее контекст), на которую ориентировано применение паттерна. Задаются условия применения
Решение	Описываются элементы решения, их отношения, обязанности, сотрудничество. Решение представляется в обобщенной форме, которая должна конкретизироваться при применении. Фактически приводится шаблон решения — его можно использовать в самых разных ситуациях
Результаты	Перечисляются следствия применения паттерна и вытекающие из них компромиссы. Такая информация позволяет оценить эффективность применения паттерна в данной ситуации

Паттерн Наблюдатель

Паттерн Наблюдатель (Observer) задает между объектами такую зависимость «один-ко-многим», при которой изменение состояния одного объекта приводит к оповещению и автоматическому обновлению всех зависящих от него объектов.

Проблема. При разбиении системы на набор совместно работающих объектов появляется необходимость поддерживать их согласованное состояние. При этом желательно минимизировать сцепление, так как высокое сцепление уменьшает возможности повторного использования. Например, во многих случаях требуется отображение данных состояния в различных графических формах и форматах. При этом объекту, формирующему состояние, не нужно знать о формах его отображения — отсутствие такого интереса благотворно влияет на необходимое сцепление. Паттерн Наблюдатель можно применять в следующих случаях:

- когда необходимо организовать непрямое взаимодействие объектов уровня логики приложения с интерфейсом пользователя. Таким образом достигается низкое сцепление между уровнями;
- когда при изменении состояния одного объекта должны изменить свое состояние все зависимые объекты, причем количество зависимых объектов заранее неизвестно;
- когда один объект должен рассылать сообщения другим объектам, не делая о них никаких предположений. За счет этого образуется низкое сцепление между объектами.

Решение. Принцип решения иллюстрирует рис. 10.42. Ключевыми элементами решения являются *субъект* и *наблюдатель*. У *субъекта* может быть любое количество зависимых от него *наблюдателей*. Когда происходят изменения в состоянии *субъекта*, *наблюдатели* автоматически об этом уведомляются. Получив уведомление, *наблюдатель* опрашивает *субъекта*, синхронизируя с ним свое отображение состояния.

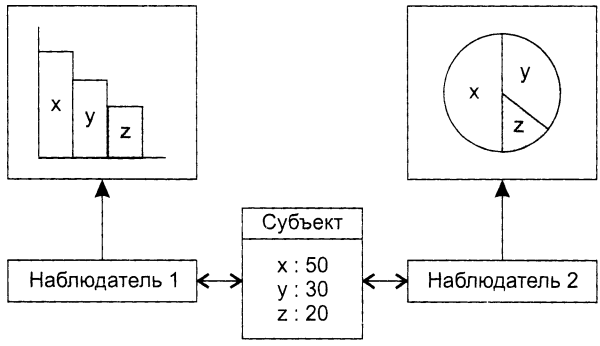


Рис. 10.42. Различные графические отображения состояния субъекта

Такое взаимодействие между элементами соответствует схеме *издатель-подписчик*. Издатель рассылает сообщения об изменении своего состояния, не имея никакой информации о том, какие объекты являются подписчиками. На получение таких уведомлений может подписаться любое количество наблюдателей.

Структурная составляющая паттерна Наблюдатель представлена на рис. 10.43. В ней определены два абстрактных класса *Субъект* и Наблюдатель. Кроме того, здесь показаны два конкретных класса КонкрСубъект и КонкрНаблюдатель, которые наследуют свойства и операции абстрактных классов. Они подключаются к паттерну в процессе его настройки. Состояние формируется Конкретным Субъектом, который унаследовал от *Субъекта* операции, позволяющие ему добавлять и удалять Конкретных Наблюдателей, а также уведомлять их об изменении своего состояния. Конкретный Наблюдатель автоматически отображает состояние и реализует абстрактную операцию *обновить() Наблюдателя*, обеспечивающую обновление отображаемого состояния.

ПРИМЕЧАНИЕ

Курсивом в данном абзаце отображены имена абстрактных классов и операций (это требование языка UML).

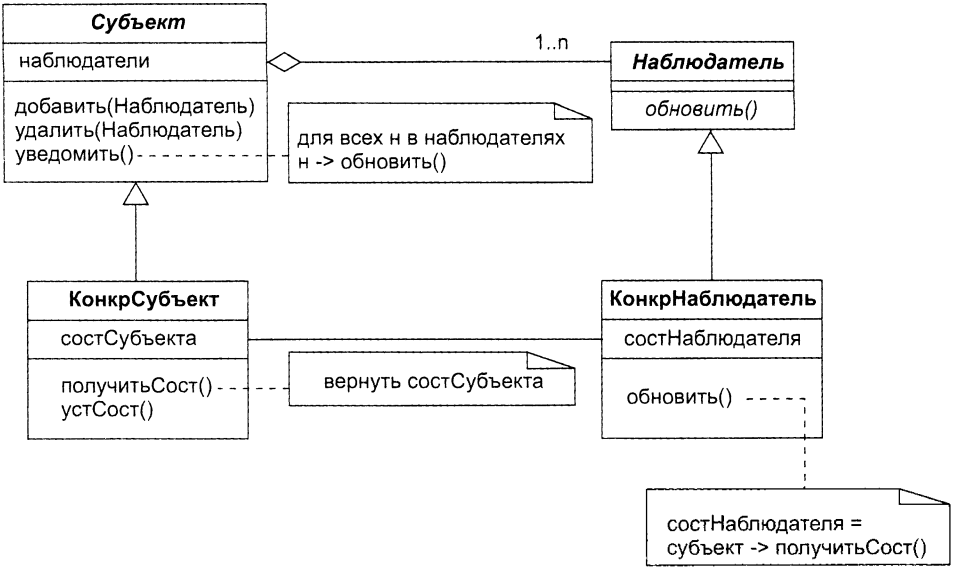


Рис. 10.43. Структурная составляющая паттерна Наблюдатель

Динамическая составляющая паттерна *Наблюдатель* показана на рис. 10.44. На рисунке представлено поведение паттерна при взаимодействии субъекта с двумя наблюдателями.

Результаты. Субъекту известно только об абстрактном наблюдателе, он ничего не знает о конкретных наблюдателях. В результате между этими объектами устанавливается минимальное сцепление (это достоинство). Изменения в субъекте могут привести к неоправданно большому количеству обновлений наблюдателей — ведь наблюдателю неизвестно, что именно изменилось в субъекте, затрагивают ли его произошедшие изменения (это недостаток).

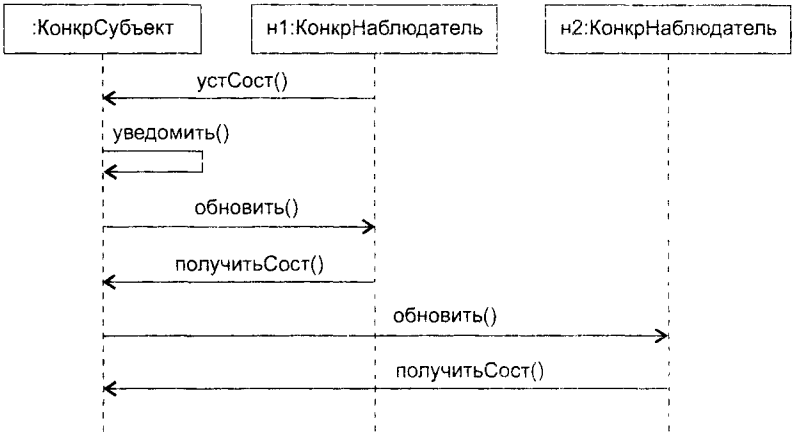


Рис. 10.44. Динамическая составляющая паттерна Наблюдатель

Обозначение паттерна Наблюдатель приведено на рис. 10.45, где показано, что у него два параметра настройки — субъект и наблюдатель.

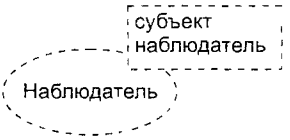


Рис. 10.45. Обозначение паттерна Наблюдатель

Эти параметры обозначают роли, которые будут играть конкретные классы, используемые при настройке паттерна на конкретное применение. Например, настройку паттерна на отображение текущего фильма кинофестиваля иллюстрирует рис. 10.46.

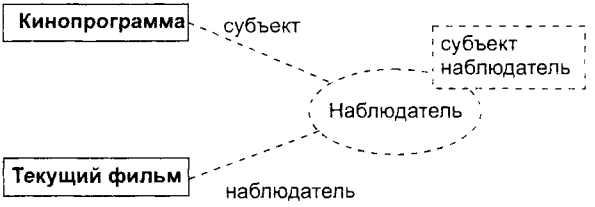


Рис. 10.46. Настройка паттерна Наблюдатель

Видим, что подключаемые конкретные классы (Кинопрограмма, Текущий фильм) соединяются с символом паттерна пунктирными линиями. Каждая пунктирная линия подписана ролью (именем параметра), которую играет конкретный класс в формируемой кооперации. Таким образом, в данном случае класс Кинопрограмма становится подклассом абстрактного класса *Субъект* в паттерне, а класс Текущий фильм — подклассом абстрактного класса *Наблюдатель* в паттерне.

Паттерн Компоновщик

Паттерн Компоновщик (Composite) обеспечивает представление иерархий часть-целое, объединяя объекты в древовидные структуры.

Проблема. Очень часто возникает необходимость создавать маленькие компоненты (примитивы), объединять их в более крупные компоненты (контейнеры), более крупные компоненты соединять в большие компоненты, большие компоненты — в огромные и т. д. При этом клиентам приходится различать компоненты-примитивы и компоненты-контейнеры, работать с ними по-разному. Это усложняет приложение. Паттерн Компоновщик позволяет ликвидировать это различие, его можно применять в следующих случаях:

- необходимо построить иерархию объектов вида часть-целое;
- нужно унифицировать использование как составных, так и индивидуальных объектов.

Решение. Ключевым элементом решения является абстрактный класс *Компонент*, который является одновременно и примитивом, и контейнером. В нем объявлены:

- абстрактная операция примитива *работать()*;
- абстрактные операции контейнера — управления примитивами-потомками *добавить(Компонент)* и *удалить(Компонент)*, а также доступа к потомку *получитьПотомка()*.

Структурная составляющая паттерна Компоновщик представлена на рис. 10.47.

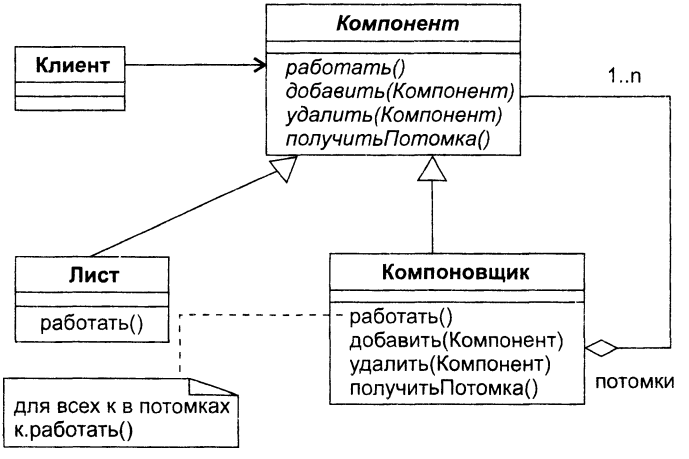


Рис. 10.47. Структурная составляющая паттерна Компоновщик

Из рисунка видно, что с помощью паттерна организуется рекурсивная композиция.

Класс *Компонент* служит простым элементом дерева, класс Компоновщик является рекурсивным элементом, а класс Лист — конечным элементом дерева. Класс *Компонент* служит родителем классов Лист и Компоновщик. Отметим, что класс

Компоновщик является агрегатом составных частей — экземпляров класса *Компонент* (таким образом задается рекурсия).

Клиенты используют интерфейс класса *Компонент* для взаимодействия с объектами дерева. Если получателем запроса клиента является объект-лист, то он и обрабатывает запрос. Если же получателем является составной объект-компоновщик, то он перенаправляет запрос своим потомкам, возможно, выполняя дополнительные действия до или после перенаправления.

Результаты. Паттерн определяет иерархии, состоящие из классов-примитивов и классов-контейнеров, облегчает добавление новых разновидностей компонентов. Он упрощает организацию клиентов (клиент не должен учитывать специфику адресуемого объекта). Недостаток применения паттерна — трудность в наложении ограничений на объекты, которые можно включать в композицию.

Обозначение паттерна Компоновщик приведено на рис. 10.48, где показано, что у него три параметра настройки — компонент, компоновщик и лист.

Рис. 10.48. Обозначение паттерна Компоновщик

Настройку паттерна на графическое приложение иллюстрирует рис. 10.49.

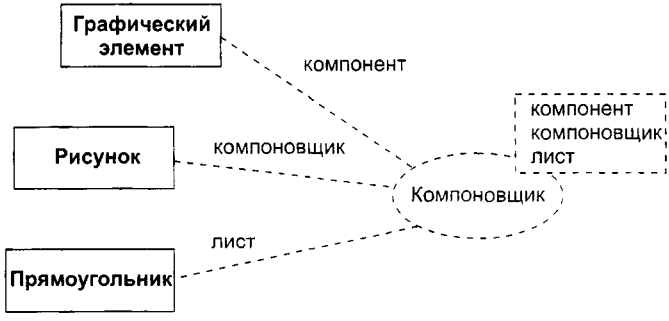


Рис. 10.49. Настройка паттерна Компоновщик

В этом случае основной операцией приложения становится операция рисовать(). Подразумевается, что такая операция входит в состав каждого из подключаемых классов, то есть классов Рисунок, Прямоугольник и Графический элемент. Операции рисовать() должны заместить операции работать() в классах паттерна.

Паттерн Команда

Паттерн Команда (Command) выполняет преобразование запроса в объект, обеспечивая:

- параметризацию клиентов с различными запросами;

- постановку запросов в очередь и их регистрацию;
- поддержку отмены операций.

Проблема. Достаточно часто нужно посылать запрос, не зная, выполнение какой конкретной операции запрошено и кто является получателем запроса. В этих случаях следует отделить объект, инициирующий запрос, от объекта, способного выполнить запрос. В результате обеспечивается высокая гибкость разработки пользовательского интерфейса — можно связывать различные пункты меню с определенной функцией, динамически подменять команды и т. д. Паттерн Команда применяется в следующих случаях:

- объекты параметризируются действием. В процедурных языках параметризация осуществляется при помощи функции обратного вызова, которая регистрируется для последующего вызова. Паттерн Команда предлагает объектно-ориентированную замену функций обратного вызова;
- необходимо обеспечить отмену операций. Это возможно благодаря хранению истории выполнения операций;
- необходимо регистрировать изменения состояния для восстановления системы в случае аварийного отказа;
- необходимо создание сложных операций, которые строятся на основе примитивных операций.

Решение. Основным элементом решения является абстрактный класс *Команда*, обеспечивающий одну абстрактную операцию *выполнять()*. Конкретные подклассы этого класса реализуют операцию выполнять(). Они задают пару получатель-действие. Получатель запоминается в экземплярной переменной подкласса. Запрос получателю посылается в ходе исполнения конкретной операции выполнять().

Структурная составляющая паттерна Команда показана на рис. 10.50. Классы этой структуры имеют следующие обязанности:

- *Команда* объявляет интерфейс для выполнения операции.
- КонкрКоманда определяет связь между экземпляром класса Получатель и действием, реализует выполнять(), вызывая нужную операцию получателя;
- Клиент создает объект класса КонкрКоманда и устанавливает его получателя;
- Инициатор просит команду выполнить запрос;
- Получатель умеет выполнять запрашиваемые операции.

В качестве Конкретной Команды могут выступать Команда Открыть, Команда Вставить. Инициатором может быть Пункт Меню, а Получателем — Документ.

Объекты этого паттерна осуществляют следующие взаимодействия:

- клиент создает объект класса КонкрКоманда и задает его получателя;
- объект класса Инициатор сохраняет объект класса КонкрКоманда;
- инициатор вызывает операцию выполнять() объекта класса КонкрКоманда;
- объект класса КонкрКоманда вызывает операцию своего получателя для исполнения запроса.

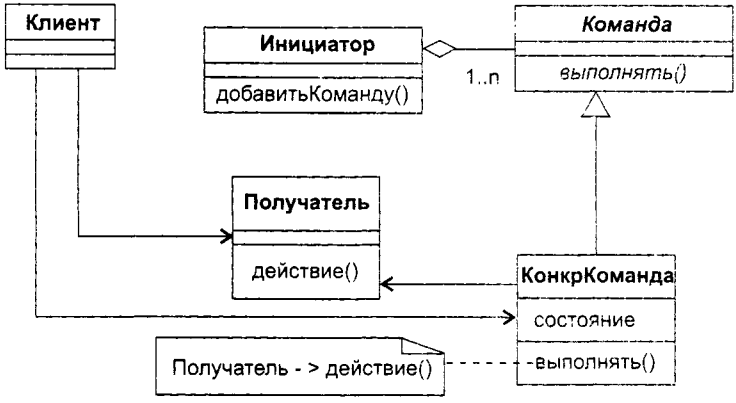


Рис. 10.50. Структурная составляющая паттерна Команда

Результаты. Применение паттерна Команда приводит к следующему:

- объект, запрашивающий операцию, отделяется от объекта, умеющего выполнять запрос;
- объекты-команды являются полноценными объектами. Их можно использовать и расширять обычным способом;
- из простых команд легко компонуются составные команды;
- легко добавляются новые команды (изменять существующие классы при этом не требуется).

Обозначение паттерна Команда приведено на рис. 10.51, где показано, что у него четыре параметра настройки — клиент, команда, инициатор и получатель.

Рис. 10.51. Обозначение паттерна Команда

Настройку паттерна на приложение с графическим меню иллюстрирует рис. 10.52.

Очевидно, что в получаемой кооперации конкретный класс Редактор играет роль клиента, классы КомандаОткрыть и КомандаВставить становятся классами Конкретных Команд (и подклассами абстрактного класса *Команда*), класс ПунктМеню замещает класс Инициатор паттерна, а конкретный класс Документ замещает класс Получатель паттерна.

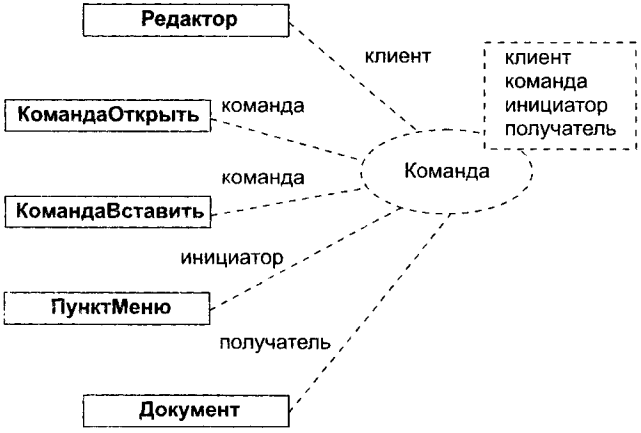


Рис. 10.52. Настройка паттерна Команда

Мышление в терминах паттернов

Хорошее проектирование начинается с рассмотрения предметной области — выяснения общей картины. Исследуя ее содержание, вы обнаруживаете иерархию проблем, которые нужно решить. Некоторые из этих проблем по своей природе глобальны, в то время как другие могут быть решены программными средствами.

А. Шаллоуей и Д. Тротт [92] предлагают подход, который дает возможность проектировщику думать в терминах паттернов:

1. Убедитесь в том, что вы выяснили всю общую картину и понимаете предметную область, в которой должна работать создаваемая программная система. В этом должна помочь модель требований.
2. Отыщите паттерны, присутствующие в предметной области, и проанализируйте полученный набор.
3. Упорядочите паттерны набора по степени их влияния на ход проектирования. Идея заключается в том, что паттерн может создавать благоприятные условия для внедрения других паттернов.
4. Примените самый влиятельный паттерн на текущем шаге проектирования, после чего исключите его из набора.
5. Выявите любые дополнительные паттерны, которые могут появиться в полученном решении. Добавьте их к набору, сформированному на шаге 3.
6. Повторяйте применение шагов 4 и 5 к оставшемуся набору паттернов.
7. Внесите в последнее проектное решение все необходимые дополнительные детали, то есть адаптируйте каждый паттерн к специфическим особенностям системы. Запишите определения классов и их методов.

Важно отметить, что паттерны являются зависимыми решениями. Паттерны проектирования, которые присутствуют на высшем уровне абстракции, будут неизменно влиять на способ применения других паттернов, на более низких уровнях абстракции. Кроме того, паттерны оказывают друг на друга взаимное влияние[1]. Следовательно, выбор архитектурного паттерна влияет на выбираемые паттерны проектирования. Аналогично при использовании конкретного паттерна проектирования вы вынуждены применить сотрудничающие с ним паттерны.

Для иллюстрации рассмотрим веб-систему дистанционного обучения. Если вы рассматриваете общую картину, следует обратиться к ряду фундаментальных проблем:

- Как обеспечить информацию об услугах дистанционного обучения?
- Как продавать услуги дистанционного обучения клиентам в интернет-среде?
- Как обеспечить в интернет-среде мониторинг и необходимый уровень безопасности?

Каждая из этих фундаментальных проблем может быть преобразована в набор субпроблем. Например, проблема «как продавать в интернет-среде» решается паттерном е-коммерции, что требует привлечения большого набора паттернов на низших уровнях абстракции. Паттерн е-коммерции (это архитектурный паттерн) подразумевает паттерны для формирования учетной записи клиента, отображения продаваемой продукции, выбора покупаемой продукции и т. д. Следовательно, если вы думаете в терминах паттернов, важно определить, существует ли паттерн для формирования учетной записи. Если вам доступен подходящий паттерн УстановкаСчета, нужно учесть, что он может сотрудничать с такими паттернами, как СозданиеВходнойФормы, УправлениеВходнымиФормами и ВалидацияФормы. Каждый из этих паттернов очерчивает решаемые проблемы и решения, которые могут быть применены.

Шаги паттерн-ориентированного проектирования

Использование идей паттерн-ориентированного проектирования приводит к следующим шагам:

1. Исследование модели требований и разработка иерархии проблем. Вначале определяют широкую проблему (высокого уровня абстракции), а затем дробят ее на узкие (меньшие) проблемы, продвигаясь к более низким уровнях абстракции.
2. Поиск архитектурного паттерна для решения выделенной широкой проблемы. Если архитектурный паттерн доступен, исследуют все паттерны, сотрудничающие с ним. На основе выбранных паттернов строят проектное решение с высоким уровнем абстракции.
3. Детализация архитектурного паттерн-решения. Используя рекомендации в описании архитектурного паттерна, ищут паттерны проектирования для реализации его подсистем. Если подходящий паттерн найден, модифицируют проектное решение, внедряя в него паттерн-решение подсистемы. Шаг повторяют до тех пор, пока не будут обработаны все подсистемы.

[1] Попадешь в хорошую компанию — будешь хорошим. Попадешь в плохую компанию — будешь плохим ☺.

4. Шаги 2–3, повторяются для учета всех широких проблем.
5. Совершенствование качества решения. Поскольку решение получено простым применением паттернов, возможны проблемы с его качеством. Модификация структурных и поведенческих решений опирается на прямые измерения проектных характеристик.

Здесь описан нисходящий процесс проектирования (от общего к частному). На практике его шаги могут перемежаться с шагами восходящего процесса, которые применимы к отдельным сегментам системы.

Проектирование пользовательского интерфейса

Чаще всего интерфейс обеспечивает взаимодействие человека с программной системой. При проектировании важно понимать сверхзадачу пользовательского интерфейса — служить переводчиком в диалоге пользователя и программной системы. Этот переводчик должен учитывать все сильные и слабые стороны человека, его умственные и физические способности. Возможности и ограничения человека положены в основу шести принципов проектирования пользовательского интерфейса, вобравших в себя лучшее из опыта разработки интерфейсов (табл. 10.4). Каждый из них выделяет один из общих вопросов, с которым ассоциируется несколько связанных между собой идей. Этими общими вопросами являются структура, простота, видимость, обратная связь, терпимость и повторное использование.

Таблица 10.4. Принципы проектирования пользовательского интерфейса

Принцип	Описание
Структурная организация	Организация пользовательского интерфейса должна быть целесообразной, продуманной и удобной. Родственные понятия следует связывать, а независимые — разделять. По внешнему виду элементы разного назначения должны различаться, а родственные — выглядеть похоже
Простота	Управление наиболее распространенными операциями должно быть максимально простым. Общение с пользователем следует вести на понятном для него языке. Должны предоставляться видимые ссылки на более сложные операции
Видимость	Все функции и данные, необходимые для решаемой задачи, должны быть видны и доступны
Обратная связь	Пользователь должен оповещаться о действиях и ошибках системы, а также о важных событиях внутри нее. Сообщения должны быть ясными, краткими и понятными пользователю
Толерантность	Интерфейс должен быть гибким и терпимым к ошибкам пользователя. Ущерб из-за ошибок пользователя следует снижать за счет возможности отмены и повтора действий. Любые разумные действия пользователя нужно разумно интерпретировать
Повторное использование	Многократно применяются как структурные компоненты системы, так и динамические фрагменты — фрагменты поведения системы. Многократное использование должно повышать устойчивость системы и уменьшать объем информации, которую пользователям приходится запоминать

Современный пользовательский интерфейс является, как правило, графическим — применяемые в нем элементы демонстрируют высокий уровень выразительности графических примитивов (табл. 10.5).

Таблица 10.5. Типовые элементы графического пользовательского интерфейса

Элементы	Описание
Окна	Отображают на экране различную структурированную информацию
Пиктограммы	Представляют различные элементы и типы данных, файлы, процессы
Меню	Текстовый ввод команд заменяется выбором команд из меню
Указатели	В качестве устройства указания команд в меню и отдельных элементов в окне используют мышь
Графические элементы (рисунки)	Могут использоваться совместно с текстовыми

Разумно спроектированный пользовательский интерфейс очень важен для успешной работы программной системы. Сложный в применении интерфейс создает предпосылки для многочисленных ошибок пользователя. Нечеткие и двусмысленные сообщения запутывают пользователя, в результате чего его действия могут привести к искажению данных и формированию неправильных результатов.

Хороший пользовательский интерфейс должен быть практичным, то есть ориентированным на удобство использования человеком.

Качество проектирования практичного интерфейса следует проверять с помощью вычислений по метрикам. Рассмотрим набор метрик для оценки практичности интерфейсов, предложенный Л. Константантайном и Л. Локвудом в замечательной работе [5].

Сущностная эффективность

Л. Константантайн и Л. Локвуд предложили проводить оценку на основе специальной разновидности диаграмм Use Case — сущностных диаграмм. В сущностной диаграмме элементы Use Case содержат предельно короткие сценарии действий. Эти сценарии отражают саму суть необходимых действий, в них нет ничего лишнего.

В соответствии со структурным принципом, поведение надо воплощать в решениях, обеспечивающих быстрое взаимодействие. *Сущностная эффективность* — это простой показатель близости рассматриваемого интерфейса к идеалу, выраженному сущностной моделью Use Case. Сущностная эффективность измеряется отношением длины сущностного сценария к длине воплощенного сценария, то есть отношением минимального количества шагов к количеству шагов, которые пользователю реально следует сделать для решения задачи:

$$СЭ = 100 \cdot \frac{S_{\text{сущн}}}{S_{\text{реальн}}}.$$

Подсчет реального количества шагов базируется на соглашении о единичном шаге пользователя. Например, можно исходить из такого представления о шаге, которое иллюстрируется табл. 10.6.

Таблица 10.6. Возможные «шаги» пользователя

Описание отдельного шага пользователя
Ввод данных в одно поле, оканчивающийся переводом строки, табуляцией или другим разделителем полей
Пропуск ненужного поля или элемента управления путем нажатия клавиши табуляции или другой клавиши навигации
Выбор поля, объекта или группы элементов щелчком, двойным щелчком или с помощью указательного устройства
Выбор поля, объекта или группы элементов нажатием клавиши или последовательности клавиш
Переход от работы с клавиатурой к работе с указательным устройством или обратно
Выполнение действия посредством щелчка или двойного щелчка указательным устройством на инструменте, командной кнопке или другом визуальном объекте
Выбор меню или элемента меню с помощью указательного устройства
Выполнение действия нажатием «горячей клавиши» или последовательности клавиш, включая активизацию меню специальной клавишей
Перетаскивание объекта с помощью указательного устройства

В качестве примера рассмотрим модель Use Case для интерфейса банкомата. Ее сущностный элемент Use Case получениеДенег задает три шага (идентификация пользователя, выбор действия, изъятие денег). В реальности при отработке этого элемента пользователю придется совершить восемь действий: вставка карточки, ввод PIN, нажатие клавиши продолжения транзакции, выбор действия и нажатие клавиши ввода, ввод суммы, нажатие клавиши подтверждения суммы, получение карточки, получение денег.

Следовательно, $СЭ = 3/8 = 37{,}5\%$. Интерфейс можно улучшить, предложив пользователю после идентификации выбрать «обычное действие» (снятие денег со счета), а затем предложив «обычную сумму» для снятия. В этом случае, если клиенту действительно нужно снять деньги и его интересует именно предложенная сумма, то для реализации сценария потребуется уже не восемь, а пять шагов (вставить карточку, ввести PIN-код, выбрать «обычное действие», вынуть карточку, забрать деньги). Это значит, что сущностная эффективность возрастает до 62,5%.

Если анализируемый элемент Use Case можно реализовать несколькими способами (как обычно и бывает), можно посчитать минимальное и максимальное количество шагов и тем самым выявить, в каких пределах лежат эти значения. Общее правило: самый длинный путь выбирается новичками, а самый короткий считается максимально быстрым путем для опытных пользователей.

Поскольку оценка сущностной эффективности заключается в сравнении реальных шагов с шагами, перечисленными в сущностном описании, результаты очень сильно зависят от качества сущностной модели Use Case. Небрежные или неполные модели порождают приукрашенные результаты. На практике лучше вначале прорецензировать описание элементов Use Case и попытаться его упростить и только затем переходить к вычислению сущностной эффективности.

Согласованность задач

Согласованность задач (*СЗ*) — вторая метрика, основанная на элементах Use Case и измеряющая эффективность и простоту. *СЗ* служит показателем соответствия сложности задач, решаемых с помощью интерфейса, ожидаемой частоте их использования. Чем популярнее задача, тем проще должно быть ее выполнение — таково правило хорошего тона. Согласованность задач вычисляется по корреляции между задачами, классифицирующимися в соответствии с ожидаемой частотой использования и сложностью воплощения. Эта метрика измеряется в процентах со знаком: от –100 до +100%. Если проект идеален с точки зрения согласованности задач, — то есть более популярные задачи всегда решаются проще и быстрее менее популярных, — то *СЗ* = 100%. Если же складывается обратная ситуация (задачи, к которым обращаются чаще, решаются дольше и сложнее), то и значение *СЗ* будет противоположным: *СЗ* = –100%. Конечно, значение *СЗ* будет равно 0%, если вообще не прослеживается какая-либо зависимость между популярностью и сложностью задач.

Л. Константайн и Л. Локвуд предлагают измерять зависимость между частотой использования задачи и сложностью ее решения с помощью статистической меры «*Кендаллово* τ (тау)». Для вычисления τ, надо просто выписать все задачи, упорядочив их по ожидаемой частоте применения и сложности исполнения. Сложность исполнения определяется количеством действий, которые необходимо совершить для решения задачи.

Кендаллово τ позволяет легко и просто вычислять согласованность задач. Оно задает отношение всех корректно упорядоченных пар к тем, которые упорядочены некорректно. Формула согласованности задач вычисляется следующим образом:

$$СЗ = 100 \cdot \tau = 100 \cdot \frac{D}{P},$$

где D — мера несогласованности: число пар задач, корректно отсортированных по реальному количеству шагов, меньше числа некорректно отсортированных пар, P — число возможных пар задач.

Если сложность или длина всех задач разная, P вычисляется так:

$$P = \frac{N(N-1)}{2},$$

где N — число упорядочиваемых задач.

Для небольших задач *СЗ* можно вычислить вручную. Рассмотрим экран с пятью задачами, которые, вероятнее всего, будут отсортированы следующим образом.

Задачи (сортировка по убыванию ожидаемой частоты)	**Реальная длина** (число действий пользователя при реализации элемента Use Case)
Задача А	7
Задача Б	7
Задача В	5
Задача Г	8
Задача Д	6

Текущие результаты проектирования, выраженные в количестве шагов реализации, показаны в правой колонке. Для начала мы разрываем связь между двумя первыми упорядоченными задачами. Предположим, задача А считается очень тяжелой по сравнению с задачей Б, несмотря на то что количество пользовательских действий, требуемое для их решения, одинаковое.

Для каждой пары чисел в правой колонке выясняем, правильно ли она отсортирована с точки зрения ожидаемой частоты. Если это так, то прибавляем 1 к индексу несогласованности, *D*, в формуле *СЗ*; для каждой пары, упорядоченной некорректно, из *D* вычитается 1. Другими словами, мы сравниваем 7+ и 7, получаем в результате –1, потом 7+ и 5, получаем снова –1, затем 7+ и 8, в результате чего получаем 1 и т. д., пока мы не пройдем по всем парам. В итоге $D = -2$. Теперь можно найти *P*:

$$P = \frac{5 \cdot 4}{2} = 10,$$

$$СЗ = -20\%.$$

Таким образом, оцениваемое сейчас решение оказывается неудачным по части пользовательского интерфейса. Можно ли его улучшить, хотя бы в отношении популярных задач? Для этого надо исключить некоторые действия в двух наиболее популярных задачах, но добьемся мы этого лишь при усложнении задачи В (третьей по рейтингу):

Задачи (сортировка по убыванию ожидаемой частоты)	**Реальная длина** (число действий пользователя при реализации элемента Use Case)
Задача А	4
Задача Б	6
Задача В	7
Задача Г	8
Задача Д	6

Вычисляя согласованность задач для обновленной модели, видим, что она улучшилась:

$$СЗ = 52{,}7\%.$$

Наблюдаемость задач

Наблюдаемость задач (*НЗ*) — это относительно несложная метрика, построенная на базе элементов Use Case. В ее основе лежит принцип видимости: пользователь должен видеть то и только то, что ему нужно знать и использовать для решения данной задачи. Эта метрика служит для измерения наблюдаемости нужных частей интерфейса.

Проще всего определять наблюдаемость задач в терминах действий, выполняемых при их выполнении, выделяя шаги, использующие скрытые возможности или нацеленные на получение доступа к тем или иным функциям. Функция считается наблюдаемой, когда вы ее видите в нужный момент; значит, шаги, предпринимаемые лишь для получения доступа к функции или наблюдения какой-либо части

пользовательского интерфейса, снижают общую наблюдаемость задач. Формула вычисления наблюдаемости задач имеет вид:

$$НЗ = 100 \cdot \left(\frac{1}{S_{общ}} \cdot \sum_{\forall i} V_i \right),$$

где $S_{общ}$ — общее число реальных шагов по реализации элементов Use Case, V_i — наблюдаемость (от 0 до 1) отрабатываемого шага i.

Формула *НЗ* выражается в виде процента от общего числа шагов, поскольку более растянутые и сложные задачи могут распределяться между несколькими контекстами взаимодействий. Наблюдаемость задачи достигает максимального значения (100%) тогда, когда все необходимое для данного действия наблюдаемо. Нулевая наблюдаемость — это исключительный случай. Можно вообразить сверхсекретный интерфейс удаленного доступа к очень важной информации. Когда пользователь подключается, никто не спрашивает у него имя, пароль; он должен сам знать, в каком порядке и с какими разделителями следует вводить эту информацию. Успешный вход в систему отображается только лишь переходом курсора на следующую строку, после чего пользователь должен указать последовательность команд, не получая при этом никаких подсказок и обратной связи. Такой интерфейс с командной строкой подразумевает, что каждое действие выполняется на основе одних лишь «внутренних знаний», без каких-либо визуальных ключей и подсказок.

Для подсчета *НЗ* необходимо отработать сущностный элемент Use Case или набор этих элементов с помощью анализируемого сценария. Затем подсчитывается общее количество шагов. Для каждого из шагов аналитик определяет, не служит ли он для доступа к невидимым функциям, которые вообще-то должны быть видимы из данной точки пользовательского интерфейса, и не использует ли он скрытые возможности, также не отображенные на интерфейсе. Правила оценки наблюдаемости действий отражены в табл. 10.7.

Таблица 10.7. Оценка наблюдаемости действий

Название категории	Описание категории	Описание действия
Скрытые	Скрытые действия рассчитаны на то, что пользователь сам знает, как работать с приложением, независимо от того, какие подсказки дает ему интерфейс и дает ли он их вообще.	Ввод требуемого кода или нажатие горячих клавиш без каких-либо визуальных приглашений и подсказок
	Примеры: запуск общего контекстного меню щелчком правой клавиши на пустом экране или нажатие горячих клавиш, о которых нигде ничего не говорится. Скрытым действиям присваивается нулевое значение наблюдаемости	Доступ к элементу или элементам, не имеющим визуального представления на интерфейсе (пример: панель задач Windows в скрытом состоянии)
		Любое действие с объектами или элементами, которые могут быть и видимыми, но их выбор на основе наблюдаемой информации неочевиден

Название категории	Описание категории	Описание действия
Экспозиционные	Действие считается экспозиционным, если его задача состоит исключительно в предоставлении доступа к некоторому элементу или в том, чтобы сделать его наблюдаемым, и при этом оно не приводит к изменению контекста взаимодействий. Экспозиционные действия оказывают вспомогательный эффект на наблюдаемость задачи; самим этим действиям присваивается значение наблюдаемости 0,5, если только для их выполнения не требуется обращаться к скрытым элементам (в этом случае их считают скрытыми; наблюдаемость = 0)	Раскрытие выпадающего списка Раскрытие меню или подменю Раскрытие контекстного меню щелчком правой кнопкой мыши на некотором объекте Раскрытие диалогового окна свойств объекта Развертывание объекта или исследование его свойств Раскрытие или вывод на экран палитры инструментов Раскрытие прикрепленной дополнительной панели диалога Переход на другую страницу или закладку диалогового окна
Отложенные	Действие считается отложенным, если оно предназначено для получения доступа или отображения каких-либо нужных элементов и в результате приводит к изменению контекста взаимодействий. Отложенные действия или действия по переключению контекстов, являющиеся первыми или последними шагами расширенных и других необязательных взаимодействий, оказывают вспомогательный эффект на наблюдаемость задачи, так как они предоставляют доступ к элементам, которые могут и не понадобиться в ходе отработки всех сценариев. Этим действиям присваивается значение наблюдаемости 0,5, если только для их выполнения не требуется обращаться к скрытым элементам (в этом случае их считают скрытыми; наблюдаемость = 0). Обязательные изменения контекстов, подразумеваемые большинством сценариев, оказывают сильное влияние на наблюдаемость задач; их собственная наблюдаемость равна нулю	Раскрытие диалогового окна Закрытие диалогового окна или окна сообщения Переключение на другое окно Переключение на другое приложение или его запуск
Непосредственные	Действие, выполняемое при отработке какого-либо сценария, считается непосредственным, если оно не попадает ни в одну из перечисленных выше категорий, то есть если оно не скрытое, не экспозиционное и не отложенное. Непосредственными являются те действия, которые выполняются посредством наблюдаемых элементов, выбор которых очевиден, и при этом они не служат для предоставления доступа или отображения других объектов. Примеры: применения инструментов для работы с объектами, ввод данных в наблюдаемое на экране поле, изменение значения флажка. Непосредственным действиям присваивается единичное значение наблюдаемости	

Наличие на экране каждого ненужного элемента снижает общую наблюдаемость. На практике это соображение имеет смысл только тогда, когда учитываются абсолютно все элементы Use Case, поддерживаемые системой, что часто оказывается делом весьма непрактичным.

Пример 10.1. Производится подготовка слайдов презентации. Требуется, чтобы объект автоматически оказывался в правом верхнем углу при появлении первого слайда. Для этого надо выполнить следующие действия.

Действие	Тип	Наблюдаемость
Выбор объекта	Непосредственное	1,0
Открытие меню Slide Show	Экспозиционное	0,5
Открытие диалогового окна Custom Animation	Отложенное	0,0
Открытие выпадающего списка	Экспозиционное	0,5
Выбор Fly From Right	Непосредственное	1.0
Щелчок на закладке Timing	Экспозиционное	0,5
Установка свойства Automatically	Непосредственное	1.0
Щелчок на кнопке OK для закрытия диалогового окна	Отложенное	0.0
		Всего: 4,5

Всего для решения задачи потребовалось 8 шагов, значит, *НЗ* = 56,25%. Улучшить наблюдаемость можно несколькими путями:

- Расположить средства управления условиями анимации и ее стилем на одной и той же закладке диалогового окна (они тесно связаны между собой).
- Считать анимацию свойством объекта, которое можно настраивать в «Инспекторе свойств», а не в модальном диалоговом окне, блокирующем все прочие взаимодействия.

Единообразие компоновки

Единообразие компоновки (*ЕК*) помогает оценить качество визуальной компоновки.

Единообразие компоновки измеряет некоторые параметры пространственного размещения компонентов, *не обращая внимание* на их содержание. Оно оценивает единообразие, систематичность компоновки пользовательского интерфейса. Это важно, поскольку беспорядочно расположенные компоненты сильно ухудшают практичность. Компоновка должна быть опрятной, умеренно систематичной: в этом случае ее будет легче понимать и использовать. Единообразие компоновки вычисляется по формуле:

$$EK = 100 \cdot \left(1 - \frac{(N_h + N_w + N_t + N_l + N_b + N_r) - M}{6 \cdot N_c - M}\right),$$

где N_c — суммарное количество визуальных компонентов на экране, в диалоговом окне или другой единице интерфейса.

N_h, N_w, N_t, N_l, N_b, и N_r — это соответственно количество компонентов с разной высотой, шириной, верхним, левым, нижним и правым отступами. M – это поправка

на минимально возможное число размещений и размеров, при котором значение *EK* будет лежать в пределах от 0 до 100:

$$M = 2 + 2 \cdot \left\lceil 2\sqrt{N_{\text{компонентов}}} \right\rceil,$$

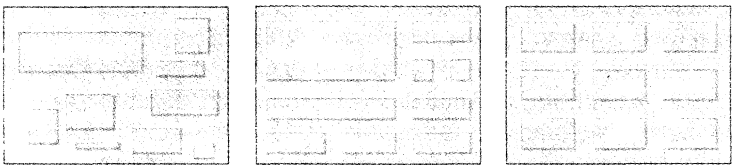
где $\lceil x \rceil$ обозначает потолок от x, то есть наименьшее целое число, большее или равное x.

Для повышения единообразия компоновки нужно соблюдать два условия:

1) визуальные компоненты следует ровно выстроить;

2) различия размеров визуальных компонентов должны быть минимальны.

Пример 10.2. На рис. 10.53 показаны три варианта компоновки диалогового окна. В первом варианте (рис. 10.53, *а*), где компоновка не отличается какой-либо логикой и стройностью, *EK* = 0%. В третьем, абсолютно строгом варианте (рис. 10.53, *в*), *EK* = 100%. Все же более типичным, реалистичным и удачным является второй вариант (рис. 10.53, *б*), для которого единообразие компоновки имеет приемлемое значение *EK* = 82,5%.



а *б* *в*

Рис. 10.53. Три варианта компоновки диалогового окна Команда

Для вычисления метрики *EK* применяют правила, представленные в таблице 10.8.

Таблица 10.8. Правила работы с визуальными компонентами

Описание правила
Визуальным компонентом считается: 1. Любой элемент пользовательского интерфейса. 2. Внешняя метка, встроенная или не встроенная в элемент пользовательского интерфейса. 3. Панель или рамка вокруг любого элемента (или элементов) и любой метки. Обычные линии, отделяющие друг от друга разные части визуального интерфейса, визуальными компонентами не считаются
Визуальные компоненты считаются выровненными в том случае, если невооруженным глазом видно, что их края действительно находятся на одном уровне
Надписи могут сильно варьироваться по высоте. Допустимо небольшое смещение верхней границы надписи вниз по отношению к верхней границе соседнего компонента, а также смещение нижней границы текста чуть вверх по отношению к нижней границе компонента. Текст надписи может быть выровнен по центру по высоте относительно компонента примерно такой же высоты

Для качественных диалоговых окон характерны значения *EK* от 50 до 80%. При выходе за рамки этого диапазона следует принимать специальные меры.

Визуальная связность

Визуальная связность позволяет измерить, насколько близко друг к другу расположены взаимосвязанные компоненты и насколько далеко — никак не связанные. Эта метрика позволяет оценить согласованность расположения визуальных компонентов и их семантических взаимосвязей. В интерфейсах с продуманной структурой должно быть справедливо следующее утверждение: концептуально связанные компоненты группируются вместе. Визуальная связность проверяет истинность этого утверждения, отражая один из важнейших параметров архитектуры пользовательского интерфейса, который существенно влияет на понятность, простоту изучения и использования.

Визуальная связность является расширением обычной метрики связности. Для оценки визуальной связности необходимо определять семантическую связь между любыми двумя визуальными элементами.

Визуальная связность для любой визуальной группы представляет собой просто отношение числа тесно связанных пар визуальных элементов к общему числу рассматриваемых пар. Это отношение вычисляется для самых дальних внутренних визуальных групп, а затем просто повторяется то же самое для других уровней, охватывая, в конечном счете, все пространство взаимодействий. При определении взаимосвязей на внешних уровнях можно сравнивать одни визуальные группы с другими, а также компоненты этих групп.

Суммарная визуальная связность проектного решения вычисляется рекурсивным суммированием связности всех групп, подгрупп и т. д. Для каждого уровня группировки:

$$BC = 100 \cdot \left(\frac{\sum_{\forall k} G_k}{\sum_{\forall k} N_k \cdot (N_k - 1)/2} \right),$$

здесь

$$G_k = \sum_{\forall i,j | i \neq j} R_{i,j},$$

где N_k — число визуальных компонентов группы k, $R_{i,j}$ — семантическая связанность между компонентами i и j в группе k, причем $0 \leq R_{i,j} \leq 1$.

На практике эту формулу упрощают, принимая $R_{i,j}$ за 1, если i и j принадлежат одному семантическому кластеру и обладают, таким образом, высокой связанностью, или за 0 в противном случае. Такая формализация снижает объемы вычислений при работе с сильно связанными парами и, в общем-то, вполне оправдывает себя при оценке реальных проектов. Корректное поведение этой метрики обеспечивается за счет поддержки организации подгрупп компонентов интерфейса, но только при условии, что эти подгруппы имеют смысл, то есть включают в себя компоненты, ассоциированные с кластером взаимосвязанных семантических концепций, а потому и достаточно тесно связанные между собой.

Формирование семантических кластеров проводят с помощью глоссария, объектной модели предметной области, или словаря данных для приложения. Удоб-

нее всего применять объектную модель предметной области, поскольку классы предметной области, их методы и атрибуты определяют общую семантическую организацию приложения.

Аспектно-ориентированное проектирование и программирование

Обычно между требованиями и программными компонентами устанавливаются достаточно сложные отношения. Одно требование может быть реализовано несколькими компонентами, а каждый компонент может включать в себя элементы нескольких требований. На практике это означает, что изменения в требованиях могут повлиять на несколько компонентов.

Аспектно-ориентированный подход реагирует на эти трудности и облегчает сопровождение и повторное использование ПО. Он базируется на абстракциях, названных аспектами, которые реализуют функциональность, востребованную многократно, причем в различных местах системы. Аспекты инкапсулируют функциональность, которая пересекает другую функциональность, включенную в систему. Они используются совместно с объектами и их методами. Исполняемый код аспектно-ориентированной системы создается путем автоматического комбинирования (переплетения) объектов, методов и аспектов.

Главная особенность любого аспекта состоит в том, что он несет в себе пару элементов: описание того, где аспект должен быть включен в систему, а также код, реализующий пересекающее понятие. Вы можете указать, что пересекающий код должен быть включен до или после конкретного вызова метода или запроса атрибута. По сути, аспекты вплетаются в основную программу и приводят к созданию новой расширенной системы.

Основное преимущество аспектно-ориентированного подхода в том, что он поддерживает разделение понятий.

Разделение понятий

Понятие во многом подобно требованию к системе или задаче, которую она должна решать. Производительность является понятием, поскольку пользователь хочет иметь быстрый отклик от системы. Заказчик заинтересован в обеспечении конкретной функциональности системы. Компании, которые оказывают поддержку системы, озабочены легкостью сопровождения. Следовательно, понятие является предметом интереса для целой группы заинтересованных сторон.

Разделение понятий считается ключевым принципом проектирования и программирования ПО. Это означает, что надо организовать систему так, чтобы каждый элемент (класс, метод, компонент) делал что-нибудь одно и только одно. При такой организации можно сосредоточиться на этом элементе без учета других элементов системы. Вы можете изучать каждую часть системы отдельно. Когда необходимо внести изменения, они локализуются на небольшом количестве элементов.

Важность разделения понятий была признана на раннем этапе программирования. Подпрограммы, которые инкапсулируют единицу функциональности, были изобретены в 1950-х. Несколько позже были разработаны такие механизмы структурирования, как процедуры и классы, улучшающие реализацию разделения понятий. Однако у всех этих механизмов имелись проблемы с понятиями, которые

«пересекают» другие понятия. Пересекающие понятия нельзя локализовать с помощью объектов или функций. Для управления этими пересекающими понятиями и были изобретены аспекты.

Например, для информационной системы института основными понятиями являются операции с записями студентов и преподавателей. В дополнение к основным понятиям системы обычно реализуют и вторичные понятия, которые могут быть пересекающими.

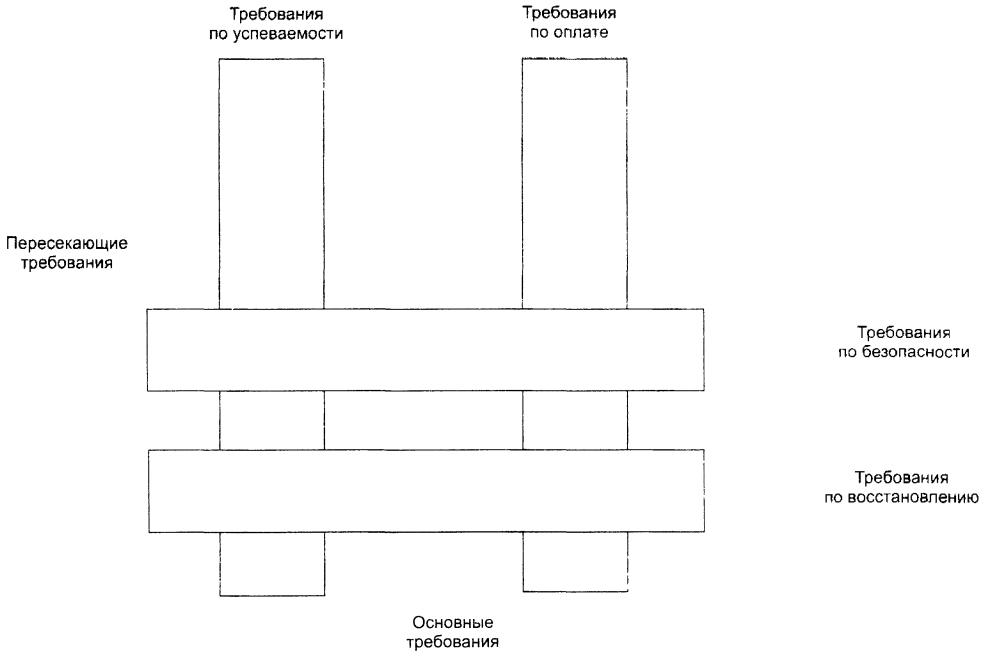


Рис. 10.54. Пересекающие понятия

Пересекающие понятия для информационной системы института показаны на рис. 10.54. Эта система реализует требования, касающиеся отслеживания успеваемости студентов. Она также реализует требования, связанные с оплатой студентов. Все это основные понятия. Однако к системе также предъявляются требования по безопасности и требования по восстановлению (в случае потери данных или сбоя). Такие требования можно назвать пересекающими понятиями, так как они влияют на реализацию всех других требований к системе.

На программном уровне добавление кода реализации пересекающих понятий приводит к двум нежелательным явлениям: спутыванию и разбрасыванию.

Спутывание происходит, когда методы класса включает в себя дополнительный код, который реализует различные требования к системе. Например, во все методы класса Студент добавляются вызовы методов безопасность() и восстановление(), реализующих понятия безопасности и восстановления.

Явление *разбрасывания* возникает при реализации одного понятия, реализации, которая разбрасывается по многим классам системы.

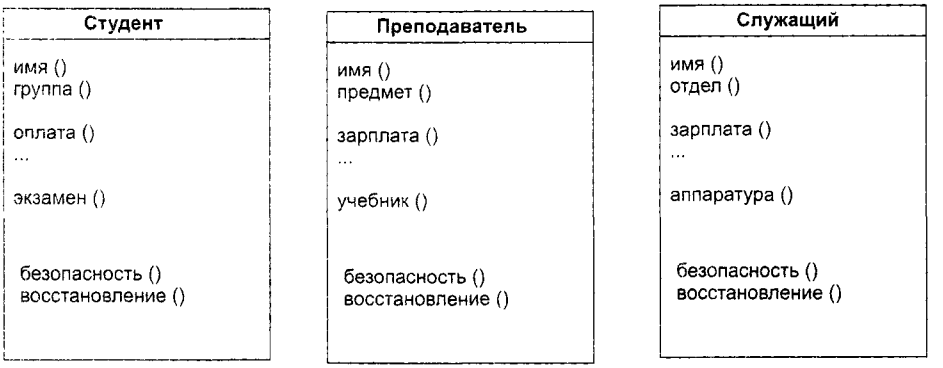


Рис. 10.55. Разбрасывание кода

На рис 10.55 приведен пример трех классов для информационной системы института. Затененная область показывает методы, необходимые для реализации пересекающих понятий безопасности и восстановления. Легко видеть, что реализация этих понятий разбросана по реализациям всех основных понятий системы.

Проблемы с разбрасыванием и спутыванием часто происходят при добавлении новых требований к системе. Представьте, что появляется новое требование сбора статистики по всем сотрудникам институтов. Эта информация находится в разных местах системы, и поэтому потребуется много времени на поиск изменяемых классов, внесение и проверку изменений. Причем всегда существует возможность пропуска некоторого кода. Кроме того, чем больше изменений необходимо сделать, тем больше вероятность внести ошибки в систему.

Основные термины аспектов

При пояснении аспектно-ориентированного подхода воспользуемся терминологией самого популярного языка в этой области — AspectJ. Основные термины представлены в табл. 10.9.

Таблица 10.9. Основные термины аспектно-ориентированного подхода

Термин	Пояснение
Advice – совет	Код, реализующий понятие
Aspect – аспект	Программная абстракция, которая определяет пересекающее понятие. Она включает в себя определение среза и советы, связанные с этим понятием
Join point – точка соединения	Событие в процессе выполнения программы, когда соответствующий аспекту совет может быть выполнен
Join point model – модель точек соединения	Набор событий, на которые ссылаются в срезе
Pointcut – срез	Инструкция, включаемая в аспект, которая определяет точки соединения, где соответствующий совет должен быть выполнен
Weaving – вплетение	Включение ткачом кода совета в указанной точке соединения

Вообразим, что в информационной системе института произошло нарушение безопасности, из-за которого изменился размер зарплаты какого-то преподавателя. Может быть, кто-то случайно оставил компьютер подключенным, и неизвестный человек получил несанкционированный доступ к системе. С другой стороны, служащий, имеющий доступ к системе, может преднамеренно поменять чью-то зарплату. Для исключения подобной возможности вводится новая стратегия безопасности. Перед любым изменением в базе данных пользователь, производящий изменения, должен заново идентифицироваться в системе. Подробная информация о том, кто сделал изменения, также регистрируется в отдельном файле.

Как реализовать эту стратегию?

- *Первый путь.* Изменить содержание метода обновления данных в каждом классе (добавить туда вызов метода проверки подлинности и метода журналирования). Это приводит к спутыванию кода. Логически обновление базы данных, проверка подлинности пользователя и журналирование события являются отдельными, не связанными понятиями. Кому-то захочется проверить подлинность где-то еще без журналирования. Третье лицо пожелает записывать события в журнал без обновления информации в БД.
- *Второй путь.* Можно поступить иначе: перед каждым вызовом метода обновления информации выполнять вызовы проверки подлинности и журналирования, добавив эти методы в каждый класс. Здесь мы получаем разбросанную реализацию.

Лучшим решением этой проблемы является применение аспекта:

```
aspect проверкаПодлинности {
    before: call (public void обновить*(..)) // это срез
    // этот совет вплетается в процесс выполнения
    int попытка = 0;
    string userPassword = Password.Get(попытка);
    while (попытка < 3 && userPassword != thisUser.password()){
        // разрешаются 3 попытки ввода пароля
        попытка = попытка + 1;
        userPassword = Password.Get(попытка); {
    }
    if (userPassword != thisUser.password()) then
        // при неправильной пароле выход из системы
        System.Logout(thisUser.uid);
    public void включить() {
    }    ...

} // проверкаПодлинности
```

Аспекты полностью отличаются от других программных абстракций тем, что аспект включает в себя определение места, где он должен быть выполнен. В других абстракциях, например в методах, существует четкое разделение между определением абстракции и ее использованием. Изучив метод, нельзя сказать, в каком месте он будет вызван. Вызовы могут быть выполнены абсолютно из любого места в программе. Аспекты, напротив, включают определение среза (pointcut), утверждения, задающего то место, где аспект будет вплетен в процесс выполнения.

В данном примере срез является простой инструкцией:

```
before: call (public void обновить*(..))
```

Смысл инструкции в том, что перед выполнением любого метода, имя которого начинается со слова обновить, за которым следует любая другая последовательность символов, должен быть выполнен код в аспекте, который следует после определения среза. Символ звездочка (*) соответствует любой строке символов, разрешенных в идентификаторах. Код, который должен выполняться, известен как «совет» и является реализацией пересекающего понятия. В данном примере совет получает пароль от человека с просьбой об изменении и проверяет его совпадение с паролем пользователя, вошедшего в систему. Если совпадения нет, пользователь выводится из системы и обновление не происходит.

Возможность указывать с помощью среза, где код должен быть выполнен, является отличительной чертой аспектов. Однако чтобы понять, что такое срез, следует разъяснить другую концепцию — идею точек соединения. Точки соединения являются событиями, которые происходят во время выполнения программы: это может быть вызов метода, инициализация переменной, обновление атрибута и т. д.

Существует множество типов событий, происходящих во время выполнения программы. Модель точки соединения определяет набор событий, на которые можно ссылаться в аспектно-ориентированных программах. Модель точки соединения не стандартизована, и каждый аспектно-ориентированный язык программирования предлагает свою модель точки соединения. Например, модель точек соединения языка AspectJ включает в себя:

- События вызова — вызовы метода или конструктора.
- События выполнения — выполнение метода или конструктора.
- События инициализации — инициализация объектов или классов.
- События данных — доступ к атрибуту или модификация атрибута.
- События исключений — обработка исключений.

Срез определяет конкретное событие (события) (например, вызов именованной процедуры), с которым ассоциируется совет. Это означает, что советы можно вплетать в программу в различных случаях, определяемых моделью точки соединения:

- Совет может быть включен до выполнения конкретного метода, списка конкретных методов или списка методов, имена которых соответствуют шаблону (например, обновить*).
- Совет может быть включен после обычного или необычного (по исключению) возврата из метода. Например, точка среза after : returning (public void обновить * (..)) может задавать журналирование после каждого выполнения любого метода обновления.
- Совет может быть включен, когда происходит доступ к атрибуту объекта. Вы можете внедрить совет для трассировки или изменения значения этого атрибута.

Включение совета в точках соединения, описанных в срезе, — задача ткача аспектов. Ткач аспектов — это расширение компиляторов, которое обрабатывает описания аспектов, а также объекты классов и методы, составляющие систему.

Далее ткач создает новую программу, в которой аспекты включены в описанных точках соединения. Аспекты интегрируются в систему так, что пересекающие понятия выполняются в нужных местах конечной системы.

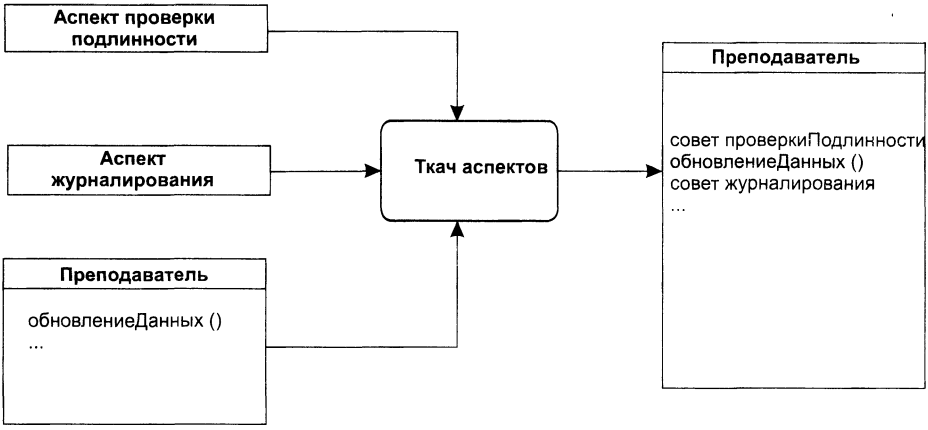


Рис. 10.56. Вплетение аспектов

На рис. 10.56 показано вплетение двух аспектов для реализации проверки подлинности и журналирования в информационной системе института.

Основы компонентной объектной модели

Компонентная объектная модель (COM) — фундамент компонентно-ориентированных средств для всего семейства операционных систем Windows. Рассмотрение этой модели является иллюстрацией комплексного подхода к созданию и использованию компонентного программного обеспечения [17].

COM определяет стандартный механизм, с помощью которого одна часть программного обеспечения предоставляет свои услуги другой части. Общая архитектура предоставления услуг в библиотеках, приложениях, системном и сетевом программном обеспечении позволяет COM изменить подход к созданию программ.

COM устанавливает понятия и правила, необходимые для определения объектов и интерфейсов; кроме того, в ее состав входят программы, реализующие ключевые функции.

В COM любая часть ПО реализует свои услуги с помощью объектов COM. Каждый объект COM поддерживает несколько интерфейсов. Клиенты могут получить доступ к услугам объекта COM только через вызовы операций его интерфейсов — у них нет непосредственного доступа к данным объекта.

Представим объект COM с интерфейсом РаботаСФайлами. Пусть в этот интерфейс входят операции открытьФайл, записатьФайл и закрытьФайл. Если разработчик захочет ввести в объект COM поддержку преобразования форматов, то объекту потребуется еще один интерфейс ПреобразованиеФорматов, возможно, с единственной операцией преобразоватьФормат. Операции каждого из интерфейсов сообща предоставляют связанные друг с другом услуги: либо работу с файлами, либо преобразование их форматов.

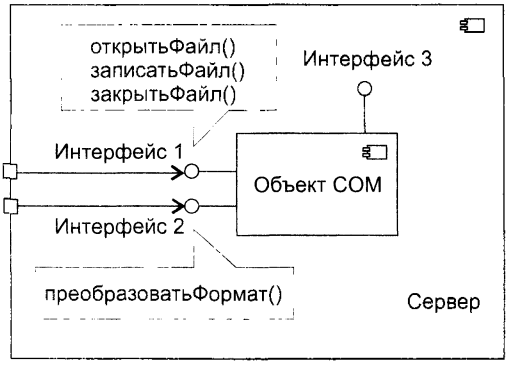


Рис. 10.57. Организация объекта COM

Как показано на рис. 10.57, объект COM всегда реализуется внутри некоторого сервера. Сервер может быть либо динамически подключаемой библиотекой (DLL), подгружаемой во время работы приложения, либо отдельным самостоятельным процессом (EXE).

Для вызова операции интерфейса клиент объекта COM должен получить указатель на его интерфейс. Клиенту требуется отдельный указатель для каждого интерфейса, операции которого он намерен вызывать. Например, как показано на рис. 10.58, клиенту нашего объекта COM нужен один указатель интерфейса для вызова операций интерфейса РаботаСФайлами, а другой — для вызова операций интерфейса ПреобразованиеФорматов.

Получив указатель на нужный интерфейс, клиент может использовать услуги объекта, вызывая операции этого интерфейса. С точки зрения программиста, вызов операции аналогичен вызову локальной процедуры или функции. Но на самом деле исполняемый при этом код может быть частью или библиотеки, или отдельного процесса, или операционной системы (он даже может располагаться на другом компьютере).

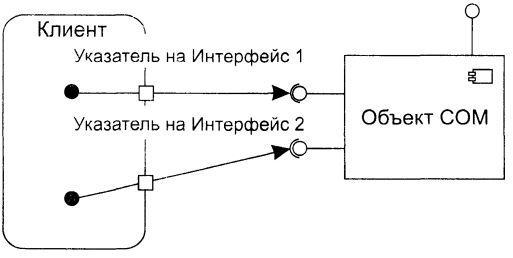


Рис. 10.58. Доступ клиента к интерфейсам объекта COM

Благодаря COM клиентам нет нужды учитывать эти отличия — доступ ко всему осуществляется единообразно. Другими словами, в COM для доступа к услугам, предоставляемым любыми типами ПО, используется одна общая модель.

Организация интерфейса COM

Каждый интерфейс объекта COM — **контракт** между этим объектом и его клиентами. Они обязуются: объект — поддерживать операции интерфейса в точном соответствии с его определениями, а клиент — корректно вызывать операции. Для обеспечения контракта надо задать:

- идентификацию каждого интерфейса;
- описание операций интерфейса;
- реализацию интерфейса.

Идентификация интерфейса

У каждого интерфейса COM два имени. Простое, символьное имя предназначено для людей, оно не уникально (допускается, чтобы это имя было одинаковым у двух интерфейсов). Другое, сложное имя предназначено для использования программами. Программное имя уникально, это позволяет точно идентифицировать интерфейс.

Принято, чтобы символьные имена COM-интерфейсов начинались с буквы I (от Interface). Например, упомянутый нами интерфейс для работы с файлами должен называться IРаботаСФайлами, а интерфейс преобразования их форматов – IПреобразованиеФорматов.

Программное имя любого интерфейса образуется с помощью глобально уникального идентификатора (globally unique identifier — GUID). GUID интерфейса считается идентификатором интерфейса (interface identifier — IID). GUID — это 16-байтовая величина (128-битовое число), генерируемая автоматически.

Уникальность во времени достигается за счет включения в каждый GUID метки времени, указателя момента создания. Уникальность в пространстве обеспечивается цифровыми параметрами компьютера, который использовался для генерации GUID.

Описание интерфейса

Для определения интерфейсов применяют специальный язык — язык описания интерфейсов (Interface Definition Language — IDL). Например, IDL-описание интерфейса для работы с файлами IРаботаСФайлами имеет вид:

```
[object,
    uuid(E7CDODOO-1827-11CF-9946-444553540000) ]
interface IРаботаСФайлами: IUnknown {
    import "unknwn.idl"
    HRESULT открытьФайл ([in] OLECHAR имя [31]);
    HRESULT записатьФайл ([in] OLECHAR имя [31]);
    HRESULT закрытьФайл ([in] OLECHAR имя [31]);
}
```

Описание интерфейса начинается со слова object, отмечающего, что будут использоваться расширения, добавленные COM к оригинальному IDL. Далее записывается программное имя (IID интерфейса), оно начинается с ключевого слова uuid (Universal Unique Identifier — универсально уникальный идентификатор). UUID является синонимом термина GUID.

В третьей строке записывается имя интерфейса — IРаботаСФайлами, за ним — двоеточие и имя другого интерфейса — IUnknown. Такая запись означает, что интерфейс IРаботаСФайлами наследует все операции, определенные для интерфейса IUnknown, то есть клиент, у которого есть указатель на интерфейс IРаботаСФайлами, может вызывать и операции интерфейса IUnknown. IUnknown является главным интерфейсом для COM, все остальные интерфейсы — его наследники.

В четвертой строке указывается оператор import. Поскольку данный интерфейс наследует от IUnknown, может потребоваться IDL-описание для IUnknown. Аргумент оператора import указывает, в каком файле находится нужное описание.

Ниже в описании интерфейса приводятся имена и параметры трех операций — открытьФайл(), записатьФайл() и закрытьФайл(). Все они возвращают величину типа HRESULT, указывающую корректность обработки вызова. Для каждого параметра в IDL приводится направление передачи (в данном примере in), тип и название.

Считается, что такого описания достаточно для заключения контракта между объектом COM и его клиентом.

По правилам COM запрещается любое изменение интерфейса (после его публикации). Для реализации изменений нужно вводить новый интерфейс. Такой интерфейс может быть наследником старого интерфейса, но отличен от него и имеет другое уникальное имя.

Значение правила запрета на изменение интерфейса трудно переоценить. Оно — залог стабильности работы в среде, где множество клиентов взаимодействует с множеством COM-объектов и где независимая модификация COM-объектов — обычное дело. Именно это правило позволяет клиентам старых версий не пострадать при введении новых версий COM-объектов. Новая версия обязана поддерживать и старый COM-интерфейс.

Реализация интерфейса

COM задает стандартный двоичный формат, который должен реализовать каждый COM-объект и для каждого интерфейса. Стандарт гарантирует, что любой клиент может вызывать операции любого объекта, причем независимо от языков программирования, на которых написаны клиент и объект.

Структуру интерфейса IРаботаСФайлами, соответствующую двоичному формату, поясняет рис. 10.59.

Внешний указатель на интерфейс (указатель клиента) ссылается на внутренний указатель объекта COM. Внутренний указатель — это адрес виртуальной таблицы. Виртуальная таблица содержит указатели на все операции интерфейса.

Первые три элемента виртуальной таблицы являются указателями на операции, унаследованные от интерфейса IUnknown. Видно, что на собственные операции интерфейса IРаботаСФайлами указывают 4, 5 и 6 элементы виртуальной таблицы. Такая ситуация типична для любого COM-интерфейса.

Обработка клиентского вызова выполняется в следующем порядке:

- с помощью указателя на виртуальную таблицу извлекается указатель на требуемую операцию интерфейса;
- указатель на операцию обеспечивает доступ к ее реализации;
- исполнение кода операции обеспечивает требуемую услугу.

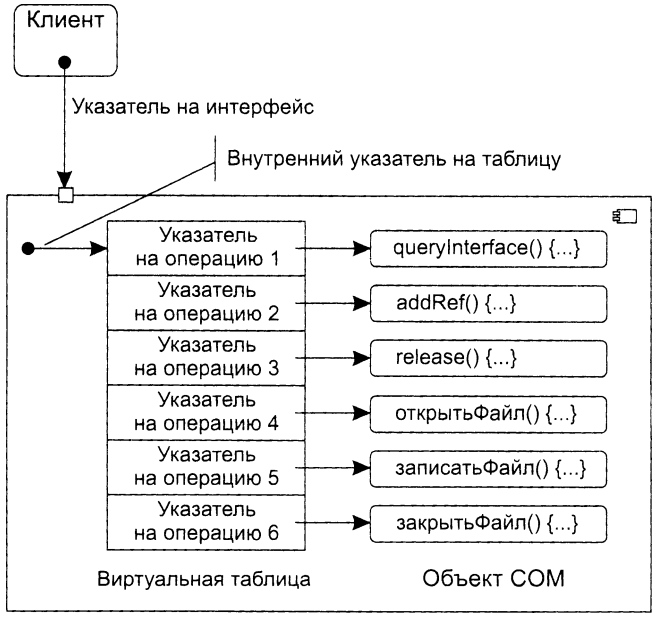


Рис. 10.59. Внутренняя структура интерфейса IРаботаСФайлами

IUnknown — базовый интерфейс COM

Интерфейс IUnknown обеспечивает минимальное «снаряжение» каждого объекта COM. Он содержит три операции и предоставляет любому объекту COM две функциональные возможности:

- операция queryInterface() позволяет клиенту получить указатель на любой интерфейс объекта (из другого указателя интерфейса);
- операции addRef() и release() обеспечивают механизм управления временем жизни объекта.

Свой первый указатель на интерфейс объекта клиент получает при создании объекта COM. Порядок получения других указателей на интерфейсы (для вызова их операций) поясняет рис. 10.60, где расписаны три шага работы. В качестве параметра операции queryInterface() задается идентификатор требуемого интерфейса (IID). Если требуемый интерфейс отсутствует, операция возвращает значение NULL.

Имеет смысл отметить и второе важное достоинство операции queryInterface(). В сочетании с требованием неизменности COM-интерфейсов она позволяет «убить двух зайцев»:

- развивать компоненты;
- обеспечивать стабильность клиентов, использующих компоненты.

Поясним это утверждение. По законам COM-этики новый COM-объект должен нести в себе и старый COM-интерфейс, а операция queryInterface() всегда обеспечит доступ к нему.

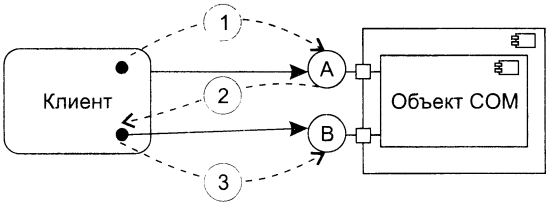


Рис. 10.60. Получение указателя на интерфейс с помощью queryInterface. С помощью указателя на интерфейс A клиент запрашивает указатель на интерфейс B, вызывая queryInterface (IID_B) (*1*); объект возвращает указатель на интерфейс B (*2*); теперь клиент может вызывать операции из интерфейса B (*3*)

А теперь обсудим правила жизни, а точнее смерти COM-объекта. В многоликой COM-среде бремя ответственности за решение вопроса о финализации должно лежать как на клиенте, так и на COM-объекте. Можно сказать, что фирма Microsoft (создатель этой модели) разработала самурайский кодекс поведения COM-объекта — он должен сам себя уничтожить. Возникает вопрос — когда? Когда он перестанет быть нужным всем своим клиентам, когда вытечет песок из часов его жизни. Роль песочных часов играет счетчик ссылок (СЧС) COM-объекта.

Правила финализации COM-объекта очень просты:

- при выдаче клиенту указателя на интерфейс выполняется СЧС+1;
- при вызове операции addRef() выполняется СЧС+1;
- при вызове операции release() выполняется СЧС–1;
- при СЧС = 0 объект уничтожает себя.

Конечно, клиент должен помогать достойному харакири объекта-самурая:

- при получении от другого клиента указателя на интерфейс COM-объекта он должен вызвать в этом объекте операцию addRef();
- в конце работы с объектом он обязан вызвать его операцию release().

Серверы COM-объектов

Каждый COM-объект существует внутри конкретного сервера. Этот сервер содержит программный код реализации операций, а также данные активного COM-объекта. Один сервер может обеспечивать несколько объектов и даже несколько COM-классов. Как показано на рис. 10.61, используются три типа серверов:

- Сервер «в процессе» (in-process) — объекты находятся в динамически подключаемой библиотеке и, следовательно, выполняются в том же процессе, что и клиент;
- Локальный сервер (out-process) — объекты находятся в отдельном процессе, выполняющемся на том же компьютере, что и клиент;
- Удаленный сервер — объекты находятся в DLL или в отдельном процессе, которые расположены на удаленном от клиента компьютере.

С точки зрения логики клиенту безразлично, в сервере какого типа находится COM-объект — создание объекта, получение указателя на его интерфейсы, вызов его операций и финализация выполняются всегда одинаково. Хотя временные затраты на организацию взаимодействия в каждом из трех случаев, конечно, отличаются друг от друга.

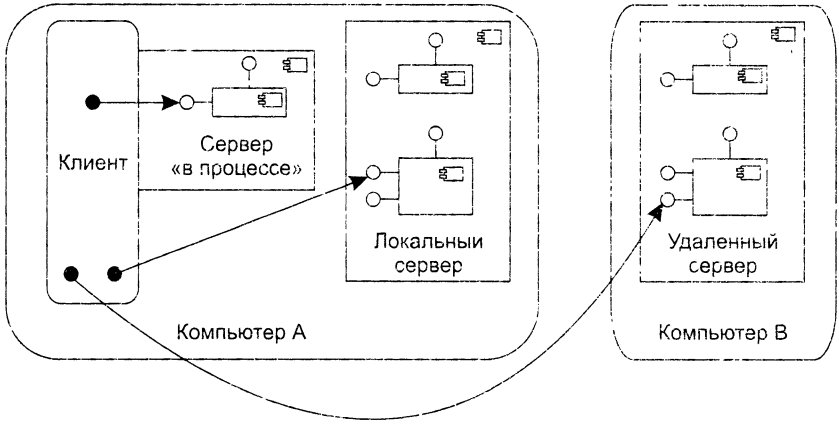


Рис. 10.61. Различные серверы COM-объектов

Преимущества COM

В качестве кратких выводов отметим основные преимущества COM:

1. COM обеспечивает удобный способ фиксации услуг, предоставляемых разными фрагментами ПО.
2. Общий подход к созданию всех типов программных услуг в COM упрощает проблемы разработки.
3. COM безразличен язык программирования, на котором пишутся COM-объекты и клиенты.
4. COM обеспечивает эффективное управление изменением программ — замену текущей версии компонента на новую версию с дополнительными возможностями.

Работа с COM-объектами

При работе с COM-объектами приходится их создавать, повторно использовать, размещать в других процессах, описывать в библиотеке операционной системы. Рассмотрим каждый из этих вопросов.

Создание COM-объектов

Создание COM-объекта базируется на использовании функций библиотеки COM Библиотека COM:

- содержит функции, предлагающие базовые услуги объектам и их клиентам;
- предоставляет клиентам возможность запуска серверов COM-объектов.

Доступ к услугам библиотеки COM выполняется с помощью вызовов обычных функций. Чаще всего имена функций библиотеки COM начинаются с префикса «со». Например, в библиотеке имеется функция coCreateInstance().

Для создания COM-объекта клиент вызывает функцию библиотеки COM coCreateInstance(). В качестве параметров этой функции посылаются идентификатор класса объекта CLSID и IID интерфейса, поддерживаемого объектом. С помощью CLSID библиотека ищет сервер класса (это делает диспетчер управления сервисами SCM — Service Control Manager). Поиск производится в системном реестре (Registry), отображающем CLSID в адрес исполняемого кода сервера. В системном реестре должны быть зарегистрированы классы всех COM-объектов.

Закончив поиск, библиотека COM запускает сервер класса. В результате создается неинициализированный COM-объект, то есть объект, данные которого не определены. Описанный процесс иллюстрирует рис. 10.62.

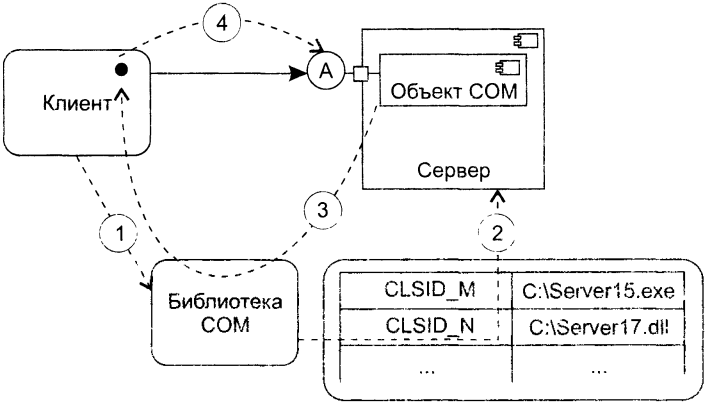


Рис. 10.62. Создание одиночного COM-объекта. Клиент вызывает coCreateInstance (CLSID M, IID A) (*1*); библиотека COM находит сервер и запускает его (*2*); библиотека COM возвращает указатель на интерфейс A (*3*); теперь клиент может вызывать операции COM-объекта (*4*)

Как правило, после получения указателя на созданный COM-объект клиент предлагает объекту самоинициализироваться, то есть загрузить себя конкретными значениями данных. Эту процедуру обеспечивают стандартные COM-интерфейсы IPersistFile, IPersistStorage и IPersistStream.

Параметры функции coCreateInstance(), используемой клиентом, позволяют также задать тип сервера, который нужно запустить (например, «в процессе» или локальный).

В более общем случае клиент может создать несколько COM-объектов одного и того же класса. Для этого клиент использует фабрику класса (class factory) — COM-объект, способный генерировать объекты одного конкретного класса.

Фабрика класса поддерживает интерфейс IClassFactory, включающий две операции. Операция createInstance() создает COM-объект — экземпляр конкретного класса, имеет параметр — идентификатор интерфейса, указатель на который надо

вернуть клиенту. Операция lockServer() позволяет сохранять сервер фабрики загруженным в память.

Клиент вызывает фабрику с помощью функции библиотеки COM coGetClassObject():

```
coGetClassObject (<CLSID создаваемого объекта>, < IID интерфейса IClassFactory>)
```

В качестве третьего параметра функции можно задать тип запускаемого сервера.

Библиотека COM запускает фабрику класса и возвращает указатель на интерфейс IClassFactory этой фабрики. Дальнейший порядок работы с помощью фабрики иллюстрирует рис. 10.63.

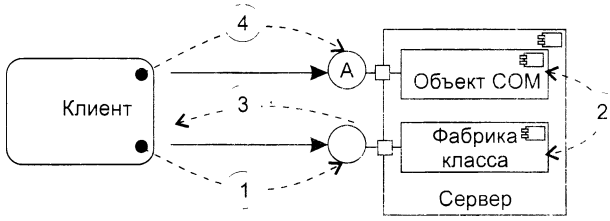


Рис. 10.63. Создание COM-объекта с помощью фабрики класса. Клиент вызывает IClassFactory :: createInstance (IID A) (*1*); фабрика класса создает COM-объект и получает указатель на его интерфейс (*2*); фабрика класса возвращает указатель на интерфейс A COM-объекта (*3*); теперь клиент может вызывать операции COM-объекта (*4*)

Клиент вызывает операцию IClassFactory::createInstance() фабрики, в качестве параметра которой задает идентификатор необходимого интерфейса объекта (IID). В ответ фабрика класса создает COM-объект и возвращает указатель на заданный интерфейс. Теперь клиент применяет возвращенный указатель для вызова операций COM-объекта.

Очень часто возникает следующая ситуация — существующий COM-класс заменили другим, поддерживающим как старые, так и дополнительные интерфейсы и имеющим другой CLSID. Появляется задача — обеспечить использование нового COM-класса старыми клиентами. Обычным решением является запись в системный реестр соответствия между старым и новым CLSID. Запись выполняется с помощью библиотечной функции coTreatAsClass():

```
coTreatAsClass (<старый CLSID>, <новый CLSID>)
```

Повторное использование COM-объектов

Известно, что основным средством повторного использования существующего кода является наследование реализации (новый класс наследует реализацию операций существующего класса). COM не поддерживает это средство. Причина — в типовой COM-среде базовые объекты и объекты-наследники создаются, выпускаются и обновляются независимо. В этих условиях изменения базовых объектов могут вызвать непредсказуемые последствия в объектах-наследниках их реализации. COM предлагает другие средства повторного использования: включение и агрегирование

Применяются следующие термины:

- внутренний объект — это базовый объект;
- внешний объект — это объект, повторно использующий услуги внутреннего объекта.

При включении (делегировании) внешний объект является обычным клиентом внутреннего объекта. Как показано на рис. 10.64, когда клиент вызывает операцию внешнего объекта, эта операция, в свою очередь, вызывает операцию внутреннего объекта.

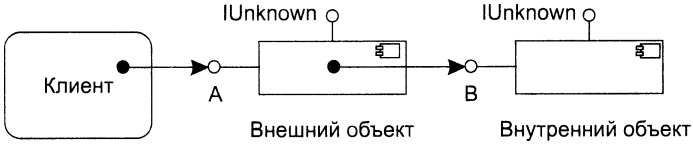


Рис. 10.64. Повторное использование COM-объекта с помощью включения

При этом внутренний объект ничего не замечает.

Достоинство включения — простота. Недостаток — низкая эффективность при длинной цепочке «делегирующих» объектов.

Недостаток включения устраняет агрегирование. Оно позволяет внешнему объекту обманывать клиентов — представлять в качестве собственных интерфейсы, реализованные внутренним объектом. Как показано на рис. 10.65, когда клиент запрашивает у внешнего объекта указатель на такой интерфейс, ему возвращается указатель на внутренний, агрегированный интерфейс. Клиент об агрегировании ничего не знает, зато внутренний объект обязательно должен знать о том, что он агрегирован.

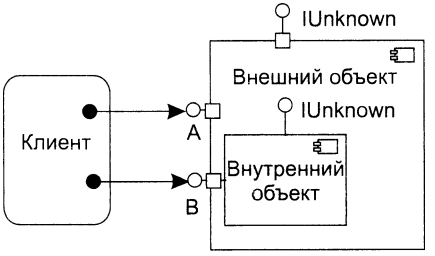


Рис. 10.65. Повторное использование COM-объекта с помощью агрегирования

Зачем требуется такое знание? В чем причина? Ответ состоит в необходимости особой реализации внутреннего объекта. Она должна обеспечить правильный подсчет ссылок и корректную работу операции queryInterface().

Представим две практические задачи:

- запрос клиентом у внутреннего объекта (с помощью операции queryInterface()) указателя на интерфейс внешнего объекта;
- изменение клиентом счетчика ссылок внутреннего объекта (с помощью операции addRef()) и информирование об этом внешнего объекта.

Ясно, что при автономном и независимом внутреннем объекте их решить нельзя. В противном же случае решение элементарно — внутренний объект должен отказаться от собственного интерфейса IUnknown и применять только операции IUnknown внешнего объекта. Иными словами, адрес собственного IUnknown должен быть замещен адресом IUnknown агрегирующего объекта. Это указывается при создании внутреннего объекта (за счет добавления адреса как параметра операции coCreateInstance() или операции IClassFactory::createInstance()).

Маршалинг

Клиент может содержать прямую ссылку на COM-объект только в одном случае: когда COM-объект размещен в сервере «в процессе». В случае локального или удаленного сервера, как показано на рис. 10.66, он ссылается на посредника.

Посредник — COM-объект, размещенный в клиентском процессе и предоставляющий клиенту те же интерфейсы, что и запрашиваемый объект. Запрос клиентом операции через такую ссылку приводит к исполнению кода посредника.

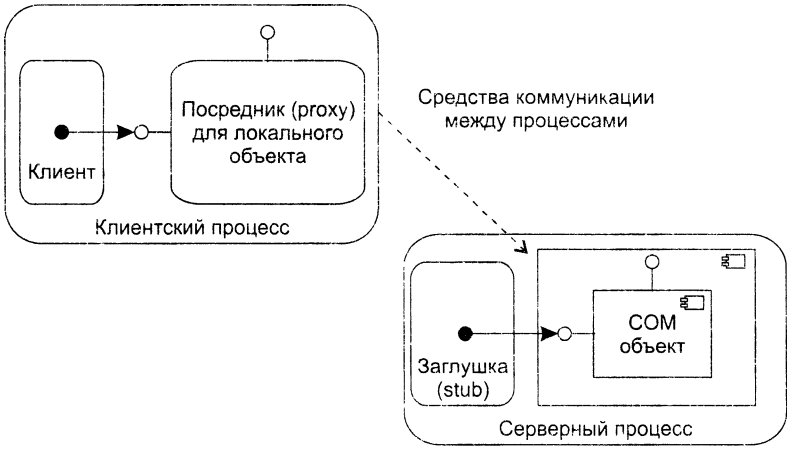


Рис. 10.66. Организация маршалинга и демаршалинга

Посредник принимает параметры, переданные клиентом, и упаковывает их для дальнейшей пересылки. Эта процедура называется *маршалингом*. Затем посредник (с помощью средства коммуникации) посылает запрос в процесс, который на самом деле реализует COM-объект.

По прибытии в процесс локального сервера запрос передается заглушке. Заглушка распаковывает параметры запроса и вызывает операцию COM-объекта. Эта процедура называется *демаршалингом*. После завершения COM-операции результаты возвращаются в обратном направлении.

Код посредника и заглушки автоматически генерируется компилятором MIDL (Microsoft IDL) по IDL-описанию интерфейса.

IDL-описание и библиотека типа

Помимо информации об интерфейсах, IDL-описание может содержать информацию о библиотеке типа.

Библиотека типа определяет важные для клиента характеристики COM-объекта: имя его класса, поддерживаемые интерфейсы, имена и адреса элементов интерфейса.

Рассмотрим пример приведенного ниже IDL-описания объекта для работы с файлами. Оно состоит из трех частей. Первые две части описывают интерфейсы IРаботаСФайлами и IПреобразованиеФорматов, третья часть — библиотеку типа ФайлыБибл. По первым двум частям компилятор MIDL генерирует код посредников и заглушек, по третьей части — код библиотеки типа.

```
---------- 1-я часть
[object,
     uuid(E7CD0D00-1827-11CF-9946-444553540000) ]
interface IРаботаСФайлами: IUnknown
    {    import "unknwn.idl"
         HRESULT открытьФайл ([in] OLECHAR имя[31]);
         HRESULT записатьФайл ([in] OLECHAR имя[31]);
         HRESULT закрытьФайл ([in] OLECHAR имя[31]);
    }
---------- 2-я часть
[object,
     uuid(5FBDD020-1863-11CF-9946-444553540000) ]
interface IПреобразованиеФорматов: IUnknown
    {    HRESULT преобразоватьФормат ([in] OLECHAR имя[31],
                                      [in] OLECHAR формат[31]);
    }
---------- 3-я часть
[uuid(B253E460-1826-11CF-9946-444553540000),
        version (1.0)]
library ФайлыБибл
{    importlib ("stdole32.tlb");
[uuid(B2ECFAA0-1827-11CF-9946-444553540000) ]
coClass СоФайлы
    {     interface IРаботаСФайлами;
          interface IПреобразованиеФорматов;
    }
}
```

Описание библиотеки типа начинается с ее уникального имени (записывается после служебного слова uuid), затем указывается номер версии библиотеки.

После служебного слова library записывается символьное имя библиотеки (ФайлыБибл).

Далее в операторе importlib указывается файл со стандартными определениями IDL — stdole32.tlb.

Тело описания библиотеки включает только один элемент — COM-класс (coClass), на основе которого создается COM-объект.

В начале описания COM-класса приводится его уникальное имя (это и есть идентификатор класса — CLSID), затем символьное имя — СоФайлы. В теле класса перечислены имена поддерживаемых интерфейсов — IРаботаСФайлами и IПреобразованиеФорматов.

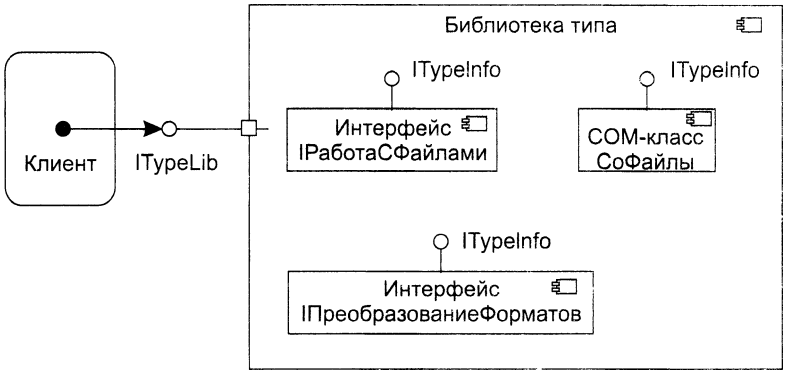


Рис. 10.67. Доступ к библиотеке типа

Как показано на рис. 10.67, доступ к библиотеке типа выполняется по стандартному интерфейсу ITypeLib, а доступ к отдельным элементам библиотеки — по интерфейсу ITypeInfo.

Развертывание программной системы на аппаратных средствах

Распределение элементов (результатов разработки) программного приложения по аппаратным узлам компьютерной системы удобно показать с помощью диаграмм развертывания (deployment diagram). В русской литературе их называют также диаграммами размещения. Диаграмма развертывания позволяет изобразить «физическую» структуру программной системы. На этапе проектирования диаграмма развертывания используется для представления физической совокупности узлов как основы для реализации системы.

Диаграммы развертывания состоят из трех основных элементов: артефактов, узлов и отношений между ними.

Артефакты

Артефакт — это физический элемент, часть программной системы. Как правило, он является исполняемой программой, но может быть также текстовым файлом, документом, сценарием или другим элементом системы, связанным с программным кодом. Артефакт может быть манифестацией (реализацией) одного или нескольких компонентов. Между артефактами возможны отношения зависимости или композиции.

Артефакт изображается в виде прямоугольника, содержащего его имя, метку со стереотипом «artifact» (рис. 10.68). Возможно также присутствие пиктограммы в виде листка бумаги (в правом верхнему углу). На рисунке также показано, что с помощью артефакта ИнтерфейсПользователя.с реализован компонент ИнтерфейсКассы. Для этого использовано отношение манифестации (зависимость со стереотипом <<manifest>>).

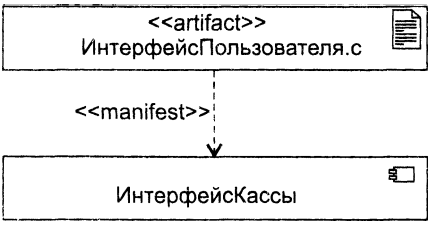


Рис. 10.68. Пример артефакта и реализуемого им компонента

Мир артефактов достаточно широк и разнообразен. В языке UML для обозначения новых разновидностей артефактов применяют механизм стереотипов. Стандартные стереотипы UML для артефактов представлены в табл. 10.10.

Таблица 10.10. Разновидности артефактов

Стереотип	Описание
<<executable>>	Программный файл, который может выполняться в компьютерной системе (имеет расширение .exe)
<<library>>	Статическая или динамическая объектная библиотека (имеет расширение .dll)
<<file>>	Физический файл в контексте разработанной системы
<<source>>	Исходный файл, который можно откомпилировать в исполняемый файл
<<script>>	Скриптовой файл, который может интерпретироваться компьютерной системой
<<document>>	Обобщенный файл, который не является исходным или исполняемым файлом

В языке UML не определены пиктограммы для перечисленных стереотипов, применяемая на практике пиктограмма для библиотеки показана на рис. 10.69.

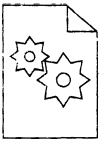


Рис. 10.69. Пиктограмма объектной библиотеки

Узлы

Узел — физический элемент, который существует в период работы программной системы и представляет собой компьютерный ресурс, имеющий память, а возможно, и способность обработки. Графически узел изображается как куб с именем (рис. 10.70).

Развертывание — это размещение артефакта или набора артефактов на узле для выполнения.

Рис. 10.70. Обозначение узла

Узел является тем местом, где физически размещаются артефакты, он играет роль квартиры для артефактов (рис. 10.71). Как видим, развертывание изображается графически путем вложения символов артефактов в символ узла.

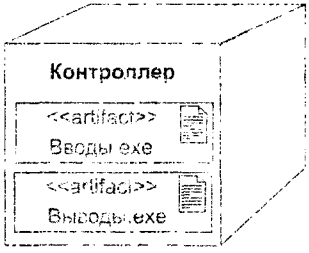


Рис. 10.71. Размещение артефактов внутри узла

Если артефактов в узле достаточно много, можно выбрать альтернативный способ: нарисовать пунктирные стрелки, идущие от символов артефактов к символу размещающего узла. Стрелки помечаются стереотипом «deploy» (рис. 10.72).

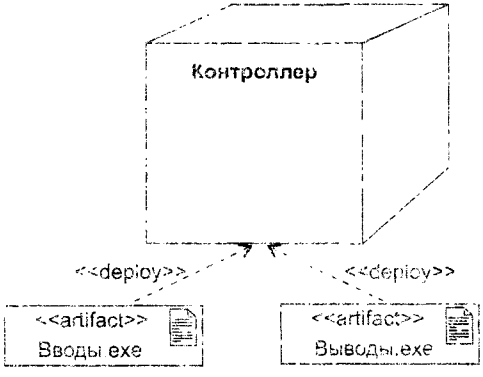


Рис. 10.72. Развертывание артефактов в узле

Узел, являющийся вычислительным ресурсом с возможностями процессорной обработки, может помечаться стереотипом <<device>>

Узлы могут быть представлены в качестве типов или экземпляров. Иначе говоря, развертывание показывается на уровне типов или на уровне экземпляров (применяют имена без подчеркивания и с подчеркиванием соответственно). Внутри символов, обозначающих типы узлов, находятся типы таких артефактов, которые можно размещать на экземплярах соответствующих узлов. Внутри экземпляров

узлов показываются экземпляры артефактов (что означает нахождение экземпляра артефакта на экземпляре узла).

На рис. 10.71 и 10.72 изображался тип Контроллера. Изображение конкретного экземпляра, принадлежащего этому типу, представлено на рис. 10.73.

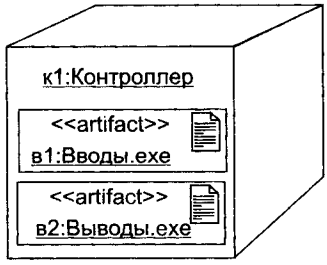


Рис. 10.73. Экземпляр узла

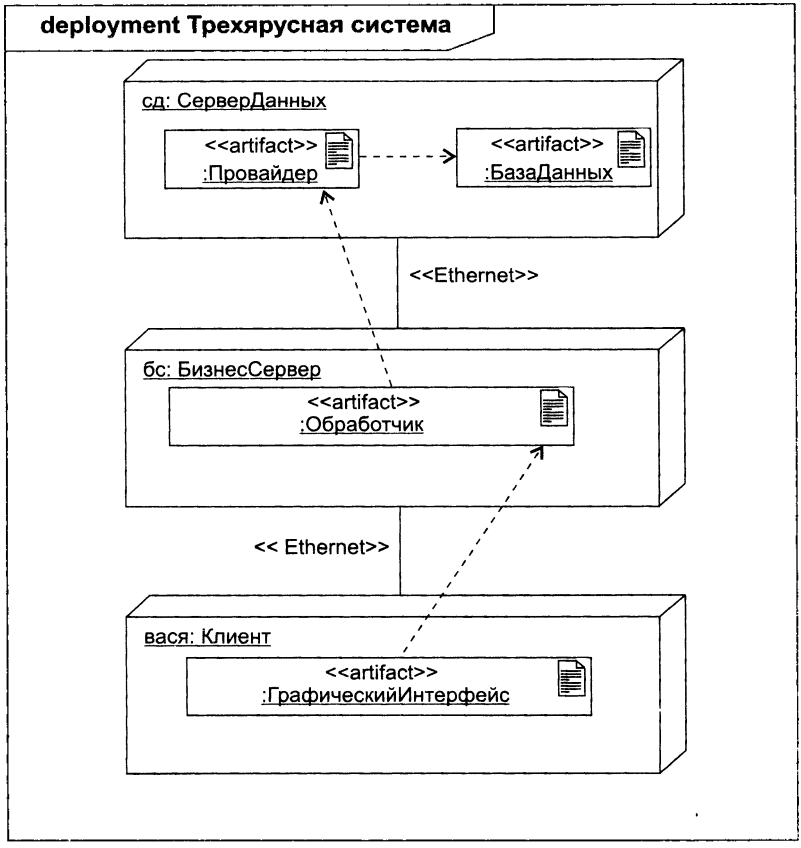


Рис. 10.74. Диаграмма развертывания трехъярусной системы

Узлы могут быть соединены друг с другом отношением ассоциации. Ассоциация между узлами обозначает коммуникационный маршрут. Природа маршрута должна быть описана при помощи стереотипа (например, задается тип канала или тип сети).

Диаграммы развертывания

Графически диаграмма развертывания – это граф из узлов (или экземпляров узлов), соединенных ассоциациями, которые показывают существующие коммуникации. Узлы (или их экземпляры) могут содержать артефакты (или их экземпляры), живущие и/или работающие в узлах. Как показано на рис. 10.74, артефакты соединяются друг с другом пунктирными стрелками зависимостей.

На этой диаграмме изображена типовая трехъярусная система:

- ярус базы данных реализован экземпляром **сд** узла **СерверДанных**;
- ярус бизнес-логики представлен экземпляром **бс** узла **БизнесСервер**;
- ярус графического интерфейса пользователя образован экземпляром **вася** узла **Клиент**.

В экземпляре сервера данных показано размещение анонимного экземпляра артефакта **Провайдер** и анонимного экземпляра артефакта **БазаДанных**. Экземпляр узла бизнес-сервера содержит анонимный экземпляр артефакта **Обработчик**, а экземпляр узла клиента – анонимный экземпляр артефакта **ГрафическийИнтерфейс**. В диаграмме явно отображены три зависимости между экземплярами артефактов. С помощью стереотипов заданы характеристики физических соединений между ярусами: все они определены как **Ethernet**-соединения.

Контрольные вопросы и упражнения

1. Дайте определение пакета.
2. Какие характеристики пакета вы знаете?
3. В чем заключается разница в изображении пустого и заполненного пакета.
4. Что такое отношение владения для пакета?
5. Какие уровни видимости в пакете вы знаете?
6. Возможна ли в пакете защищенная видимость? Ответ обоснуйте.
7. В чем суть механизма импортирования пакетов?
8. Охарактеризуйте правила видимости элементов пакета. Приведите примеры поясняющие их особенности.
9. Поясните понятие компонента и его интерфейса.
10. Какие разновидности интерфейса вы знаете?
11. Какой смысл имеет понятие «порт» в компоненте?
12. Что является частью пакета? Как она именуется?
13. Сравните характеристики пакета и компонента. В чем они схожи? Чем отличаются друг от друга?
14. Какие секции входят в графическое обозначение класса?

15. Какие секции класса можно не показывать?
16. Какие имеются разновидности области действия атрибута (операции)?
17. Поясните общий синтаксис представления атрибута.
18. Какие уровни видимости вы знаете? Их смысл?
19. Какие свойства атрибутов вам известны?
20. Поясните общий синтаксис представления операции.
21. Какой вид имеет форма представления параметра операции?
22. Какие свойства операций вам известны?
23. Что означают три точки в списке атрибутов (операций)?
24. Как организуется группировка атрибутов (операций)?
25. Как ограничить количество экземпляров класса?
26. Перечислите известные вам «украшения» отношения ассоциации.
27. Может ли структурная модель программной системы не иметь отношений ассоциации?
28. Какой смысл имеет квалификатор? К чему он относится?
29. Какие отношения могут иметь пометки видимости? Что эти пометки обозначают?
30. Какой смысл имеет класс-ассоциация?
31. Чем отличается агрегация от композиции? Разновидностями какого отношения (в UML) они являются?
32. Что обозначает в UML простая зависимость?
33. Какой смысл имеет отношение обобщения?
34. Какие недостатки у множественного наследования?
35. Перечислите недостатки ромбовидной решетки наследования.
36. В чем смысл отношения реализации?
37. Что обозначает мощность «многие-ко-многим» и в каких отношениях она применяется?
38. Что такое абстрактный класс (операция) и как он (она) отображается?
39. Как запретить полиморфизм операции?
40. Как обозначить корневой класс?
41. Используя результаты упражнения 29 девятой главы, составьте диаграмму классов для банкомата.
42. Преобразуйте диаграмму коммуникации упражнения 30 девятой главы в диаграмму последовательности. По построенной диаграмме последовательности спроектируйте диаграмму классов.
43. По диаграмме последовательности из упражнения 31 девятой главы спроектируйте диаграмму классов.
44. Дайте развернутую характеристику четырех принципов детального проектирования. Каких недостатков можно избежать при их применении?

45. Охарактеризуйте принципы упаковки классов. Насколько они согласованы?
46. Каково назначение кооперации? Какие составляющие ее образуют?
47. Могут ли разные кооперации использовать одинаковые классы? Обоснуйте ответ.
48. Что такое паттерн?
49. Чем паттерн отличается от кооперации? Чем они схожи?
50. Как описывается паттерн?
51. Что нужно сделать для применения паттерна?
52. Поясните суть мышления в терминах паттернов.
53. Дайте развернутую характеристику шагов паттерн-ориентированного проектирования. Поясните эти шаги на конкретных примерах.
54. Опишите принципы проектирования пользовательского интерфейса.
55. Какие метрики позволяют оценить качество пользовательского интерфейса? Свяжите эти метрики с принципами проектирования интерфейса.
56. В чем смысл разделения понятий?
57. Что такое спутывание и разбрасывание кода? Поясните на примерах.
58. Поясните внутреннюю структуру аспекта.
59. Чем отличается точка соединения от среза?
60. Какая разница между сеткой соединения и моделью точек соединения?
61. Какую выгоду приносят аспекты?
62. Всегда ли нужно применять аспекты?
63. Какие трудности в применении аспектов вы видите?
64. Создайте аспект для модели банкомата, рассмотренной в девятой главе.
65. Каково назначение COM? Какие преимущества дает использование COM?
66. Чем COM-объект отличается от обычного объекта?
67. Что должен иметь клиент для использования операции COM-объекта?
68. Как идентифицируется COM-интерфейс?
69. Как описывается COM-интерфейс?
70. Как реализуется COM-интерфейс?
71. Чего нельзя делать с COM-интерфейсом? Обоснуйте ответ.
72. Объясните назначение и применение операции queryInterface.
73. Объясните назначение и применение операций addRef и release.
74. Что такое сервер COM-объекта? Какие типы серверов вы знаете?
75. В чем назначение библиотеки COM?
76. Как создается одиночный COM-объект?
77. Как создаются несколько COM-объектов одного и того же класса?
78. Как обеспечить использование нового COM-класса старыми клиентами?
79. В чем состоят особенности повторного использования COM-объектов?

80. Какие требования предъявляет агрегация к внутреннему COM-объекту?
81. Что такое маршалинг и демаршалинг?
82. Поясните назначение посредника и заглушки.
83. Зачем нужна библиотека типа? Как она описывается?
84. Какие вершины и ребра образуют диаграмму развертывания?
85. Чем отличается узел от артефакта?
86. Чем отличается экземпляр узла от типа узла?
87. Как применяют диаграммы развертывания?
88. Когда целесообразно применять типы узлов?

Глава 12

Метрики объектно-ориентированных программных систем

При разработке объектно-ориентированных программных систем значительная часть затрат приходится на создание визуальных моделей. Очень важно корректно и всесторонне оценить качество этих моделей, используя для этого числовые оценки. Решение данной задачи требует введения специального метрического аппарата. Такой аппарат развивает идеи классического оценивания сложных программных систем, основанного на метриках сложности, связности и сцепления. Вместе с тем он учитывает специфические особенности объектно-ориентированных решений. В этой главе обсуждаются наиболее известные объектно-ориентированные метрики, а также описывается методика их применения.

Метрические особенности объектно-ориентированных программных систем

Объектно-ориентированные метрики вводятся с целью:

- улучшить понимание качества продукта;
- оценить эффективность процесса разработки;
- улучшить качество работы на этапе проектирования.

Все эти цели важны, но для программного инженера главная цель — повышение качества продукта. Возникает вопрос: как измерить качество объектно-ориентированной системы?

Для любого инженерного продукта метрики должны ориентироваться на его уникальные характеристики. Например, для электропоезда вряд ли полезна метрика «расход угля на километр пробега». С точки зрения метрик выделяют пять характеристик объектно-ориентированных систем: локализация, инкапсуляция, информационная закрытость, наследование и способы абстрагирования

объектов. Эти характеристики оказывают максимальное влияние на объектно-ориентированные метрики.

Локализация

Локализация фиксирует способ группировки информации в программе. В классических методах, где используется функциональная декомпозиция, информация локализуется вокруг функций. Функции в них реализуются как процедурные модули. В методах, управляемых данными, информация группируется вокруг структур данных. В объектно-ориентированной среде информация группируется внутри классов или объектов (инкапсуляцией как данных, так и процессов).

Поскольку в классических методах основной механизм локализации — функция, программные метрики ориентированы на внутреннюю структуру или сложность функций (длина модуля, связность, цикломатическая сложность) или на способ, которым функции связываются друг с другом (сцепление модулей).

Так как в объектно-ориентированной системе базовым элементом является класс, то локализация здесь основывается на объектах. Поэтому метрики должны применяться к классу (объекту) как к комплексной сущности. Кроме того, между операциями (функциями) и классами могут быть отношения не только «один-к-одному». Поэтому метрики, отображающие способы взаимодействия классов, должны быть приспособлены к отношениям «один-ко-многим», «многие-ко-многим».

Инкапсуляция

Вспомним, что инкапсуляция — это упаковка (связывание) совокупности элементов. Для классических ПС примерами низкоуровневой инкапсуляции являются записи и массивы. Механизмом инкапсуляции среднего уровня являются подпрограммы (процедуры, функции).

В объектно-ориентированных системах инкапсулируются обязанности класса, представляемые его атрибутами (а для агрегатов и атрибутами других классов), операциями и состояниями.

Для метрик учет инкапсуляции приводит к смещению фокуса измерений с одного модуля на группу атрибутов и обрабатывающих модулей (операций). Кроме того, инкапсуляция переводит измерения на более высокий уровень абстракции (пример — метрика «количество операций на класс»). Напротив, классические метрики ориентированы на низкий уровень — количество булевых условий (цикломатическая сложность) и количество строк программы.

Информационная закрытость

Информационная закрытость делает невидимыми операционные детали программного компонента. Другим компонентам доступна только необходимая информация.

Качественные объектно-ориентированные системы поддерживают высокий уровень информационной закрытости. Таким образом, метрики, измеряющие степень достигнутой закрытости, тем самым отображают качество объектно-ориентированного проекта.

Наследование

Наследование — это механизм, обеспечивающий тиражирование обязанностей одного класса в другие классы. Наследование распространяется через все уровни иерархии классов. Стандартные ПС не поддерживают эту характеристику.

Поскольку наследование — основная характеристика объектно-ориентированных систем, на ней фокусируются многие объектно-ориентированные метрики (количество детей — потомков класса, количество родителей, высота класса в иерархии наследования).

Абстракция

Абстракция — это механизм, который позволяет проектировщику выделять главное в программном компоненте (как атрибуте, так и операции), без учета второстепенных деталей. По мере перемещения на более высокие уровни абстракции мы игнорируем все большее количество деталей, обеспечивая все более общее представление понятия или элемента. По мере перемещения на более низкие уровни абстракции мы вводим все большее количество деталей, обеспечивая более точное представление понятия или элемента.

Класс — это абстракция, которая может быть представлена на различных уровнях детализации и различными способами (например, как список операций, последовательность состояний, последовательности взаимодействий). Поэтому объектно-ориентированные метрики должны представлять абстракции в терминах измерений класса. Примеры: количество экземпляров класса в приложении, количество родовых классов на приложение, отношение количества родовых к количеству неродовых классов.

Эволюция мер связи для объектно-ориентированных программных систем

В разделах «Связность модуля» и «Сцепление модулей» главы 6 было показано, что классической мерой сложности внутренних связей модуля является связность, а классической мерой сложности внешних связей — сцепление. Рассмотрим развитие этих мер применительно к объектно-ориентированным системам.

Связность объектов

В классическом методе Л. Констентайна и Э. Йордана определены семь типов связности:

1. *Связность по совпадению.* В модуле отсутствуют явно выраженные внутренние связи.
2. *Логическая связность.* Части модуля объединены по принципу функционального подобия.
3. *Временная связность.* Части модуля не связаны, но необходимы в один и тот же период работы системы.
4. *Процедурная связность.* Части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения.

5. *Коммуникативная связность.* Части модуля связаны по данным (работают с одной и той же структурой данных)
6. *Информационная (последовательная) связность.* Выходные данные одной части используются как входные данные в другой части модуля.
7. *Функциональная связность.* Части модуля вместе реализуют одну функцию.

Этот метод функционален по своей природе, поэтому наибольшей связностью здесь объявлена функциональная связность. Вместе с тем одним из принципиальных преимуществ объектно-ориентированного подхода является естественная связанность объектов.

Максимально связанным является объект, в котором представляется единая сущность и в который включены все операции над этой сущностью. Например, максимально связанным является объект, представляющий таблицу символов компилятора, если в него включены все функции, такие как «Добавить символ», «Поиск в таблице» и т. д.

Следовательно, восьмой тип связности можно определить так:

8. *Объектная связность.* Каждая операция обеспечивает функциональность, которая предусматривает, что все атрибуты объекта будут модифицироваться, отображаться и использоваться как базис для предоставления услуг.

Высокая связность — желательная характеристика, так как она означает, что объект представляет единую часть в предметной области, существует в едином пространстве. При изменении системы все действия над частью инкапсулируются в едином компоненте. Поэтому для производства изменения нет нужды модифицировать много компонентов.

Если функциональность в объектно-ориентированной системе обеспечивается наследованием от суперклассов, то связность объекта, который наследует атрибуты и операции, уменьшается. В этом случае нельзя рассматривать объект как отдельный модуль — должны учитываться все его суперклассы. Системные средства просмотра содействуют такому учету. Однако понимание элемента, который наследует атрибуты от нескольких суперклассов, резко усложняется.

Обсудим конкретные метрики для вычисления связности классов.

Метрики связности по данным

Л. Отт и Б. Мехра разработали модель секционирования класса [79]. Секционирование основывается на экземплярных переменных класса. Для каждого метода класса получают ряд секций, а затем производят объединение всех секций класса. Измерение связности основывается на количестве лексем данных (data tokens), которые появляются в нескольких секциях и «склеивают» секции в модуль. Под лексемами данных здесь понимают определения констант и переменных или ссылки на константы и переменные.

Базовым понятием методики является секция данных. Она составляется для каждого выходного параметра метода. *Секция данных* – это последовательность лексем данных в операторах, которые требуются для вычисления этого параметра.

Например, на рис. 12.1 представлен программный текст метода SumAndProduct. Все лексемы, входящие в секцию переменной SumN, выделены рамками. Сама секция для SumN записывается как следующая последовательность лексем:

$$N_1 \bullet SumN_1 \bullet I_1 \bullet SumN_2 \bullet 0_1 \bullet I_2 \bullet 1_2 \bullet N_2 \bullet SumN_3 \bullet SumN_4 \bullet I_3.$$

Заметим, что индекс в «1_2» указывает на второе вхождение лексемы «1» в текст метода. Аналогичным образом определяется секция для переменной ProdN:

$$N_1 \bullet ProdN_1 \bullet I_1 \bullet ProdN_2 \bullet 1_1 \bullet I_2 \bullet 1_2 \bullet N_2 \bullet ProdN_3 \bullet ProdN_4 \bullet I_4.$$

```
procedure SumAndProduct
(N: integer;
 var SumN, ProdN : integer );
 var
   I : integer;
begin
   SumN := 0;
   ProdN := 1;
   for I := 1 to N do begin
                          SumN := SumN + I;
                          ProdN := ProdN * I
                       end
end;
```

Рис. 12.1. Секция данных для переменной SumN

Для определения отношений между секциями данных можно показать профиль секций данных в методе. Для нашего примера профиль секций данных приведен в табл. 12.1.

Таблица 12.1. Профиль секций данных для метода SumAndProduct

SumN	ProdN	Оператор
		procedure SumAndProduct
1	1	(N:integer;
1	1	var SumN, ProdN:integer)
		var
1	1	I:integer;
		begin
2		SumN:=0
	2	ProdN:=1
3	3	for I:=1 to N do begin
3		SumN:=SumN+I
	3	ProdN:=ProdN*I
		end
		end;

Видно, что в столбце переменной для каждой секции указывается количество лексем из i-й строки метода, которые включаются в секцию.

Еще одно базовое понятие методики — секционированная абстракция. *Секционированная абстракция* — это объединение всех секций данных метода. Например, секционированная абстракция метода SumAndProduct имеет вид:

$$SA(SumAndProduct) =$$
$$\{N_1 \bullet SumN_1 \bullet I_1 \bullet SumN_2 \bullet 0_1 \bullet I_2 \bullet 1_2 \bullet N_2 \bullet SumN_3 \bullet SumN_4 \bullet I_3,$$
$$N_1 \bullet ProdN_1 \bullet I_1 \bullet ProdN_2 \bullet 1_1 \bullet I_2 \bullet 1_2 \bullet N_2 \bullet ProdN_3 \bullet ProdN_4 \bullet I_4\}.$$

Введем главные определения.

Секционированной абстракцией класса (*Class Slice Abstraction*) *CSA(C)* называют объединение секций всех экземплярных переменных класса. Полный набор секций составляют путем обработки всех методов класса.

Склеенными лексемами называют те лексемы данных, которые являются элементами более чем одной секции данных.

Сильно склеенными лексемами называют те склеенные лексемы, которые являются элементами всех секций данных.

Сильная связность по данным (*Strong Data Cohesion*) — это метрика, основанная на количестве лексем данных, входящих во все секции данных для класса. Иначе говоря, сильная связность по данным учитывает количество сильно склеенных лексем в классе *C*, она вычисляется по формуле:

$$SDC(C) = \frac{|SG(CSA(C))|}{|лексемы(C)|},$$

где $SG(CSA(C))$ — объединение сильно склеенных лексем каждого из методов класса *C*; $лексемы(C)$ — множество всех лексем данных класса *C*.

Таким образом, класс без сильно склеенных лексем имеет нулевую сильную связанность по данным.

Слабая связность по данным (*Weak Data Cohesion*) — метрика, которая оценивает связность, базируясь на склеенных лексемах. Склеенные лексемы не требуют связывания всех секций данных, поэтому данная метрика определяет более слабый тип связности. Слабая связность по данным вычисляется по формуле:

$$WDC(C) = \frac{|G(CSA(C))|}{|лексемы(C)|},$$

где $G(CSA(C))$ — объединение склеенных лексем каждого из методов класса.

Класс без склеенных лексем не имеет слабой связанности по данным.

Наиболее точной метрикой связности между секциями данных является *клейкость данных* (*Data Adhesiveness*). Клейкость данных определяется как отношение суммы из количеств секций, содержащих каждую склеенную лексему, к произведению количества лексем данных в классе на количество секций данных. Метрика вычисляется по формуле:

$$DA(C) = \frac{\sum d \in G(CSA(C))" d \in Секции}{|лексемы(C)| \times |CSA(C)|}.$$

Приведем пример. Применим метрики к классу, профиль секций которого показан в табл. 12.2.

Таблица 12.2. Профиль секций данных для класса Stack

array	top	size	Класс Stack
			class Stack {int *array, top, size;
			public:
			Stack (int s) {
2		2	size=s;
2		2	array=new int [size];
	2		top=0;}
			int IsEmpty () {
	2		return top==0};
			int Size () {
		2	return size};
			int Vtop () {
3	3		return array [top-1]; }
			void Push (int item) {
2	2	2	if (top==size)
			printf ("Empty stack.\n");
			else
3	3	3	array [top++]=item;}
			int Pop () {
	1		if (IsEmpty ())
			printf ("Full stack.\n");
			else
	1		--top;}
			};

Очевидно, что $CSA(Stack)$ включает три секции с 19 лексемами, имеет 5 сильно склеенных лексем и 12 склеенных лексем.

Расчеты по рассмотренным метрикам дают следующие значения:

$$SDC(CSA(Stack)) = 5/19 = 0.26,$$
$$WDC(CSA(Stack)) = 12/19 = 0.63,$$
$$DA(CSA(Stack)) = (7*2 + 5*3)/(19*3) = 0.51.$$

Метрики связности по методам

Д. Биемен и Б. Кенг предложили метрики связности класса, которые основаны на прямых и косвенных соединениях между парами методов [31]. Если существуют общие экземплярные переменные (одна или несколько), используемые в паре методов, то говорят, что эти методы соединены прямо. Пара методов может быть соединена косвенно, через другие прямо соединенные методы.

На рис. 12.2 представлены отношения между элементами класса Stack. Прямоугольниками обозначены методы класса, а овалами – экземплярные переменные. Связи показывают отношения использования между методами и переменными.

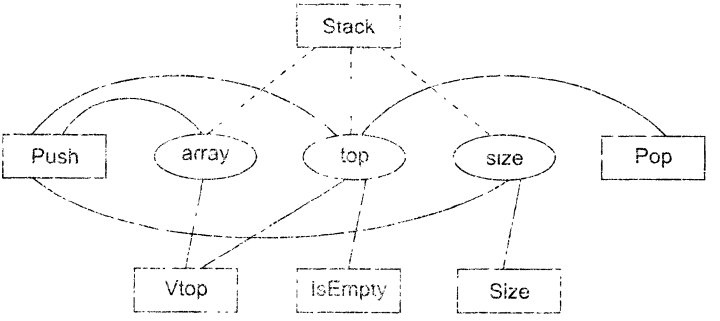


Рис. 12.2. Отношения между элементами класса Stack

Из рисунка видно, что экземплярная переменная top используется методами Stack, Push, Pop, Vtop и IsEmpty. Таким образом, все эти методы попарно прямо соединены. Напротив, методы Size и Pop соединены косвенно: Size соединен прямо с Push, который, в свою очередь, прямо соединен с Pop. Метод Stack является конструктором класса, то есть функцией инициализации. Обычно конструктору доступны все экземплярные переменные класса, он использует эти переменные совместно со всеми другими методами. Следовательно, конструкторы создают соединения и между такими методами, которые никак не связаны друг с другом. Поэтому ни конструкторы, ни деструкторы здесь не учитываются. Связи между конструктором и экземплярными переменными на рис. 12.2 показаны пунктирными линиями.

Для формализации модели вводятся понятия абстрактного метода и абстрактного класса.

Абстрактный метод $AM(M)$ – это представление реального метода M в виде множества экземплярных переменных, которые прямо или косвенно используются методом.

Экземплярная переменная прямо используется методом M, если она появляется в методе как лексема данных. Экземплярная переменная может быть определена в том же классе, что и M, или же в родительском классе этого класса. Множество экземплярных переменных, прямо используемых методом M, обозначим как $DU(M)$.

Экземплярная переменная косвенно используется методом M, если 1) экземплярная переменная прямо используется другим методом M', который вызывается (прямо или косвенно) из метода M, и 2) экземплярная переменная, прямо используемая методом M', находится в том же объекте, что и M.

Множество экземплярных переменных, косвенно используемых методом M, обозначим как $IU(M)$.

Количественно абстрактный метод формируется по выражению:

$$AM(M) = DU(M) \cup IU(M).$$

Абстрактный класс $AC(C)$ — это представление реального класса C в виде совокупности абстрактных методов, причем каждый абстрактный метод соответствует видимому методу класса C. Количественно абстрактный класс формируется по выражению:

$$AC(C)=[[AM(M)|M \in V(C)]],$$

где $V(C)$ — множество всех видимых методов в классе C и в классах - предках для C.

Отметим, что AM-представления различных методов могут совпадать, поэтому в AC могут быть дублированные элементы. В силу этого AC записывается в форме мультимножества (двойные квадратные скобки рассматриваются как его обозначение).

Локальный абстрактный класс $LAC(C)$ — это совокупность абстрактных методов, где каждый абстрактный метод соответствует видимому методу, определенному только внутри класса C. Количественно абстрактный класс формируется по выражению:

$$LAC(C)=[[AM(M)|M \in LV(C)]],$$

где $LV(C)$ — множество всех видимых методов, определенных в классе C.

Абстрактный класс для стека, приведенного в табл. 12.2, имеет вид:

$$AC(Stack)=[[\{top\},\{size\},\{array,top\},\{array,top,size\},\{pop\}]].$$

Поскольку класс Stack не имеет суперкласса, то справедливо:

$$AC(Stack)=LAC(Stack).$$

Пусть $NP(C)$ — общее количество пар абстрактных методов в $AC(C)$. NP определяет максимально возможное количество прямых или косвенных соединений в классе. Если в классе C имеются N методов, тогда $NP(C) = N^*(N-1)/2$. Обозначим:

- $NDC(C)$ — количество прямых соединений $AC(C)$;
- $NIC(C)$ — количество косвенных соединений в $AC(C)$.

Тогда метрики связности класса можно представить в следующем виде:

- *сильная связность класса* (*Tight Class Cohesion* (*TCC*)) определяется относительным количеством прямо соединенных методов:

$$TCC(C)=NDC(C)/NP(C);$$

- *слабая связность класса* (*Loose Class Cohesion* (*LCC*)) определяется относительным количеством прямо или косвенно соединенных методов:

$$LCC(C)=(NDC(C)+NIC(C))/NP(C).$$

Очевидно, что всегда справедливо следующее неравенство:

$$LCC(C) >= TCC(C).$$

Для класса Stack метрики связности имеют следующие значения:

$$TCC(Stack) = 7/10 = 0{,}7,$$
$$LCC(Stack) = 10/10 = 1.$$

Метрика *TCC* показывает, что 70% видимых методов класса Stack соединены прямо, а метрика *LCC* показывает, что все видимые методы класса Stack соединены прямо или косвенно.

Метрики *TCC* и *LCC* индицируют степень связанности между видимыми методами класса. Видимые методы либо определены в классе, либо унаследованы им. Конечно, очень полезны метрики связности для видимых методов, которые определены только внутри класса — ведь здесь исключается влияние связности суперкласса. Очевидно, что метрики локальной связности класса определяются на основе локального абстрактного класса. Отметим, что для локальной связности экземплярные переменные и вызываемые методы могут включать унаследованные переменные.

Сцепление объектов

В классическом методе Л. Констентайна и Э. Йордана определены шесть типов сцепления, которые ориентированы на процедурное проектирование [104].

Принципиальное преимущество объектно-ориентированного проектирования в том, что природа объектов приводит к созданию слабо сцепленных систем. Фундаментальное свойство объектно-ориентированного проектирования заключается в скрытости содержания объекта. Как правило, содержание объекта невидимо внешним элементам. Степень автономности объекта достаточно высока. Любой объект может быть замещен другим объектом с таким же интерфейсом.

Тем не менее наследование в объектно-ориентированных системах приводит к другой форме сцепления. Объекты, которые наследуют атрибуты и операции, сцеплены с их суперклассами. Изменения в суперклассах должны проводиться осторожно, так как эти изменения распространяются во все классы, которые наследуют их характеристики.

Таким образом, сами по себе объектно-ориентированные механизмы не гарантирует минимального сцепления. Конечно, классы — мощное средство абстракции данных. Их введение уменьшило поток данных между модулями и, следовательно, снизило общее сцепление внутри системы. Однако количество типов зависимостей между модулями выросло. Появились отношения наследования, делегирования, реализации и т. д. Более разнообразным стал состав модулей в системе (классы, объекты, свободные функции и процедуры, пакеты). Отсюда вывод: необходимость измерения и регулирования сцепления в объектно-ориентированных системах обострилась.

Рассмотрим объектно-ориентированные метрики сцепления, предложенные М. Хитцем и Б. Монтазери [60].

Зависимость изменения между классами

Зависимость изменения между классами CDBC (Change Dependency Between Classes) определяет потенциальный объем изменений, необходимых после модификации класса-сервера *SC* (server class) на этапе сопровождения. До тех пор пока реальное количество необходимых изменений класса-клиента *CC* (client class) неизвестно, *CDBC* указывает количество методов, на которые влияет изменение *SC*.

CDBC зависит от:

- области видимости изменяемого класса-сервера внутри класса-клиента (определяется типом отношения между *CS* и *CC*);
- вида доступа *CC* к *CS* (интерфейсный доступ или доступ реализации).

Возможные типы отношений приведены в табл. 12.3, где n — количество методов класса *CC*, α — количество методов *CC*, потенциально затрагиваемых изменением.

Таблица 12.3. Вклад отношений между клиентом и сервером в зависимость изменения

Тип отношения	α
SC не используется классом *CC*	0
SC — класс экземплярной переменной в классе *CC*	n
Локальные переменные типа *SC* используются внутри j методов класса *CC*	j
SC является суперклассом *CC*	n
SC является типом параметра для j методов класса *CC*	j
CC имеет доступ к глобальной переменной класса *SC*	n

Конечно, здесь предполагается, что те элементы класса-сервера *SC*, которые доступны классу-клиенту *CC*, являются предметом изменений. Авторы исходят из следующей точки зрения: если класс *SC* является «зрелой» абстракцией, то предполагается, что его интерфейс более стабилен, чем его реализация. Таким образом, многие изменения в реализации *SC* могут выполняться без влияния на его интерфейс. Поэтому вводится фактор стабильности интерфейса для класса-сервера, он обозначается как k ($0 \le k \le 1$). Вклад доступа к интерфейсу в зависимость изменения можно учесть умножением на $(1 - k)$.

Метрика для вычисления степени *CDBC* имеет вид:

$$A = \sum_{\substack{\text{доступ}\\ \text{к реализации}}} \alpha_i + (1-k) \times \sum_{\substack{\text{доступ}\\ \text{к интерфейсу}}} \alpha_i ;$$

$$CDBC\ (CC, SC) = min\ (\ n, A\ ).$$

Пути минимизации *CDBC*:

1. Ограничение доступа к интерфейсу класса-сервера.
2. Ограничение видимости классов-серверов (спецификаторами доступа public protected, private).

Локальность данных

Локальность данных LD (*Locality of Data*) — метрика, отражающая качество абстракции, реализуемой классом. Чем выше локальность данных, тем выше самодостаточность класса. Эта характеристика оказывает сильное влияние на такие внешние характеристики, как повторная используемость и тестируемость класса.

Метрика *LD* представляется как отношение количества локальных данных в классе к общему количеству данных, используемых этим классом.

Будем использовать терминологию языка C++. Обозначим как $M_i(1 \le i \le n)$ методы класса. В их число не будем включать методы чтения/записи экземплярных переменных. Тогда формулу для вычисления локальности данных можно записать в виде:

$$LD = \frac{\sum_{i=1}^{a} |L_i|}{\sum_{i=1}^{a} |T_i|},$$

где

- $L_i(1 \le i \le n)$ — множество локальных переменных, к которым имеют доступ методы M_i (прямо или с помощью методов чтения/записи). Такими переменными являются: непубличные экземплярные переменные класса; унаследованные защищенные экземплярные переменные их суперклассов; статические переменные, локально определенные в M_i;
- $T_i(1 \le i \le n)$ — множество всех переменных, используемых в M_i, кроме динамических локальных переменных, определенных в M_i.

Для обеспечения надежности оценки здесь исключены все вспомогательные переменные, определенные в M_i, — они не играют важной роли в проектировании.

Защищенная экземплярная переменная, которая унаследована классом *C*, является локальной переменной для его экземпляра (и, следовательно, является элементом L_i), даже если она не объявлена в классе *C*. Использование такой переменной методами класса не вредит локальности данных, однако нежелательно, если мы заинтересованы уменьшить значение *CDBC*.

Набор метрик Чидамбера и Кемерера

В 1994 году С. Чидамбер и К. Кемерер (Chidamber и Kemerer) предложили шесть проектных метрик, ориентированных на классы [44]. Класс — фундаментальный элемент объектно-ориентированной (ОО) системы. Поэтому измерения и метрики для отдельного класса, иерархии классов и сотрудничества классов бесценны для программного инженера, который должен оценить качество проекта.

Набор Чидамбера—Кемерера наиболее часто цитируется в программной индустрии и научных исследованиях. Рассмотрим каждую из метрик набора.

Метрика 1. Взвешенные методы на класс WMC (Weighted Methods Per Class)

Допустим, что в классе C определены n методов со сложностью $c_1, c_2, \ldots, c_n$. Для оценки сложности может быть выбрана любая метрика сложности (например, цикломатическая сложность). Главное — нормализовать эту метрику так, чтобы номинальная сложность для метода принимала значение 1. В этом случае

$$WMC = \sum_{i=1}^{n} c_i.$$

Количество методов и их сложность являются индикатором затрат на реализацию и тестирование классов. Кроме того, чем больше методов, тем сложнее дерево наследования (все подклассы наследуют методы их родителей). С ростом количества методов в классе его применение становится все более специфическим, тем самым ограничивается возможность многократного использования. По этим причинам метрика *WMC* должна иметь разумно низкое значение.

Очень часто применяют упрощенную версию метрики. При этом полагают $C_i = 1$, и тогда *WMC* — количество методов в классе.

Оказывается, что подсчитывать количество методов в классе достаточно сложно. Возможны два противоположных варианта учета.

1. Подсчитываются только методы текущего класса. Унаследованные методы игнорируются. Обоснование — унаследованные методы уже подсчитаны в тех классах, где они определялись. Таким образом, инкрементность класса — лучший показатель его функциональных возможностей, который отражает его право на существование. Наиболее важным источником информации для понимания того, что делает класс, являются его собственные операции. Если класс не может отреагировать на сообщение (например, в нем отсутствует собственный метод), тогда он пошлет сообщение родителю.
2. Подсчитываются методы, определенные в текущем классе, и все унаследованные методы. Этот подход подчеркивает важность пространства состояний в понимании класса (а не инкрементности класса).

Существует ряд промежуточных вариантов. Например, подсчитываются текущие методы и методы, прямо унаследованные от родителей. Аргумент в пользу данного подхода — на поведение дочернего класса наиболее сильно влияет специализация родительских классов.

На практике приемлем любой из описанных вариантов. Главное — не менять вариант учета от проекта к проекту. Только в этом случае обеспечивается корректный сбор метрических данных.

Метрика *WMC* дает относительную меру сложности класса. Если считать, что все методы имеют одинаковую сложность, то это будет просто количество методов в классе. Существуют рекомендации по сложности методов. Например, М. Лоренц считает, что средняя длина метода должна ограничиваться 8 строками для Smalltalk и 24 строками для C++ [69]. Вообще, класс, имеющий максимальное количество

методов среди классов одного с ним уровня, является наиболее сложным; скорее всего, он специфичен для данного приложения и содержит наибольшее количество ошибок.

Метрика 2. Высота дерева наследования DIT (Depth of Inheritance Tree)

DIT определяется как максимальная длина пути (в ребрах) от листа до корня дерева наследования классов. Для показанной на рис. 12.3 иерархии классов метрика *DIT* равна 3.

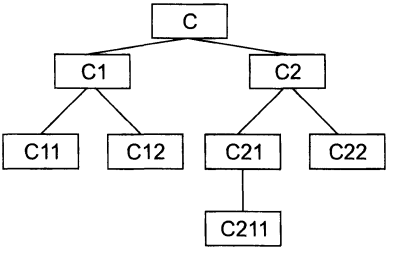


Рис. 12.3. Дерево наследования классов

Соответственно для отдельного класса *DIT* это длина максимального пути от данного класса до корневого класса в иерархии классов.

По мере роста *DIT*, вероятно, что классы нижнего уровня будут наследовать много методов. Это приводит к трудностям в предсказании поведения класса. Высокая иерархия классов (большое значение *DIT*) приводит к большей сложности результата проектирования, так как означает привлечение большего количества методов и классов.

Вместе с тем большое значение *DIT* подразумевает, что многие методы могут использоваться многократно.

Метрика 3. Количество детей NOC (Number of children)

Подклассы, которые непосредственно подчинены суперклассу, называются его детьми. Значение *NOC* равно количеству детей, то есть количеству непосредственных наследников класса в иерархии классов. На рис. 12.3 класс *C2* имеет двух детей — подклассы *C21* и *C22*.

С увеличением *NOC* возрастает многократность использования, так как наследование – это форма повторного использования.

Однако при возрастании *NOC* ослабляется абстракция родительского класса. Это означает, что в действительности некоторые из детей уже не являются членами родительского класса и могут быть неправильно использованы.

Кроме того, количество детей характеризует потенциальное влияние класса на проект. По мере роста *NOC* возрастает количество тестов, необходимых для проверки каждого ребенка.

Метрики *DIT* и *NOC* — количественные характеристики формы и размера структуры классов. Хорошо структурированная объектно-ориентированная система чаще бывает организована как лес классов, чем как сверхвысокое дерево. По

мнению Г. Буча, следует строить сбалансированные по высоте и ширине структуры наследования: обычно не выше, чем 7±2 уровня, и не шире, чем 7±2 ветви [38].

Метрика 4. Сцепление между классами объектов CBO (Coupling between object classes)

CBO — это количество сотрудничеств, предусмотренных для класса, то есть количество классов, с которыми он соединен. Соединение означает, что методы данного класса используют методы или экземплярные переменные другого класса.

Другое определение метрики имеет вид: *CBO* равно количеству сцеплений класса; сцепление образует вызов метода или атрибута в другом классе.

Данная метрика характеризует статическую составляющую внешних связей классов.

С ростом *CBO* многократность использования класса, вероятно, уменьшается. Очевидно, что чем больше независимость класса, тем легче его повторно использовать в другом приложении.

Высокое значение *CBO* усложняет модификацию и тестирование, которое следует за выполнением модификации. Понятно, что чем больше количество сцеплений, тем выше чувствительность всего проектного решения к изменениям в отдельных его частях. Минимизация межобъектных сцеплений улучшает модульность и содействует инкапсуляции результата проектирования.

CBO для каждого класса должно иметь разумно низкое значение. Это согласуется с рекомендациями по уменьшению сцепления стандартного программного обеспечения.

Метрика 5. Отклик для класса RFC (Response For a Class)

Введем вспомогательное определение. Множество отклика класса *RS* — это множество методов, которые могут выполняться в ответ на прибытие сообщений в объект этого класса. Формула для определения *RS* имеет вид:

$$RS = \{M\} \cup_{all\_i} \{R_i\},$$

где $\{R_i\}$ — множество методов, вызываемых методом i, $\{M\}$ — множество всех методов в классе.

Метрика *RFC* равна количеству методов во множестве отклика, то есть равна мощности этого множества:

$$RFC = \text{card}\{RS\}.$$

Приведем другое определение метрики: *RFC* — это количество методов класса плюс количество методов других классов, вызываемых из данного класса.

Метрика *RFC* является мерой потенциального взаимодействия данного класса с другими классами, позволяет судить о динамике поведения соответствующего объекта в системе. Данная метрика характеризует динамическую составляющую внешних связей классов.

Если в ответ на сообщение может быть вызвано большое количество методов, то усложняется тестирование и отладка класса, так как от разработчика тестов

требуется больший уровень понимания класса, растет длина тестовой последовательности.

С ростом *RFC* увеличивается сложность класса. Наихудшая величина отклика может использоваться при определении времени тестирования.

Метрика 6. Недостаток связности в методах LCOM (Lack of Cohesion in Methods)

Каждый метод внутри класса обращается к одному или нескольким атрибутам (экземплярным переменным). Метрика *LCOM* показывает, насколько методы не связаны друг с другом через атрибуты (переменные). Если все методы обращаются к одинаковым атрибутам, то $LCOM = 0$.

Введем обозначения:

- *НЕ_СВЯЗАНЫ* — количество пар методов без общих экземплярных переменных;
- *СВЯЗАНЫ* — количество пар методов с общими экземплярными переменными.
- I_j — набор экземплярных переменных, используемых методом M_j.

Очевидно, что

$$НЕ\_СВЯЗАНЫ = card\{I_{ij} \mid I_i \cap I_j = 0\},$$

$$СВЯЗАНЫ = card\{I_{ij} \mid I_i \cap I_j \neq 0\}.$$

Тогда формула для вычисления недостатка связности в методах примет вид:

$$LCOM = \begin{cases} НЕ\_СВЯЗАНЫ - СВЯЗАНЫ, & если\,(НЕ\_СВЯЗАНЫ > СВЯЗАНЫ); \\ 0, & в\,противном\,случае. \end{cases}$$

Можно определить метрику по-другому: *LCOM* — это количество пар методов, не связанных по атрибутам класса, минус количество пар методов, имеющих такую связь.

Недостаток связанности, то есть значение больше нуля, является свидетельством того, что класс следует разделить на два или более меньших подклассов.

Рассмотрим примеры применения метрики *LCOM*.

Пример 1. В классе имеются методы: *M1*, *M2*, *M3*, *M4*. Каждый метод работает со своим набором экземплярных переменных:

$$I_1 = \{a, b\};\ I_2 = \{a, c\};\ I_3 = \{x, y\};\ I_4 = \{m, n\}.$$

В этом случае:

$$НЕ\_СВЯЗАНЫ = \text{card}\,(I_{13}, I_{14}, I_{23}, I_{24}, I_{34}) = 5;\ СВЯЗАНЫ = \text{card}\,(I_{12}) = 1.$$
$$LCOM = 5 - 1 = 4.$$

Пример 2. В классе используются методы: *M1*, *M2*, *M3*. Для каждого метода задан свой набор экземплярных переменных:

$$I_1 = \{a, b\};\ I_2 = \{a, c\};\ I_3 = \{x, y\}.$$
$$НЕ\_СВЯЗАНЫ = \text{card}\,(I_{13}, I_{23}) = 2;\ СВЯЗАНЫ = \text{card}\,(I_{12}) = 1.$$
$$LCOM = 2 - 1 = 1.$$

Связность методов внутри класса должна быть высокой, так как это содействует инкапсуляции. Если *LCOM* имеет высокое значение, то методы слабо связаны друг с другом через атрибуты. Это увеличивает сложность, в связи с чем возрастает вероятность ошибок при проектировании.

Высокие значения *LCOM* означают, что класс, вероятно, надо спроектировать лучше (разбиением на два или более отдельных класса). Любое вычисление *LCOM* помогает определить недостатки в проектировании классов, так как эта метрика характеризует качество упаковки данных и методов в оболочку класса.

Вывод: связность в классе желательно сохранять высокой, то есть следует добиваться низкого значения *LCOM*.

Набор метрик Чидамбера–Кемерера – одна из пионерских работ по комплексной оценке качества ОО проектирования. Известны многочисленные предложения по усовершенствованию, развитию данного набора. Рассмотрим некоторые из них.

Недостатком метрики *WMC* является зависимость от реализации. Приведем пример. Рассмотрим класс, предлагающий операцию интегрирования. Возможны две реализации:

1) несколько простых операций

   Set_interval (min, max),
   Set_method (method),
   Set_precision (precision),
   Set_function_to_integrate (function),
   Integrate;

2) одна сложная операция

   Integrate (function, min, max, method, precision).

Для обеспечения независимости от этих реализаций можно предложить метрику *WMC2*:

$$WMC2 = \sum_{i=1}^{n} (Количество\ параметров\ i\text{-}го\ метода.$$

Для нашего примера *WMC2* = 5 и для первой и для второй реализации. Заметим, для первой реализации *WMC* = 5, а для второй реализации *WMC* = 1.

Дополнительно можно определить метрику *Среднее число аргументов метода ANAM* (Average Number of Arguments per Method):

$$ANAM = WMC2/WMC.$$

Полезность метрики *ANAM* объяснить еще легче. Она ориентирована на принятые в ОО проектировании решения – применять простые операции с малым количеством аргументов, а не сложные операции с многочисленными аргументами.

Самые большие нарекания вызвала метрика *LCOM* – исследователи выявили несколько аномалий ее поведения:

- *LCOM* дает нулевое значение для классов с самой разной связностью. Для преодоления этой проблемы была предложена метрика *LCOM**.

- *LCOM* учитывает связность методов только по данным, оставляя без внимания остальные механизмы их взаимодействия. Как следствие, классы с самой разной связностью имеют одинаковое значение *LCOM*.
- *LCOM* не рассматривает возможность доступа к данным класса через атрибуты с помощью функций get()/set(). Для таких классов может ошибочно формироваться высокое значение метрики.

Пример 3. На рис. 12.4 приведены примеры четырех классов, в которых методы изображены в виде овалов, а атрибуты, обрабатываемые методами, показаны точками внутри овалов. Кроме того, в прямоугольнике каждого класса приведена схема расчета *LCOM*.

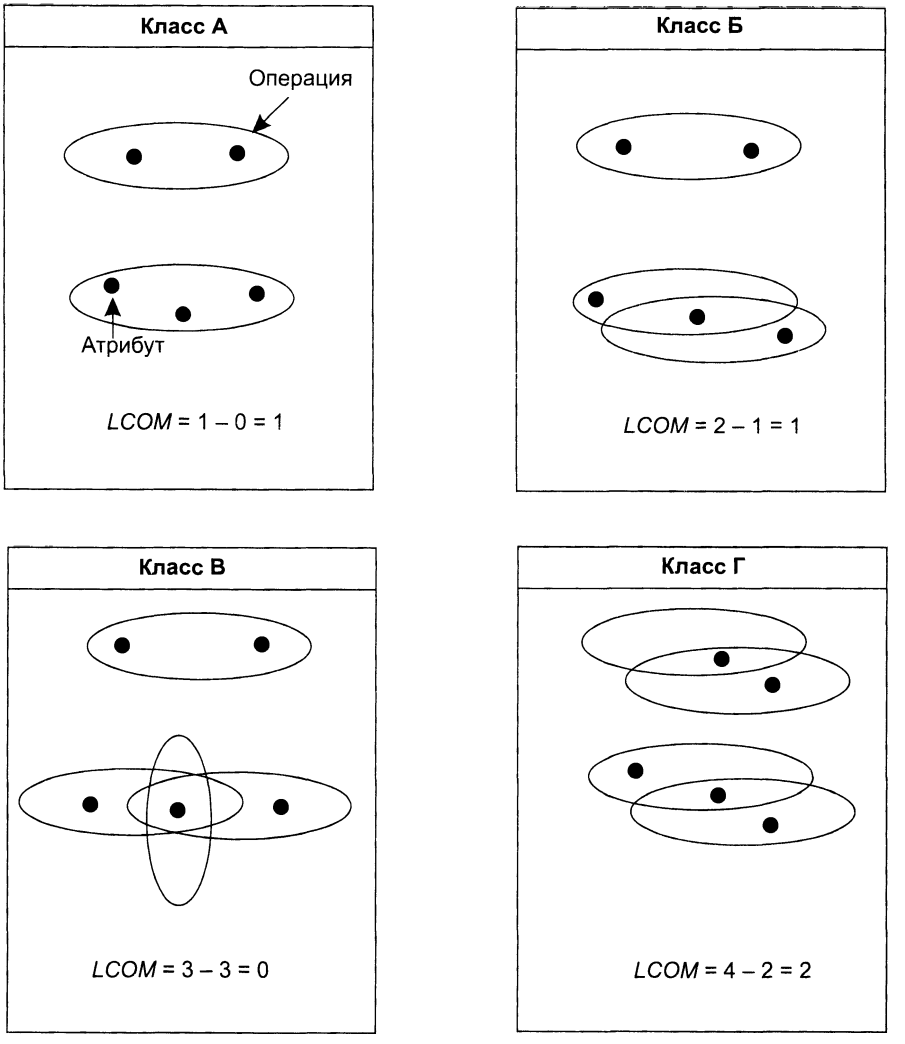


Рис. 12.4. Примеры классов с различной связностью

В соответствии с самой идеей связности по данным, все эти классы не связаны. То есть все классы нужно разделить на подклассы. В классах A и Б недостаток связности равен единице, в классе Г он еще больше (равен двум). Класс В считается структурно связанным (значение метрики равно нулю) и согласно метрике *LCOM* почему-то не должен быть кандидатом на разделение. Тем не менее трудно объяснить, почему добавление одного метода в класс A, приводящее к классу категории Б, не должно менять связность, а выполнение того же действия применительно к классу Б может давать противоречивый эффект: уменьшать *LCOM* в случае класса В или увеличивать *LCOM* в случае класса Г.

Б. Хендерсон–Селлерс, Л. Константайн и И. Грэхем предложили дополнительную метрику *LCOM** [55]. Нулевое значение метрики свидетельствует о высокой, хорошей связности, а высокое — о плохой связности.

Формула для *LCOM** имеет вид:

$$LCOM^* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a} m(A_j)\right) - m}{1 - m},$$

где m — количество методов класса, a — количество атрибутов класса, $a(M_i)$ — количество атрибутов, к которым имеет доступ метод M_i, а $m(A_j)$ — количество методов, которые имеют доступ к атрибуту A_j.

Если каждый метод имеет доступ ко всем атрибутам, значение метрики равно нулю (максимально хорошая связность). Если каждый метод работает только с одним, «своим» атрибутом, метрика дает единичное значение (максимально плохая связность). При наличии атрибутов, которые недоступны методам класса, значение метрики изменяется в диапазоне от 1 до 2. Такие значения свидетельствуют об ошибке проектирования (например, в классе есть «мертвые» атрибуты, или атрибуты, доступные только из внешней среды). Наконец, при нулевом количестве атрибутов или единственном методе метрика *LCOM** не определена.

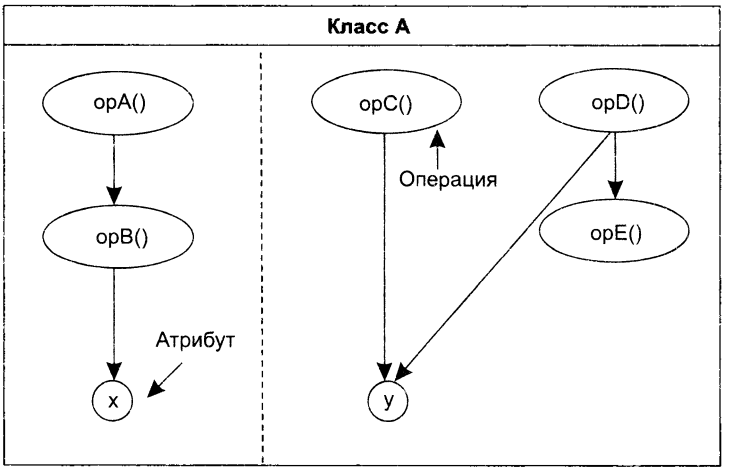
М. Хитц и Б. Монтазери определяют недостаток связности *LCOM4* как количество связанных компонентов в классе [59]. Связанным компонентом они называют набор таких методов (например, *a* и *b*) и атрибутов, для которых справедлив любой из трех случаев:

- оба метода обращаются к одному и тому же атрибуту;
- метод *a* вызывает метод *b*;
- метод *b* вызывает метод *a*.

В связанном классе должен быть только один связанный компонент. Наличие двух и более связанных компонентов говорит о необходимости разделения класса на части. Для определения значения *LCOM4* надо нарисовать граф связей класса. Вершинами этого графа являются методы и атрибуты, а ребрами — управляющие и информационные связи между вершинами. Управляющие связи показывают передачи управления, а информационные связи — передачу данных.

Например, на рис. 12.5 показан граф связей класса A, где овалы соответствуют его методам (opA(), opB(), opC(), opD(), opE()), а кружки — атрибутам (x и y). Между ними отображаются стрелки связей. Поскольку в графе два связанных

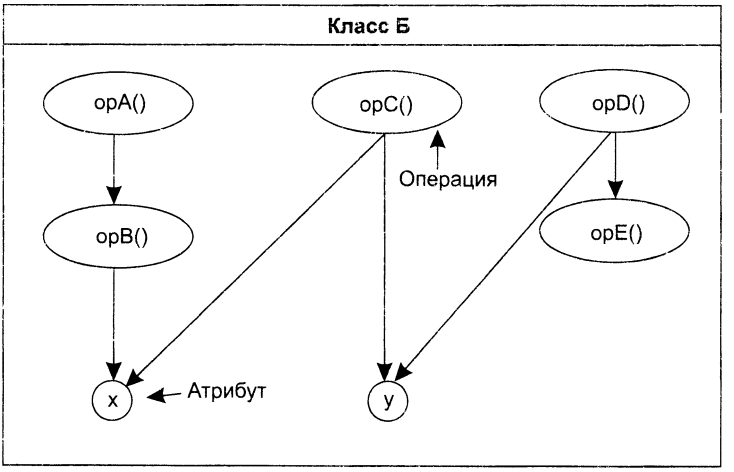
компонента (границы между ними отмечены пунктирной линией), *LCOM4* = 2. Это значит, что класс нужно разделить на два подкласса {opA(), opB(), x} и {opC(), opD(), opE(), y}.



LCOM4 = 2

Рис. 12.5. Класс с плохой связностью

Другой пример иллюстрируется классом на рис. 12.6.



LCOM4 = 1

Рис. 12.6. Класс с хорошей связностью

Здесь констатируется хорошая связность *LCOM4* = 1, поскольку имеется только один связанный компонент.

Еще один вариант усовершенствования вводит метрику, симметричную метрике *LCOM*. В то время как формула метрики *LCOM* имеет вид

$$LCOM = \max(0, НЕ\_СВЯЗАНЫ - СВЯЗАНЫ),$$

симметричная ей метрика *Нормализованная NLCOM* вычисляется по формуле

$$NLCOM = СВЯЗАНЫ/(НЕ\_СВЯЗАНЫ+СВЯЗАНЫ).$$

Диапазон значений этой метрики: $0 \leq NLCOM \leq 1$, причем, чем ближе *NLCOM* к 1, тем выше связанность класса.

В наборе Чидамбера—Кемерера отсутствует метрика для прямого измерения информационной закрытости класса. В силу этого была предложена метрика *Поведенческая закрытость информации BIH* (Behaviourial Information Hiding):

$$BIH = (WEOC/WIEOC),$$

где *WEOC — Взвешенные внешние операции на класс* (фактически это *WMC*); *WIEOC — Взвешенные внутренние и внешние операции на класс.*

WIEOC вычисляется так же, как и *WMC*, но учитывает полный набор операций, реализуемых классом. Если *BIH* = 1, класс показывает другим классам все свои возможности. Чем меньше *BIH*, тем меньше видимо поведение класса. *BIH* может рассматриваться и как мера сложности. Сложные классы, вероятно, будут иметь малые значения *BIH*, а простые классы — значения, близкие к 1. Если класс с высокой *WMC* имеет значение *BIH*, близкое к 1, следует выяснить — почему он настолько видим извне.

Использование метрик Чидамбера—Кемерера

Поскольку основу логического представления ПО образует структура классов, для оценки ее качества удобно использовать метрики Чидамбера—Кемерера. Пример расчета метрик для структуры, показанной на рис. 12.7, представлен в табл. 12.4.

Прокомментируем результаты расчета. Класс `Class A` имеет три метода (`op_a1()`, `op_a2()`, `op_a3()`), трех детей (`Class B`, `Class C`, `Class D`) и является корневым классом. Поэтому метрики *WMC*, *NOC* и *DIT* имеют соответственно значения 3, 3 и 0.

Метрика *CBO* для класса `Class A` равна 1, так как он использует один метод из другого класса (метод `op_e()` из класса `Class E`, он вызывается из метода `op_a3()`). Метрика *RFC* для класса `Class A` равна 4, так как в ответ на прибытие в этот класс сообщений возможно выполнение 4-х методов (три объявлены в этом классе, а четвертый метод `op_e()` вызывается из `op_a3()`).

Таблица 12.4. Пример расчета метрик Чидамбера—Кемерера

Имя класса	WMC	DIT	NOC	CBO	RFC	LCOM
Class A	3	0	3	1	4	1
Class B	1	1	0	0	1	0
Class C	1	1	0	0	1	0
Class D	2	1	0	2	3	0

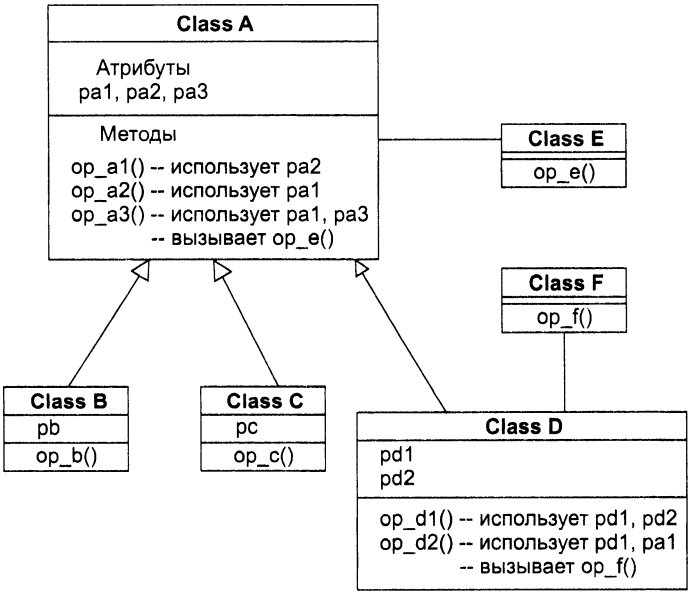


Рис. 12.7. Структура классов для расчета метрик Чидамбера—Кемерера

Для вычисления метрики *LCOM* надо определить количество пар методов класса. Оно рассчитывается по формуле:

$$C_m^2 = m!/(2(m-2)!),$$

где m — количество методов класса.

Поскольку в классе три метода, возможны три пары: op_a1()&op_a2(), op_a1()&op_a3() и op_a2()&op_a3(). Первая и вторая пары не имеют общих атрибутов, третья пара имеет общий атрибут (pa1). Таким образом, количество несвязанных пар равно 2, количество связанных пар равно 1 и $LCOM = 2 - 1 = 1$.

Отметим также, что для класса Class D метрика *CBO* равна 2, так как здесь используются атрибут pa1 и метод op_f() из других классов. Метрика *LCOM* в этом классе равна 0, поскольку методы op_d1() и op_d2() связаны по атрибуту pd1, а отрицательное значение запрещено.

Метрики Лоренца и Кидда

Коллекция метрик Лоренца и Кидда — результат практического, промышленного подхода к оценке ОО-проектов [69].

Метрики, ориентированные на классы

М. Лоренц и Д. Кидд подразделяют метрики, ориентированные на классы, на четыре категории: метрики размера, метрики наследования, внутренние и внешние метрики.

Размерно-ориентированные метрики основаны на подсчете атрибутов и операций для отдельных классов, а также их средних значений для всей ОО-системы. Метрики наследования акцентируют внимание на способе повторного использования операций в иерархии классов. Внутренние метрики классов рассматривают вопросы связности и кодирования. Внешние метрики исследуют сцепление и повторное использование.

Метрика 1. Размер класса CS (Class Size)

Общий размер класса определяется с помощью следующих измерений:

- общее количество операций (вместе с приватными и наследуемыми экземплярными операциями), которые инкапсулируются внутри класса;
- количество атрибутов (вместе с приватными и наследуемыми экземплярными атрибутами), которые инкапсулируются классом.

Метрика *WMC* Чидамбера и Кемерера также является взвешенной метрикой размера класса.

Большие значения *CS* указывают, что класс имеет слишком много обязанностей. Они уменьшают возможность повторного использования класса, усложняют его реализацию и тестирование.

При определении размера класса унаследованным (публичным) операциям и атрибутам придают больший удельный вес. Причина – приватные операции и атрибуты обеспечивают специализацию и более локализованы в проекте.

Могут вычисляться средние количества атрибутов и операций класса. Чем меньше среднее значение размера, тем больше вероятность повторного использования класса.

Рекомендуемое значение $CS \leq 20$ методов.

Метрика 2. Количество операций, переопределяемых подклассом NOO (Number of Operations Overridden by a Subclass)

Переопределением называют случай, когда подкласс замещает операцию, унаследованную от суперкласса, своей собственной версией.

Большие значения *NOO* обычно указывают на проблемы проектирования. Ясно, что подкласс должен расширять операции суперкласса. Расширение проявляется в виде новых имен операций. Если же велико *NOO*, то разработчик нарушает абстракцию суперкласса. Это ослабляет иерархию классов, усложняет тестирование и модификацию программного обеспечения.

Рекомендуемое значение $NOO \leq 3$ методов.

Метрика 3. Количество операций, добавленных подклассом NOA (Number of Operations Added by a Subclass)

Подклассы специализируются добавлением приватных операций и атрибутов. С ростом *NOA* подкласс удаляется от абстракции суперкласса. Обычно при увеличении высоты иерархии классов (увеличении *DIT*) должно уменьшаться значение *NOA* на нижних уровнях иерархии.

Для рекомендуемых значений $CS = 20$ и $DIT = 6$ рекомендуемое значение $NOA \leq 4$ методов (для класса-листа).

Метрика 4. Индекс специализации SI (Specialization Index)

Обеспечивает грубую оценку степени специализации каждого подкласса. Специализация достигается добавлением, удалением или переопределением операций.

$$SI = (NOO \times уровень)/M_{общ},$$

где *уровень* — номер уровня в иерархии, на котором находится подкласс, $M_{общ}$ — общее количество методов класса.

Пример расчета индексов специализации приведен на рис. 12.8.

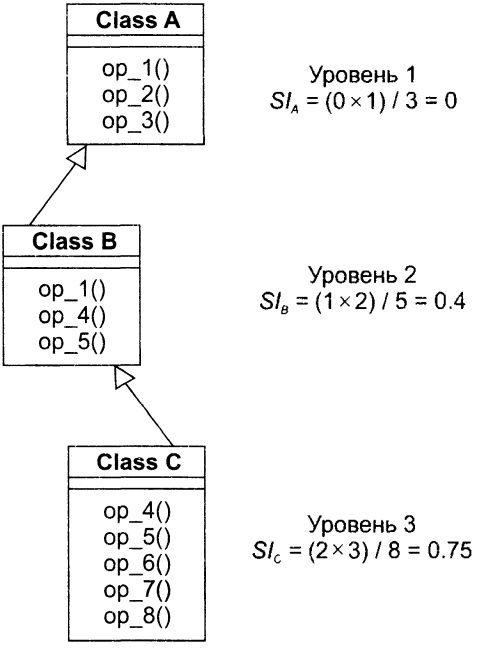


Рис. 12.8. Расчет индексов специализации классов

Чем выше значение *SI*, тем больше вероятность того, что в иерархии классов есть классы, нарушающие абстракцию суперкласса.

Рекомендуемое значение $SI \leq 0{,}15$.

Операционно-ориентированные метрики

Эта группа метрик ориентирована на оценку операций в классах. Обычно методы имеют тенденцию быть небольшими как по размеру, так и по логической сложности. Тем не менее реальные характеристики операций могут быть полезны для глубокого понимания системы.

Метрика 5. Средний размер операции OS_{avg} (Average Operation Size)

В качестве индикатора размера может использоваться количество строк программы, однако LOC-оценки приводят к известным проблемам. Альтернативный вариант — *«количество сообщений, посланных операцией»*.

Рост значения метрики означает, что обязанности размещены в классе не очень удачно.

Рекомендуемое значение $OS_{avg} \leq 9$.

Метрика 6. Сложность операции OC (Operation Complexity)

Сложность операции может вычисляться с помощью стандартных метрик сложности, то есть с помощью LOC- или FP-оценок, метрики цикломатической сложности, метрики Холстеда.

М. Лоренц и Д. Кидд предлагают вычислять *OC* суммированием оценок с весовыми коэффициентами, приведенными в табл. 12.5.

Таблица 12.5. Весовые коэффициенты для метрики OC

Параметр	Вес
Вызовы функций API	5.0
Присваивания	0,5
Арифметические операции	2,0
Сообщения с параметрами	3,0
Вложенные выражения	0,5
Параметры	0,3
Простые вызовы	7,0
Временные переменные	0,5
Сообщения без параметров	1,0

Поскольку операция должна быть ограничена конкретной обязанностью, желательно уменьшать *OC*.

Рекомендуемое значение $OC \leq 65$ (для предложенного суммирования).

Метрика 7. Среднее количество параметров на операцию NP_{avg} (Average Number of Parameters per operation)

Чем больше параметров у операции, тем сложнее сотрудничество между объектами. Поэтому значение NP_{avg} должно быть как можно меньшим.

Рекомендуемое значение $NP_{avg} = 0{,}7$.

Метрики для ОО-проектов

Основными задачами менеджера проекта являются планирование, координация, отслеживание работ и управление программным проектом.

Одним из ключевых вопросов планирования является оценка размера программного продукта. Прогноз размера продукта обеспечивают следующие ОО-метрики.

Метрика 8. Количество описаний сценариев NSS (Number of Scenario Scripts)

Это количество прямо пропорционально количеству классов, требуемых для реализации требований, количеству состояний для каждого класса, а также количеству методов, атрибутов и сотрудничеств. Метрика *NSS* — эффективный индикатор размера программы.

Рекомендуемое значение *NSS* — не менее одного сценария на публичный протокол подсистемы, отражающий основные функциональные требования к подсистеме.

Метрика 9. Количество ключевых классов NKC (Number of Key Classes)

Ключевой класс прямо связан с коммерческой проблемной областью, для которой предназначена система. Маловероятно, что ключевой класс может появиться в результате повторного использования существующего класса. Поэтому значение *NKC* достоверно отражает предстоящий объем разработки. М. Лоренц и Д. Кидд предполагают, что в типовой ОО-системе на долю ключевых классов приходится 20–40% от общего количества классов. Как правило, оставшиеся классы реализуют общую инфраструктуру (GUI, коммуникации, базы данных).

Рекомендуемое значение: если $NKC < 0{,}2$ от общего количества классов системы, следует углубить исследование проблемной области (для обнаружения важнейших абстракций, которые нужно реализовать).

Метрика 10. Количество подсистем NSUB (Number of SUBsystem)

Количество подсистем обеспечивает понимание следующих вопросов: размещение ресурсов, планирование (с акцентом на параллельную разработку), общие затраты на интеграцию.

Рекомендуемое значение: $NSUB > 3$.

Значения метрик *NSS*, *NKC*, *NSUB* полезно накапливать как результат каждого выполненного ОО-проекта. Так формируется метрический базис фирмы, в который также включаются метрические значения по классами и операциям. Эти исторические данные могут использоваться для вычисления метрик производительности (среднее количество классов на разработчика или среднее количество методов на человеко-месяц). Совместное применение метрик позволяет оценивать затраты, продолжительность, персонал и другие характеристики текущего проекта.

Набор метрик Фернандо Абреу

Набор метрик *MOOD* (Metrics for Object Oriented Design), предложенный Ф. Абреу в 1994 году, — другой пример академического подхода к оценке качества ОО-проектирования [19]. Основными целями *MOOD*-набора являются:

1) покрытие базовых механизмов объектно-ориентированной парадигмы, таких как инкапсуляция, наследование, полиморфизм, посылка сообщений;

2) формальное определение метрик, позволяющее избежать субъективности измерения;
3) независимость от размера оцениваемого программного продукта;
4) независимость от языка программирования, на котором написан оцениваемый продукт.

Набор *MOOD* включает в себя следующие метрики:

1) фактор закрытости метода (*MHF*);
2) фактор закрытости атрибута (*AHF*);
3) фактор наследования метода (*MIF*);
4) фактор наследования атрибута (*AIF*);
5) фактор полиморфизма (*POF*);
6) фактор сцепления (*COF*).

Каждая из этих метрик относится к основному механизму объектно-ориентированной парадигмы: инкапсуляции (*MHF* и *AHF*), наследованию (*MIF* и *AIF*), полиморфизму (*POF*) и посылке сообщений (*COF*). В определениях *MOOD* не используются специфические конструкции языков программирования.

Метрика 1. Фактор закрытости метода MHF (Method Hiding Factor)

Введем обозначения:

- $M_v(C_i)$ — количество видимых методов в классе C_i (интерфейс класса);
- $M_h(C_i)$ — количество скрытых методов в классе C_i (реализация класса);
- $M_d(C_i) = M_v(C_i) + M_h(C_i)$ — общее количество методов, определенных в классе C_i (унаследованные методы не учитываются).

Тогда формула метрики *MHF* примет вид:

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)},$$

где TC — количество классов в системе.

Если видимость m-го метода i-го класса из j-го класса вычислять по выражению

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1, & if \begin{cases} i \neq j \\ C_j \ \ может \ \ вызвать \ \ M_{mi}, \end{cases} \\ 0, & else \end{cases}$$

а процентное количество классов, которые видят m-й метод i-го класса, определять по соотношению

$$V(M_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(M_{mi}, C_j)}{TC - 1},$$

то формулу метрики *MHF* можно представить в виде:

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)}.$$

В числителе этой формулы *MHF* — сумма закрытости всех методов во всех классах. Закрытость метода — процентное количество классов, из которых данный метод невидим. Знаменатель *MHF* — общее количество методов, определенных в рассматриваемой системе.

С увеличением *MHF* уменьшаются плотность дефектов в системе и затраты на их устранение. Обычно разработка класса представляет собой пошаговый процесс, при котором к классу добавляется все больше и больше деталей (скрытых методов). Такая схема разработки способствует возрастанию как значения *MHF*, так и качества класса.

Метрика 2. Фактор закрытости атрибута AHF (Attribute Hiding Factor)

Введем обозначения:

- $A_v(C_i)$ — количество видимых атрибутов в классе C_i (интерфейс класса);
- $A_h(C_i)$ — количество скрытых атрибутов в классе C_i (реализация класса);
- $A_d(C_i) = A_v(C_i) + A_h(C_i)$ — общее количество атрибутов, определенных в классе C_i (унаследованные атрибуты не учитываются).

Тогда формула метрики *AHF* примет вид:

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)},$$

где *TC* — количество классов в системе.

Если видимость *m*-го атрибута *i*-го класса из *j*-го класса вычислять по выражению

$$is\_visible(A_m, C_j) = \begin{cases} 1, & if \begin{cases} j \neq i \\ C_j \ \ может \ \ обращаться \ \ к \ \ A_m, \end{cases} \\ 0, & else \end{cases}$$

а процентное количество классов, которые видят m-й атрибут i-го класса, определять по соотношению

$$V(A_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(A_{mi}, C_j)}{TC - 1},$$

то формулу метрики *AHF* можно представить в виде:

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)}.$$

В числителе этой формулы *AHF* — сумма закрытости всех атрибутов во всех классах. Закрытость атрибута — процентное количество классов, из которых данный атрибут невидим. Знаменатель *AHF* — общее количество атрибутов, определенных в рассматриваемой системе.

В идеальном случае все атрибуты должны быть скрыты и доступны только для методов соответствующего класса (*AHF* = 100%).

Метрика 3. Фактор наследования метода MIF (Method Inheritance Factor)

Введем обозначения:

- $M_i(C_i)$ — количество унаследованных и не переопределенных методов в классе C_i;
- $M_o(C_i)$ — количество унаследованных и переопределенных методов в классе C_i;
- $M_n(C_i)$ — количество новых (не унаследованных и переопределенных) методов в классе C_i;
- $M_d(C_i) = M_n(C_i) + M_o(C_i)$ — количество методов, определенных в классе C_i;
- $M_a(C_i) = M_d(C_i) + M_i(C_i)$ — общее количество методов, доступных в классе C

Тогда формула метрики *MIF* примет вид:

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}.$$

Числителем *MIF* является сумма унаследованных (и не переопределенных) методов во всех классах рассматриваемой системы. Знаменатель *MIF* — это общее количество доступных методов (локально определенных и унаследованных) для всех классов.

Значение *MIF* = 0 указывает, что в системе отсутствует эффективное наследование, например все унаследованные методы переопределены.

С увеличением *MIF* уменьшается плотность дефектов и затраты на исправление ошибок. Очень большие значения *MIF* (70–80%) приводят к обратному эффекту, но этот факт нуждается в дополнительной экспериментальной проверке.

Сформулируем «осторожный» вывод: умеренное использование наследования — подходящее средство для снижения плотности дефектов и затрат на доработку.

Метрика 4. Фактор наследования атрибута AIF (Attribute Inheritance Factor)

Введем обозначения:

- $A_i(C_i)$ — количество унаследованных и не переопределенных атрибутов в классе C_i;
- $A_o(C_i)$ — количество унаследованных и переопределенных атрибутов в классе C_i;
- $A_n(C_i)$ — количество новых (не унаследованных и переопределенных) атрибутов в классе C_i;
- $A_d(C_i) = A_n(C_i) + A_o(C_i)$ — количество атрибутов, определенных в классе C_i;
- $A_a(C_i) = A_d(C_i) + A_i(C_i)$ — общее количество атрибутов, доступных в классе C_i.

Тогда формула метрики *AIF* примет вид:

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}.$$

Числителем *AIF* является сумма унаследованных (и не переопределенных) атрибутов во всех классах рассматриваемой системы. Знаменатель *AIF* — это общее количество доступных атрибутов (локально определенных и унаследованных) для всех классов.

Метрика 5. Фактор полиморфизма POF (Polymorphism Factor)

Введем обозначения:

- $M_o(C_i)$ — количество унаследованных и переопределенных методов в классе C_i;
- $M_n(C_i)$ — количество новых (не унаследованных и переопределенных) методов в классе C_i;
- $DC(C_i)$ — количество потомков класса C_i;
- $M_d(C_i) = M_n(C_i) + M_o(C_i)$ — количество методов, определенных в классе C_i.

Тогда формула метрики *POF* примет вид:

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}.$$

Числитель *POF* фиксирует реальное количество возможных полиморфных ситуаций. Очевидно, что сообщение, посланное в класс C_i, связывается (статически или динамически) с реализацией именуемого метода. Этот метод, в свою очередь, может или представляться несколькими «формами» или переопределяться (в потомках C_i).

Знаменатель *POF* представляет максимальное количество возможных полиморфных ситуаций для класса C_i. Имеется в виду случай, когда все новые методы, определенные в C_i, переопределяются во всех его потомках.

Умеренное использование полиморфизма уменьшает как плотность дефектов, так и затраты на доработку. Однако при *POF* > 10% возможен обратный эффект.

Метрика 6. Фактор сцепления COF (Coupling Factor)

В данном наборе сцепление фиксирует наличие между классами отношения клиент-поставщик (client-supplier). Отношение клиент-поставщик ($C_c \Rightarrow C_s$) здесь означает, что класс-клиент содержит по меньшей мере одну не унаследованную ссылку на атрибут или метод класса-поставщика.

Если наличие отношения клиент-поставщик определять по выражению

$$is\_client(C_c, C_s) = \begin{cases} 1 & if \quad C_c \neq C_s \cap C_s \neq C_t \\ 0 & else \end{cases},$$

то формула для вычисления метрики *COF* примет вид:

$$COF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} is\_client(C_i, C_j) \right]}{TC^2 - TC}.$$

Знаменатель *COF* соответствует максимально возможному количеству сцеплений в системе с *TC*-классами (потенциально каждый класс может быть поставщиком для других классов). Из рассмотрения исключены рефлексивные отношения – когда класс является собственным поставщиком. Числитель *COF* фиксирует реальное количество сцеплений, не относящихся к наследованию.

С увеличением сцепления классов плотности дефектов и затрат на доработку также возрастают. Сцепления отрицательно влияют на качество ПО, их нужно сводить к минимуму. Практическое применение этой метрики доказывает, что сцепление увеличивает сложность, уменьшает инкапсуляцию и возможности повторного использования, затрудняет понимание и сопровождаемость ПО.

Аспектно-ориентированные метрики

Как известно, при создании метрик для оценки качества программной системы необходимо рассматривать меру сложности модулей, меру сложности внешних связей (между модулями) и меру сложности внутренних связей (внутри модулей). Традиционно с внешними связями сопоставляют характеристику «сцепление», а с внутренними связями – характеристику «связность». Сцепление представляет собой меру взаимодействия модулей по данным, которую желательно уменьшать

Связность модуля, в свою очередь, определяется мерой зависимости его частей. Чем выше связность модуля, тем лучше результат проектирования.

В настоящее время задача создания метрик проектирования для объектно-ориентированных программных систем практически решена. Эти метрики группируются относительно понятия «класс» — основного строительного элемента таких систем.

Аспектно-ориентированный подход (АОП) является надстройкой над объектно-ориентированным подходом. Эта надстройка расширяет ООП, а не заменяет его. АОП вводит в системную структуру новые строительные элементы (аспекты), которые взаимодействуют с базовыми элементами — объектами и классами. За счет этого повышается мощность и эффективность инструментария разработки.

В качестве основы для оценки аспектных структур целесообразно использовать характеристику «сцепление», так как аспект, по определению, должен быть сильно сцеплен (должен интенсивно взаимодействовать) с базовыми элементами системы. Чем выше сцепление аспекта, тем ниже сцепление объектов системы и, следовательно, тем выше качество программной системы.

Удельный вес аспекта SWA (Specific Weight of Aspect)

Метрика *SWA* (Specific Weight of Aspect) оценивает степень воздействия конкретного аспекта на базовые элементы системы (экземпляры классов) при реализации определенного пересекающего требования. Чем больше функциональность аспекта, тем больше его сцепление и тем ниже сцепление экземпляров классов.

Понятно, что реализация пересекающего требования включает в себя функциональность (атрибуты и операции) разных классов. Чем больше такой функциональности заключено в аспект, тем выше его удельный вес.

Будем считать, что некоторый аспект воздействует на n классов. Обозначим количество операций и атрибутов, «изымаемых» аспектом из i-го класса, как $f_a(i)$, а общее количество операций и атрибутов i-го класса как $f_{all}(i)$. В этом случае метрика *SWA* рассчитывается следующим образом:

$$SWA = \frac{\sum_{i=1}^{n} f_a(i)}{\sum_{i=1}^{n} f_{all}(i)}.$$

Очевидно, что метрика *SWA* определяет степень полезности аспекта, она показывает, насколько данный аспект «освобождает» классы системы от некоторого пересекающего требования. Значение метрики может изменяться в пределах от 0 до 1. Чем больше удельный вес аспекта, тем выше качество проектного решения. Если значение близко к нулю, то введение аспекта не имеет смысла. Значения *SWA*, близкие к единице, говорят о максимальном влиянии аспекта на достижение конечной цели АОП — инкапсуляции всех пересекающих требований в отдельные элементы.

Эффективность аспекта EFA (EFficiency of Aspect)

Метрика *EFA* (EFficiency of Aspect) ориентирована на измерение качества аспекта с точки зрения системы. Основное назначение аспекта — уменьшить сцепление

в программной системе. Ведь аспект избавляет классы от необходимости взаимодействия при реализации пересекающего требования. Данная метрика должна показывать, насколько эффективно решается эта задача. Очевидно, что для вычисления метрики нужно измерить сцепление классов до и после введения аспекта.

По-прежнему будем считать, что некоторый аспект воздействует на n классов. Иными словами, эти классы «работают» на пересекающее требование, реализуемое аспектом. Назовем эти классы задействованными. Обозначим величину сцепления i-го задействованного класса до введения аспекта как $C_{ini}(i)$, а после введения аспекта — как $C_{fin}(i)$. Тогда метрика *EFA* рассчитывается по формуле:

$$EFA = \frac{\sum_{i=1}^{n} C_{im}(i) - \sum_{i=1}^{n} C_{fm}(i)}{\sum_{i=1}^{n} C_{im}(i)}.$$

Для вычисления сцепления можно использовать стандартные объектно-ориентированные метрики, например, из набора Чидамбера—Кемерера.

Видим, что метрика *EFA* может принимать значения в диапазоне от 0 до 1. Чем больше значение, тем выше эффективность аспекта. Если значение близко к нулю, то введение аспекта не дало желаемого результата. Значения *EFA*, близкие к единице, говорят о максимальном снижении сцепления и, следовательно, о максимальном качестве аспекта.

Использование предложенных метрик позволяет получать обоснованные решения в ходе аспектной разработки. Имея метрики, ориентированные на аспект, можно определить еще одну метрику, которая напрямую не измеряет свойства и характеристики отдельного аспекта, а определяет степень оправданности и эффективности введения всего аспектного измерения в общую структуру разрабатываемого программного продукта.

Метрики для объектно-ориентированного тестирования

Рассмотрим проектные метрики, которые, по мнению Р. Байндера (Binder), прямо влияют на тестируемость ОО-систем [33]. Р. Байндер сгруппировал эти метрики в три категории, отражающие важнейшие проектные характеристики.

Метрики инкапсуляции

К метрикам инкапсуляции относятся: «Недостаток связности в методах *LCOM*», «Процент публичных и защищенных *PAP* (Percent Public and Protected)» и «Публичный доступ к компонентным данным *PAD* (Public Access to Data members)».

Метрика 1. Недостаток связности в методах LCOM

Чем выше значение *LCOM*, тем больше состояний надо тестировать, чтобы гарантировать отсутствие побочных эффектов при работе методов.

Метрика 2. Процент публичных и защищенных PAP (Percent Public and Protected)

Публичные атрибуты наследуются от других классов и поэтому видимы для этих классов. Защищенные атрибуты являются специализацией и приватны для определенного подкласса. Эта метрика показывает процент публичных атрибутов класса. Высокие значения *PAP* увеличивают вероятность побочных эффектов в классах. Тесты должны гарантировать обнаружение побочных эффектов.

Метрика 3. Публичный доступ к компонентным данным PAD (Public Access to Data members)

Метрика показывает количество классов (или методов), которые имеют доступ к атрибутам других классов, то есть нарушают их инкапсуляцию. Высокие значения приводят к возникновению побочных эффектов в классах. Тесты должны гарантировать обнаружение таких побочных эффектов.

Метрики наследования

К метрикам наследования относятся «Количество корневых классов *NOR* (Number Of Root classes)», «Коэффициент объединения по входу *FIN*», «Количество детей *NOC*» и «Высота дерева наследования *DIT*».

Метрика 4. Количество корневых классов NOR (Number Of Root classes)

Эта метрика подсчитывает количество деревьев наследования в проектной модели. Для каждого корневого класса и дерева наследования должен разрабатываться набор тестов. С увеличением *NOR* возрастают затраты на тестирование.

Метрика 5. Коэффициент объединения по входу FIN

В контексте ОО систем *FIN* фиксирует множественное наследование. Значение $FIN > 1$ указывает, что класс наследует свои атрибуты и операции от нескольких корневых классов. Следует избегать $FIN > 1$ везде, где это возможно.

Метрика 6. Количество детей NOC

Название говорит само за себя. Метрика заимствована из набора Чидамбера–Кемерера.

Метрика 7. Высота дерева наследования DIT

Метрика заимствована из набора Чидамбера–Кемерера. Методы суперкласса должны повторно тестироваться для каждого подкласса.

В дополнение к перечисленным метрикам Р. Байндер выделил метрики сложности класса (это метрики Чидамбера–Кемерера – *WMC*, *CBO*, *RFC* и метрики для подсчета количества методов), а также метрики полиморфизма.

Метрики полиморфизма

Рассмотрим следующие метрики полиморфизма: «Процентное количество не переопределенных запросов *OVR*», «Процентное количество динамических запросов *DYN*», «Скачок класса *Bounce-C*» и «Скачок системы *Bounce-S*».

Метрика 8. Процентное количество не переопределенных запросов OVR

Процентное количество от всех запросов в тестируемой системе, которые не приводили к перекрытию модулей. Перекрытие может приводить к непредусмотренному связыванию. Высокое значение *OVR* увеличивает возможности возникновения ошибок.

Метрика 9. Процентное количество динамических запросов DYN

Процентное количество от всех сообщений в тестируемой системе, чьи приемники определяются в период выполнения. Динамическое связывание может приводить к непредусмотренному связыванию. Высокое значение *DYN* означает, что для проверки всех вариантов связывания метода потребуется много тестов.

Метрика 10. Скачок класса Bounce-C

Количество скачущих маршрутов, видимых тестируемому классу. Скачущий маршрут — это маршрут, который в ходе динамического связывания пересекает несколько иерархий классов-поставщиков. Скачок может приводить к непредусмотренному связыванию. Высокое значение *Bounce-C* увеличивает возможности возникновения ошибок.

Метрика 11. Скачок системы Bounce-S

Количество скачущих маршрутов в тестируемой системе. В этой метрике суммируется количество скачущих маршрутов по каждому классу системы. Высокое значение *Bounce-S* увеличивает возможности возникновения ошибок.

Контрольные вопросы и упражнения

1. Какие факторы объектно-ориентированных систем влияют на метрики оценки и как проявляется это влияние?
2. Какое влияние оказывает наследование на связность классов?
3. Охарактеризуйте метрики связности классов по данным.
4. Охарактеризуйте метрики связности классов по методам.
5. Какие характеристики объектно-ориентированных систем ухудшают сцепление классов?
6. Объясните, как определить сцепление классов с помощью метрики «зависимость изменения между классами».
7. Поясните смысл метрики локальности данных.
8. Какие метрики входят в набор Чидамбера и Кемерера? Какие задачи они решают?

9. Как можно подсчитывать количество методов в классе?
10. Какие метрики Чидамбера и Кемерера оценивают сцепление классов? Поясните их смысл.
11. Какая метрика Чидамбера и Кемерера оценивает связность класса? Поясните ее смысл.
12. Как добиться независимости метрики *WMC* от реализации?
13. Как можно оценить информационную закрытость класса?
14. Поясните основные недостатки метрики *LCOM* Чидамбера и Кемерера. Какие модификации этой метрики вы знаете? Чем они отличаются друга? Какие дополнительные параметры учитывают?
15. Даны три класса:
    - A (включает приватные атрибуты pa1, pa2 и публичные операции opA1(), opA2());
    - B (включает приватный атрибут pb1 и публичные операции opB1(), opB2());
    - C (включает приватные атрибуты pc1, pc2 и публичные операции opC1(), opC2()).

    Классы B и C являются наследниками класса A. Нарисовать диаграмму классов.

    Вычислить значения метрик *WMC*, *DIT*, *NOC* для всех классов.
16. Даны четыре класса:
    - A (включает приватные атрибуты pa1, pa2 и публичные операции opA1(), opA2());
    - B (включает приватный атрибут pb1 и публичные операции opB1(), opB2());
    - C (включает приватные атрибуты pc1, pc2 и публичные операции opC1(), opC2());
    - D (включает приватные атрибуты pd1, pd2 и публичные операции opD1(), opD2()).

    Классы B и C являются наследниками класса A. При выполнении операции opB1() вызывается операция opD2(). Нарисовать диаграмму классов.

    Вычислить значения метрик CBO, RFC для всех классов.
17. Даны два класса:
    - A (включает приватные атрибуты pa1 типа Integer, pa2 типа Real, pa3 типа Boolean и публичные операции opA1(pa1: Integer), opA2(pa2: Real), opA3(pa1: Integer, pa3:Boolean));
    - B (включает приватный атрибут pb1 и публичные операции opB1(), opB2()).
    - При выполнении операции opA1() вызывается операция opB2 (). Нарисовать диаграмму классов. Вычислить значение метрики LCOM для класса A.

18. Даны два класса:
    - A (включает приватные атрибуты pa1 типа Integer, pa2 типа Real, pa3 типа Boolean и публичные операции opA1(pa1:Integer), opA2 (pa2:Real), opA3 (pa1:Integer, pa3:Boolean));
    - B (включает приватный атрибут pb1 и публичные операции opB1(), opB2 ()).

    При выполнении операции opA1() вызывается операция opB2 (). Нарисовать диаграмму классов. Вычислить значение метрик WMC, CBO, RFC для класса A.
19. Сравните наборы Чидамбера—Кемерера и Лоренца—Кидда. Чем они похожи? В чем различие?
20. На какие цели ориентирован набор метрик Фернандо Абреу?
21. Охарактеризуйте состав набора метрик Фернандо Абреу.
22. Сравните наборы Чидамбера—Кемерера и Фернандо Абреу. Чем они похожи? В чем различие?
23. Сравните наборы Лоренца—Кидда и Фернандо Абреу. Чем они похожи? В чем различие?
24. Дайте характеристику метрик для объектно-ориентированного тестирования.

Глава 13

Примеры объектно-ориентированных процессов разработки

В первой главе рассматривались основы организации процессов разработки ПО. В данной главе внимание сосредоточено на детальном обсуждении унифицированного процесса разработки объектно-ориентированного ПО, на базе которого возможно построение самых разнообразных схем создания программных приложений. Далее описывается содержание XP-процесса экстремальной разработки, являющегося базовым носителем адаптивной, гибкой технологии, применяемой в условиях частого изменения требований заказчика. Дополнительно здесь приводится характеристика другого набирающего силу процесса гибкой разработки — Scrum-процесса. Применение унифицированного процесса и XP-процесса иллюстрируется примерами.

Основные понятия унифицированного процесса разработки

Рассматриваемый подход является развитием спиральной модели Боэма [22, 62, 67, 81]. В этом случае процесс разработки программной системы организуется в виде эволюционно-инкрементного жизненного цикла. Эволюционная составляющая цикла основывается на доопределении требований в ходе работы, инкрементная составляющая — на планомерном приращении реализации требований.

В этом цикле разработка представляется как серия итераций, результаты которых развиваются от начального макета до конечной системы. Каждая итерация включает формирование требований, анализ, проектирование, реализацию и тестирование. Предполагается, что вначале известны не все требования, их дополнение и изменение осуществляется на всех итерациях жизненного цикла. Структура типовой итерации показана на рис. 13.1.

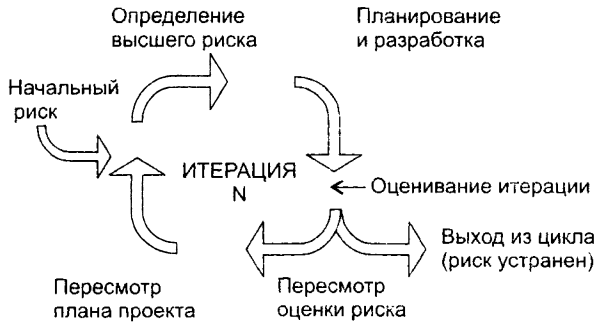


Рис. 13.1. Типовая итерация эволюционно-инкрементного жизненного цикла

Видно, что критерием управления этим жизненным циклом является уменьшение риска. Работа начинается с оценки начального риска. В ходе выполнения каждой итерации риск пересматривается. Риск связывается с каждой итерацией так, что ее успешное завершение уменьшает риск. План последовательности реализаций гарантирует, что наибольший риск устраняется в первую очередь.

Такая методика построения системы нацелена на выявление и уменьшение риска в самом начале жизненного цикла. В итоге минимизируются затраты на уменьшение риска.

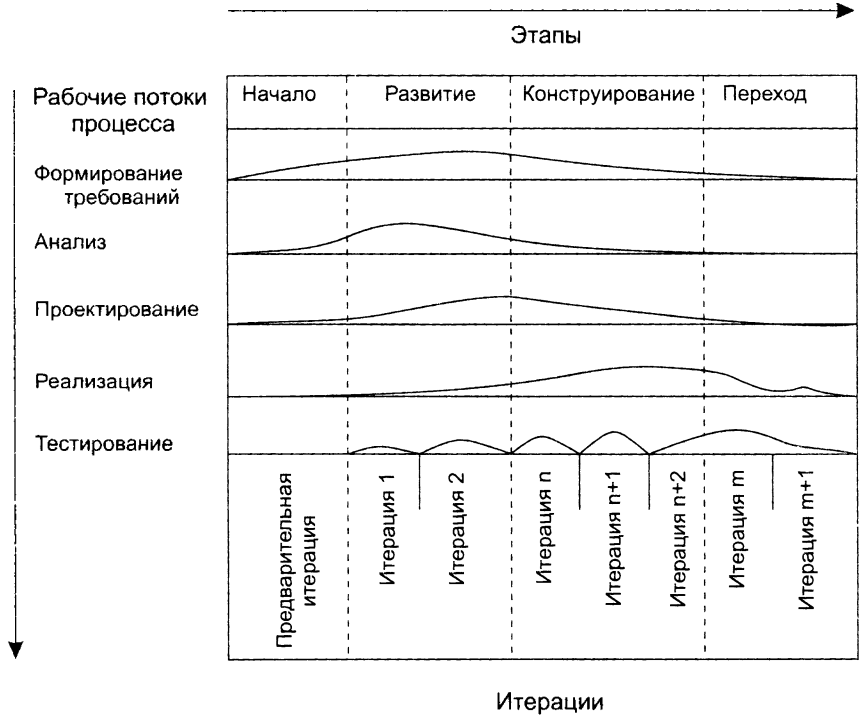


Рис. 13.2. Два измерения унифицированного процесса разработки

Как показано на рис. 13.2, в структуре унифицированного процесса разработки выделяют два измерения:

- горизонтальная ось представляет время и демонстрирует характеристики жизненного цикла процесса;
- вертикальная ось представляет рабочие потоки процесса, которые являются логическими группировками действий.

Первое измерение задает динамический аспект развития процесса в терминах циклов, этапов, итераций и контрольных вех. Второе измерение задает статический аспект процесса в терминах компонентов процесса, рабочих потоков, приводящих к выработке искусственных объектов (артефактов), и участников.

Этапы и итерации

По времени в жизненном цикле процесса выделяют четыре этапа:

- начало (Inception) — спецификация представления продукта;
- развитие (Elaboration) — планирование необходимых действий и требуемых ресурсов;
- конструирование (Construction) — построение программного продукта в виде серии инкрементных итераций;
- переход (Transition) — внедрение программного продукта в среду пользователя (промышленное производство, доставка и применение).

В свою очередь, каждый этап процесса разделяется на итерации. Итерация — это полный цикл разработки, вырабатывающий промежуточный продукт. По мере перехода от итерации к итерации промежуточный продукт инкрементно усложняется, постепенно превращаясь в конечную систему. В состав каждой итерации входят все рабочие потоки - от формирования требований до тестирования. От итерации к итерации меняется лишь удельный вес каждого рабочего потока — он зависит от этапа. На этапе Начало основное внимание уделяется формированию требований, на этапе Развитие — анализу и проектированию, на этапе Конструирование — реализации, на этапе Переход — тестированию. Каждый этап и итерация уменьшает некоторый риск и завершается контрольной вехой. К вехе привязывается техническая проверка степени достижения ключевых целей. По результатам проверки возможна модификация дальнейших действий.

Рабочие потоки процесса

Рабочие потоки процесса имеют следующее содержание:

- Формирование требований — описание того, что система должна делать;
- Анализ — преобразование требований к системе в классы и объекты, выявляемые в предметной области;
- Проектирование — создание статического и динамического представления системы, выполняющего выявленные требования и являющегося эскизом реализации;
- Реализация — производство программного кода, который превращается в исполняемую систему;
- Тестирование — проверка всей системы в целом.

Каждый рабочий поток определяет набор связанных артефактов и действий. Артефакт — это документ, отчет или выполняемый элемент. Артефакт может вырабатываться, обрабатываться или потребляться. Действие описывает задачи — шаги обдумывания, шаги исполнения и шаги проверки. Шаги выполняются участниками процесса (для создания или модификации артефактов).

Между артефактами потоков существуют зависимости. Например, модель Use case, генерируемая в ходе формирования требований, уточняется моделью анализа из процесса анализа, обеспечивается проектной моделью из процесса проектирования, реализуется моделью реализации из процесса реализации и проверяется тестовой моделью из процесса тестирования.

Модели

Модель — наиболее важная разновидность артефакта. Модель упрощает реальность, создается для лучшего понимания разрабатываемой системы. Предусмотрены девять моделей, вместе они покрывают все решения по визуализации, спецификации, конструированию и документированию программных систем:

- бизнес-модель. Определяет абстракцию организации, для которой создается система;
- модель области определения. Фиксирует контекстное окружение системы;
- модель Use Case. Определяет функциональные требования к системе;
- модель анализа. Интерпретирует требования к системе в терминах проектной модели;
- модель проектирования. Определяет словарь проблемы и ее решение;
- модель развертывания. Определяет аппаратную топологию, в которой исполняется система;
- модель реализации. Определяет части, которые используются для сборки и реализации физической системы;
- тестовая модель. Определяет тестовые варианты для проверки системы;
- модель процессов. Определяет параллелизм в системе и механизмы синхронизации.

Технические артефакты

Технические артефакты подразделяются на четыре основных набора:

- набор требований. Описывает, что должна делать система;
- набор проектирования. Описывает, как должна быть организована система;
- набор реализации. Описывает сборку разработанных программных компонентов;
- набор развертывания. Обеспечивает всю информацию о поставляемой конфигурации.

Набор требований группирует всю информацию о том, что система должна делать. Он может включать модель Use Case, модель нефункциональных требований,

модель области определения, модель анализа, а также другие формы выражения нужд пользователя.

Набор проектирования группирует всю информацию о том, как будет конструироваться система при учете всех ограничений (времени, бюджета, традиций, повторного использования, качества и т. д.).

Он может включать модель проектирования, тестовую модель и другие формы выражения сущности системы (например, макеты).

Набор реализации группирует все данные о программных элементах, образующих систему (программный код, файлы конфигурации, файлы данных, программные компоненты, информацию о сборке системы).

Набор развертывания группирует всю информацию об упаковке, отправке, установке и запуске системы.

Этапы унифицированного процесса разработки

Обсудим назначение, цели, содержание и основные итоги каждого этапа унифицированного процесса разработки.

Этап НАЧАЛО (Inception)

Главное назначение этапа — запустить проект.

Цели этапа НАЧАЛО:

- определить область применения создаваемой системы (ее предназначение, границы, интерфейсы с внешней средой, критерий признания — приемки);
- определить критические элементы Use Case системы (основные сценарии поведения, задающие ее функциональность и покрывающие главные проектные решения);
- определить общие черты архитектуры, обеспечивающей основные сценарии, создать демонстрационный макет;
- определить общую стоимость и план всего проекта и обеспечить детализированные оценки для этапа развития;
- идентифицировать основные элементы риска.

Основные действия этапа НАЧАЛО:

- формулировка области применения проекта — выявление требований и ограничений, рассматриваемых как критерий признания конечного продукта;
- планирование и подготовка бизнес-варианта и альтернатив развития для управления риском, определение персонала, проектного плана, а также выявление зависимостей между стоимостью, планированием и полезностью;
- синтезирование предварительной архитектуры, развитие компромиссных решений проектирования; определение решений разработки, покупки и повторного использования, для которых можно оценить стоимость, планирование и ресурсы.

В итоге этапа НАЧАЛО создаются следующие артефакты:

- спецификация представления основных проектных требований, ключевых характеристик и главных ограничений;
- начальная модель Use Case (20% от полного представления);
- начальный словарь проекта;
- начальный бизнес-вариант (содержание бизнеса, критерий успеха – прогноз дохода, прогноз рынка, финансовый прогноз);
- начальное оценивание риска;
- проектный план, в котором показаны этапы и итерации.

Этап РАЗВИТИЕ (Elaboration)

Главное назначение этапа — создать архитектурный базис системы.

Цели этапа РАЗВИТИЕ:

- определить оставшиеся требования, функциональные требования формулировать как элементы Use Case;
- определить архитектурную платформу системы;
- отслеживать риск, устранить источники наибольшего риска;
- разработать план итераций этапа КОНСТРУИРОВАНИЕ.

Основные действия этапа РАЗВИТИЕ:

- развитие спецификации представления, полное формирование критических элементов Use Case, задающих дальнейшие решения;
- развитие архитектуры, выделение ее компонентов.

В итоге этапа РАЗВИТИЕ создаются следующие артефакты:

- модель Use Case (80% от полного представления);
- дополнительные требования (нефункциональные требования, а также другие требования, которые не связаны с конкретным элементом Use Case);
- описание программной архитектуры;
- выполняемый архитектурный макет;
- пересмотренный список элементов риска и пересмотренный бизнес-вариант;
- план разработки для всего проекта, включающий крупноблочный проектный план и показывающий итерации и критерий эволюции для каждой итерации.

Обсудим более подробно главную цель этапа РАЗВИТИЕ — создание архитектурного базиса.

Архитектура объектно-ориентированной системы многомерна — она описывается множеством параллельных представлений. Как показано на рис. 13.3, обычно используется «4 + 1»-представление [67].

Представление Use Case описывает систему как множество взаимодействий с точки зрения внешних актеров. Это представление создается на этапе НАЧАЛО жизненного цикла и управляет оставшейся частью процесса разработки.

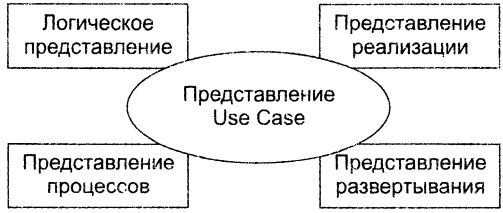


Рис. 13.3. «4 + 1»-представление архитектуры

Логическое представление содержит набор пакетов, классов и отношений. Изначально создается на этапе развития и усовершенствуется на этапе конструирования.

Представление процессов создается для параллельных программных систем, содержит процессы, потоки управления, межпроцессорные коммуникации и механизмы синхронизации. Представление изначально создается на этапе развития, усовершенствуется на этапе конструирования.

Представление реализации содержит модули и подсистемы. Представление изначально создается на этапе развития и усовершенствуется на этапе конструирования.

Представление развертывания содержит физические узлы системы и соединения между узлами. Создается на этапе развития.

В качестве примера рассмотрим порядок создания логического представления архитектуры. Для решения этой задачи исследуются элементы Use Case, разработанные на этапе НАЧАЛО. Рассматриваются экземпляры элементов Use Case — сценарии. Каждый сценарий преобразуется в диаграмму последовательности. Далее в диаграммах последовательности выделяются объекты. Объекты группируются в классы. Классы могут группироваться в пакеты.

Согласно взаимодействиям между объектами, в диаграммах последовательности устанавливаются отношения между классами. Для обеспечения функциональности в классы добавляются атрибуты (они определяют их структуру) и операторы (они определяют поведение). Для размещения общей структуры и поведения создаются суперклассы.

В качестве другого примера рассмотрим разработку плана итераций для этапа КОНСТРУИРОВАНИЕ. Такой план должен задавать управляемую серию архитектурных реализаций, каждая из которых увеличивает свои функциональные возможности, а конечная — покрывает все требования к полной системе. Главным источником информации являются элементы Use Case и диаграммы последовательности. Будем называть их обобщенно — сценариями. Сценарии группируются так, чтобы обеспечивать реализацию определенной функциональности системы. Кроме того, группировки должны устранять наибольший (в данный момент) риск в проекте.

План итераций включает в себя следующие шаги:

1. Определяются все элементы риска в проекте. Устанавливаются их приоритеты.
2. Выбирается группа сценариев, которым соответствует элемент риска с наибольшим приоритетом. Сценарии исследуются. Порядок исследования определяется

не только степенью риска, но и важностью для заказчика, а также потребностью ранней разработки базовых сценариев.

3. В результате анализа сценариев формируются классы и отношения, которые их реализуют.
4. Программируются сформированные классы и отношения.
5. Разрабатываются тестовые варианты.
6. Тестируются классы и отношения. Цель — проверить выполнение функционального назначения сценария.
7. Результаты объединяются с результатами предыдущих итераций, проводится тестирование интеграции.
8. Оценивается итерация. Выделяется необходимая повторная работа. Она назначается на будущую итерацию.

Этап КОНСТРУИРОВАНИЕ (Construction)

Главное назначение этапа — создать программный продукт, который обеспечивает начальные операционные возможности.

Цели этапа КОНСТРУИРОВАНИЕ:

- минимизировать стоимость разработки путем оптимизации ресурсов и устранения необходимости доработок;
- добиться быстрого получения приемлемого качества;
- добиться быстрого получения контрольных версий (альфа, бета и т. д.).

Основные действия этапа КОНСТРУИРОВАНИЕ:

- управление ресурсами, контроль ресурсов, оптимизация процессов;
- полная разработка компонентов и их тестирование (по сформулированному критерию эволюции);
- оценивание реализаций продукта (по критерию признания из спецификации представления).

В итоге этапа КОНСТРУИРОВАНИЕ создаются следующие артефакты:

- программный продукт, готовый для передачи в руки конечных пользователей;
- описание текущей реализации;
- руководство пользователя.

Реализации продукта создаются в серии итераций. Каждая итерация выделяет конкретный набор элементов риска, выявленных на этапе развития. Обычно в итерации реализуется один или несколько элементов Use Case. Типовая итерация включает следующие действия:

1. Идентификация реализуемых классов и отношений.
2. Определение в классах типов данных (для атрибутов) и сигнатур (для операций). Добавление сервисных операций, например операций доступа и управления. Добавление сервисных классов (классов-контейнеров, классов-контроллеров). Реализация отношений ассоциации, агрегации и наследования.
3. Создание текста на языке программирования.

4. Создание (обновление) документации.
5. Тестирование функций реализации продукта.
6. Объединение текущей и предыдущей реализаций. Тестирование интеграции.

Этап ПЕРЕХОД (Transition)

Главное назначение этапа — применить программный продукт в среде пользователей и завершить реализацию продукта.

Этап начинается с предъявления пользователям бета-реализации продукта. В ней обнаруживаются ошибки, они корректируются в последующих β-реализациях. Параллельно решаются вопросы размещения, упаковки и сопровождения продукта. После завершения бета-периода тестирования продукт считается реализованным.

Оценка качества проектирования

Качество проектирования оценивают с помощью объектно-ориентированных метрик, введенных в главе 12.

Этап РАЗВИТИЕ

Качество логического представления архитектуры оценивают по метрикам:

- *WMC* — взвешенные методы на класс;
- *NOC* — количество детей;
- *DIT* — высота дерева наследования;
- *NOM* — суммарное количество методов, определенных во всех классах системы;
- *NC* — общее количество классов в системе.

Метрики *WMC*, *NOC* вычисляются для каждого класса, кроме того, формируются их средние значения в системе. Метрики *DIT*, *NOM*, *NC* вычисляются для всей системы.

Этап КОНСТРУИРОВАНИЕ

На каждой итерации конструирования продукта вычисляются метрики:

- *WMC* — взвешенные методы на класс;
- *NOC* — количество детей;
- *CBO* — сцепление между классами объектов;
- *RFC* — отклик для класса;
- *LCOM* — недостаток связности в методах;
- *CS* — размер класса;
- *NOO* — количество операций, переопределяемых подклассом;
- *NOA* — количество операций, добавленных подклассом;
- *SI* — индекс специализации;
- OS_{avg} — средний размер операции;
- NP_{avg} — среднее количество параметров на операцию;

- NC — общее количество классов в системе;
- LOC_{Σ} — суммарная LOC-оценка всех методов системы;
- DIT — высота дерева наследования;
- NOM — суммарное количество методов в системе.

Метрики WMC, NOC, CBO, RFC, $LCOM$, CS, NOO, NOA, SI, Os_{avg}, NP_{avg} вычисляются для каждого класса, кроме того, формируются их средние значения в системе. Метрики DIT, NOM, NC, LOC_{Σ} вычисляются для всей системы.

На последней итерации дополнительно вычисляется набор метрик *MOOD*, предложенный Абреу:

- MHF — фактор закрытости метода;
- AHF — фактор закрытости атрибута;
- MIF — фактор наследования метода;
- AIF — фактор наследования атрибута;
- POF — фактор полиморфизма;
- COF — фактор сцепления.

Разработка простого интерфейса пользователя для встроенной системы

Для иллюстрации унифицированного процесса рассмотрим фрагмент разработки, выполненной автором совместно с Ольвией Комашиловой. Поставим задачу: разработать оконный интерфейс пользователя, который будет использоваться прикладными программами.

Этап НАЧАЛО

Оконный интерфейс пользователя (WUI — window user interface) — среда, управляемая событиями. Действия в среде инициируются функциями обратного вызова (call-back functions), которые вызываются в ответ на событие — пользовательский ввод. Ядром WUI является цикл обработки событий, который организуется менеджером ввода.

Итак, мы начинаем с рабочего потока формирования требований.

WUI должен обеспечивать следующие типы неперекрывающихся окон:

- простое окно, в которое может быть выведен текст;
- окно меню, в котором пользователь может задать вариант действий — выбор подменю или функции обратного вызова.

Идентификация актеров

Актерами для WUI являются:

- пользователь прикладной программы, использующей WUI;
- администратор системы, управляющий работой WUI.

Внешнее окружение WUI имеет вид, представленный на рис. 13.4.

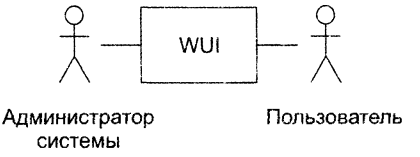


Рис. 13.4. Внешнее окружение WUI

Идентификация элементов Use Case

В WUI могут быть выделены два элемента Use Case:

- управление окнами;
- использование окон.

Диаграмма Use Case для WUI представлена на рис. 13.5.

Рис. 13.5. Диаграмма Use Case для WUI

Описания элементов Use Case

Описание элемента Use Case Управление окнами.

```
Действия начинаются администратором системы. Администратор может создать, удалить
или модифицировать окно.
```

Описание элемента Use Case Использование окон.

```
Действия начинаются пользователем прикладной программы. Обеспечивается возможность
работы с меню и простыми окнами.
```

Этап РАЗВИТИЕ

На этом этапе создаются сценарии для элементов Use Case, разрабатываются диаграммы последовательности (формализующие текстовые представления сценариев), проектируются диаграммы классов и планируется содержание следующего этапа разработки.

Сценарии для элемента Use Case Управление окнами

В элементе Use Case Управление окнами заданы три потока событий — три сценария.

1. Сценарий Создание окна.

```
Устанавливаются координаты окна на экране, стиль рамки окна. Образ окна сохра-
няется в памяти. Окно выводится на экран. Если создается окно меню, содержащее
```

обращение к функции обратного вызова, то происходит установка этой функции. В конце менеджер окон добавляет окно в список управляемых окон WUI.

2. Сценарий Изменение стиля рамки.

   Указывается символ, с помощью которого будет изображаться рамка. Образ окна сохраняется в памяти. Окно перерисовывается на экране.

3. Сценарий Уничтожение окна.

   Менеджер окон получает указание удалить окно. Менеджер окон снимает окно с регистрации (в массиве управляемых окон WUI). Окно снимает отображение с экрана.

Развитие описания элемента Use Case Использование окон

Действия начинаются с ввода пользователем символа. Символ воспринимается менеджером ввода. В зависимости от значения введенного символа выполняется один из следующих вариантов:

при значении ENTER – вариант ОКОНЧАНИЯ ВВОДА;
при переключающем значении – вариант ПЕРЕКЛЮЧЕНИЯ;
при обычном значении – символ выводится в активное окно.

Вариант ОКОНЧАНИЯ ВВОДА:

при активном окне меню выбирается пункт меню. В ответ либо выполняется функция обратного вызова (закрепленная за этим пунктом меню), либо вызывается подменю (соответствующее данному пункту меню);
при активном простом окне выполняется переход на новую строку окна.

Вариант ПЕРЕКЛЮЧЕНИЯ.

При вводе переключающего символа:
ESC — активным становится окно меню;
TAB — активным становится следующее простое окно;
Ctrl-E — все окна закрываются, и сеанс работы заканчивается.

Далее из описания элемента Use Case Использование окон выделяются два сценария: Использование простого окна и Использование окна меню. Будем считать, что рабочий поток формирования требований завершен и инженер-формирователь, отвечающий за формирование требований, передает полную диаграмму Use Case инженеру-аналитику, отвечающему за анализ требований.

Аналитик использует диаграмму Use Case как исходные данные для своей работы. Его работа соответствует рабочему потоку анализа требований: преобразовать сценарии элементов Use Case в диаграммы последовательности — за счет этого достигается формализация описаний, требуемая для выполнения следующего рабочего потока — потока проектирования (построения диаграмм классов).

Для создания диаграмм последовательности аналитик проводит грамматический разбор каждого сценария из элемента Use Case: значащие существительные превращаются в обобщенные объекты и параметры сообщений, а значащие глаголы — в сообщения, пересылаемые между обобщенными объектами.

Диаграммы последовательности

Рассмотрим порядок построения диаграммы последовательности для сценария Создание окна из элемента Use Case Управление окнами.

В этом сценарии пять предложений, значит, аналитика ожидают пять шагов построения диаграммы.

Анализ первого предложения «Устанавливаются координаты окна на экране, стиль рамки окна» выявляет два обобщенных объекта, то есть двух участников взаимодействия: окно и экран. В отношении координат и стиля рамки аналитик принимает решение: до уровня объекта эти существительные не дотягивают, резонно их считать атрибутами объекта и использовать как параметры сообщения. Правда, тут аналитика начинают «терзать смутные сомнения» — ему кажется, что инженер-формирователь что-то упустил. Ведь непонятно, кто занимается «установкой». Тогда он просматривает предыдущее, более общее описание элемента Use Case Управление окнами и находит недостающее существительное — это администратор системы. Администратор становится третьим участником взаимодействия. Теперь следует разобраться с сообщениями. Аналитик понимает, что для каждого сообщения нужно выявить: имя, параметры (если повезет), откуда и куда посылается. Здесь нет никаких сомнений: сообщение только одно, его имя является производным от глагола устанавливаются, параметры — это координаты и стиль рамки, посылается сообщение от линии жизни администратора к линии жизни окна (рис. 13.6).

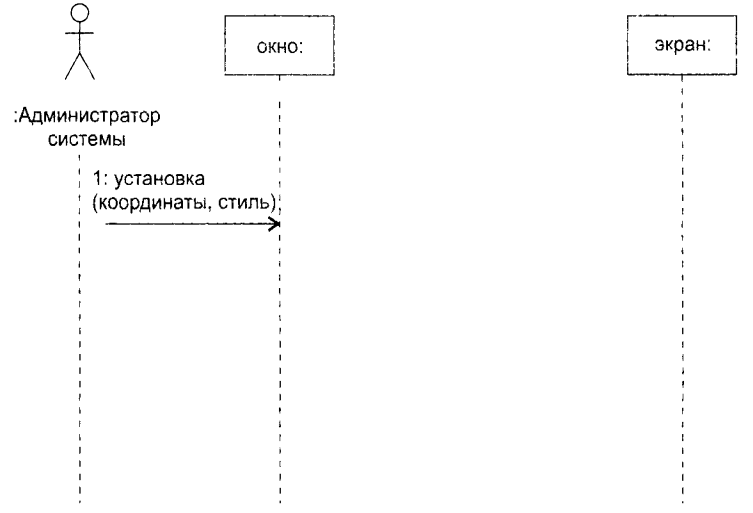


Рис. 13.6. Первый шаг строительства диаграммы последовательности Создание окна

Во втором предложении «Образ окна сохраняется в памяти» имеются три существительных. Окно уже введено в диаграмму. Образ аналитик счел обобщенным представлением всех атрибутов окна, а память находится вне «контура» программной

системы (аналитик решил, что это компьютерный ресурс, выделяемый обобщенному объекту окно). На сообщение указывает глагол сохраняется, посылается сообщение окном самому себе (рис. 13.7).

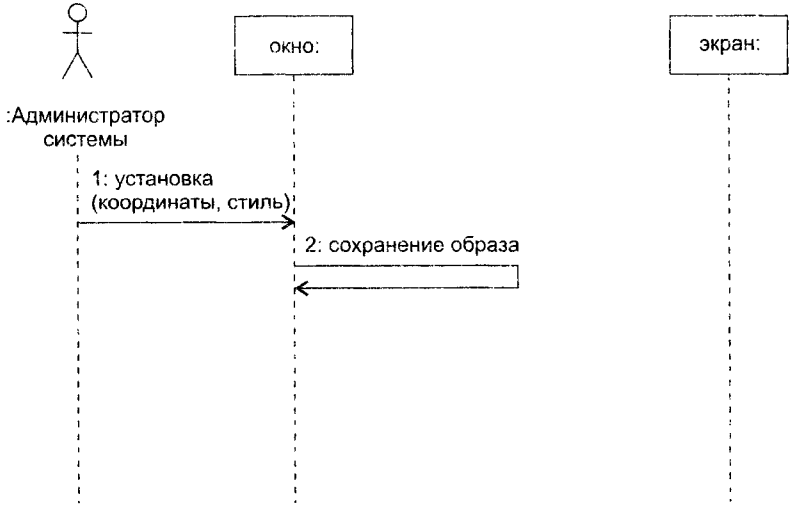


Рис. 13.7. Второй шаг строительства диаграммы последовательности Создание окна

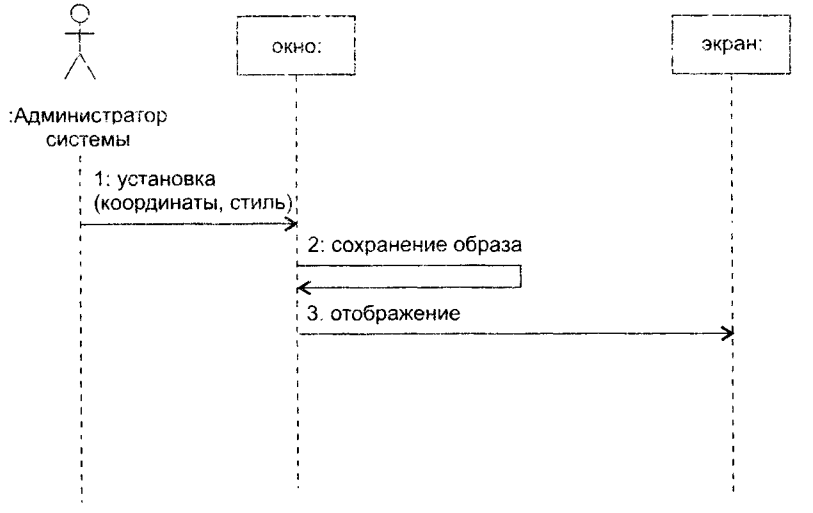


Рис. 13.8. Третий шаг строительства диаграммы последовательности Создание окна

В третьем предложении «Окно выводится на экран» нет кандидатов на новые обобщенные объекты, сообщение, именуемое глаголом выводится, посылается окном экрану (рис. 13.8).

Обработка четвертого предложения «Если создается окно меню, содержащее обращение к функции обратного вызова, то происходит установка этой функции» потребовала от аналитика некоторого напряжения. Пришлось обратиться к истокам, то есть внимательно исследовать назначение будущего приложения. Исследование вылилось в утверждение: функция обратного вызова является методом специализированного окна категории меню, причем содержание этого метода может меняться и требует самонастройки окна посредством механизма косвенной адресации. Выводы на базе утверждения оказались простыми: новых объектов в предложении нет, есть только сообщение установка с параметром функция обратного вызова, инициирующее самонастройку окна (рис. 13.9).

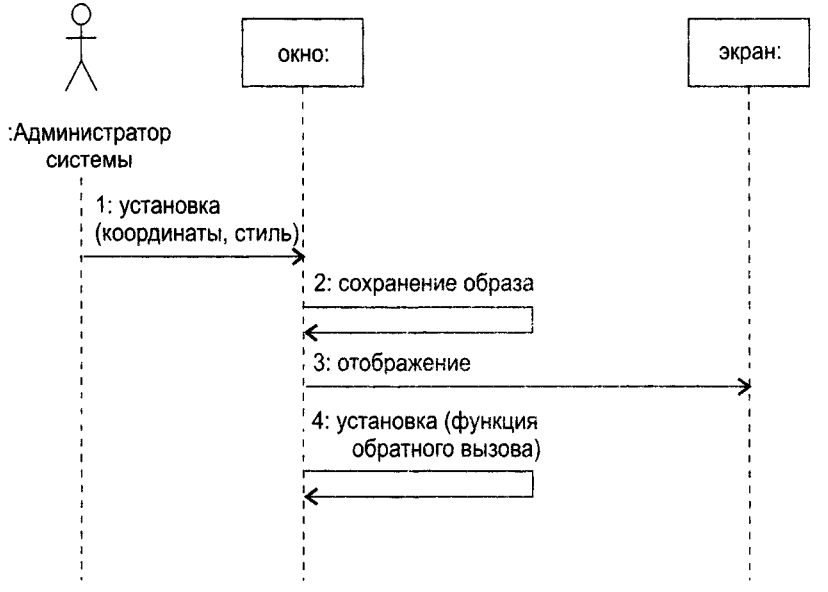


Рис. 13.9. Четвертый шаг строительства диаграммы последовательности Создание окна

Заключительное, пятое предложение «В конце менеджер окон добавляет окно в список управляемых окон WUI» привело к простому и понятному грамматическому разбору: в диаграмму нужно добавить новый обобщенный объект менеджер окон, которому окно посылает сообщение добавить окно (рис.13.10).

Аналогичным образом аналитик строит другие диаграммы последовательности, обрабатывая соответствующие сценарии (рис. 13.11–13.14).

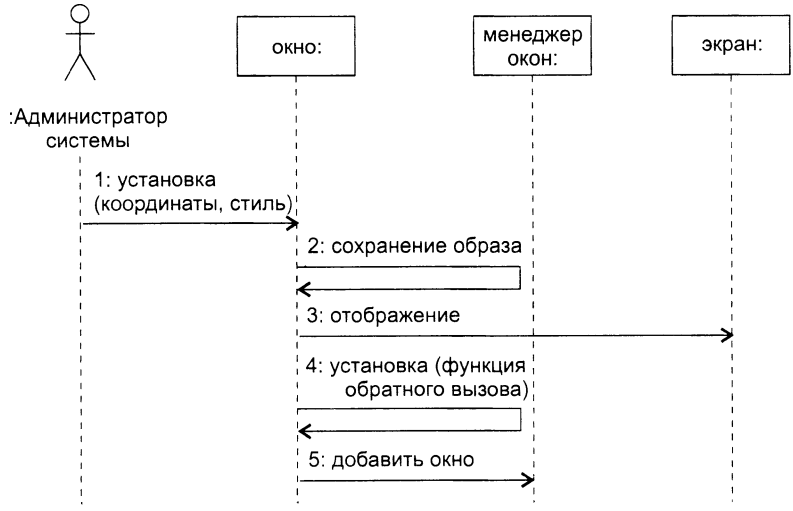


Рис. 13.10. Заключительный шаг строительства диаграммы последовательности Создание окна

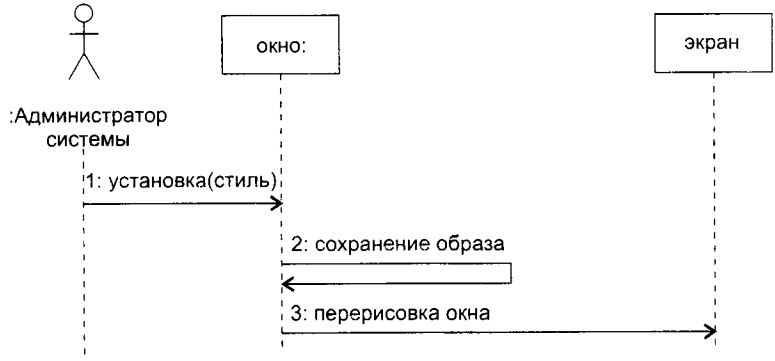


Рис. 13.11. Диаграмма последовательности Изменение стиля рамки

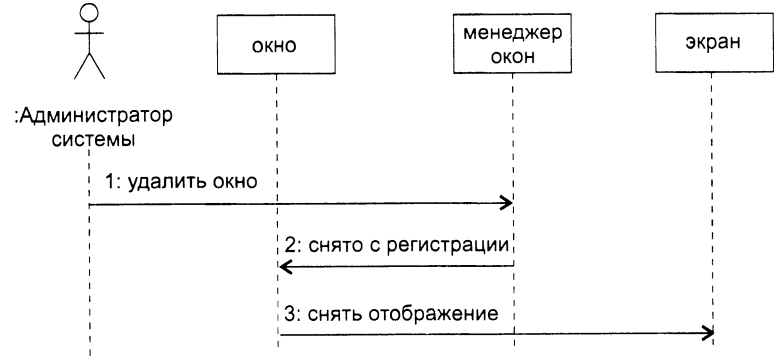


Рис. 13.12. Диаграмма последовательности Уничтожение окна

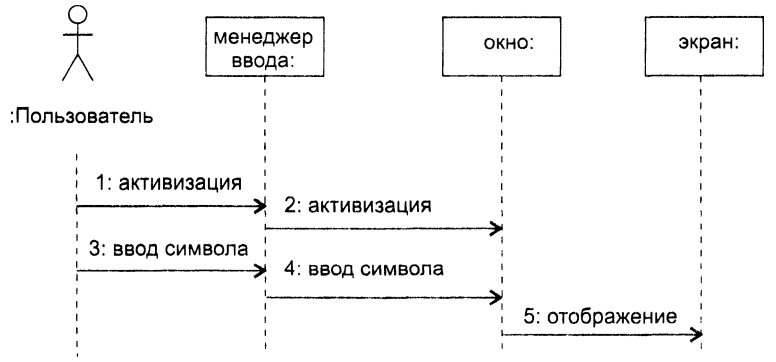


Рис. 13.13. Диаграмма последовательности Использование простого окна

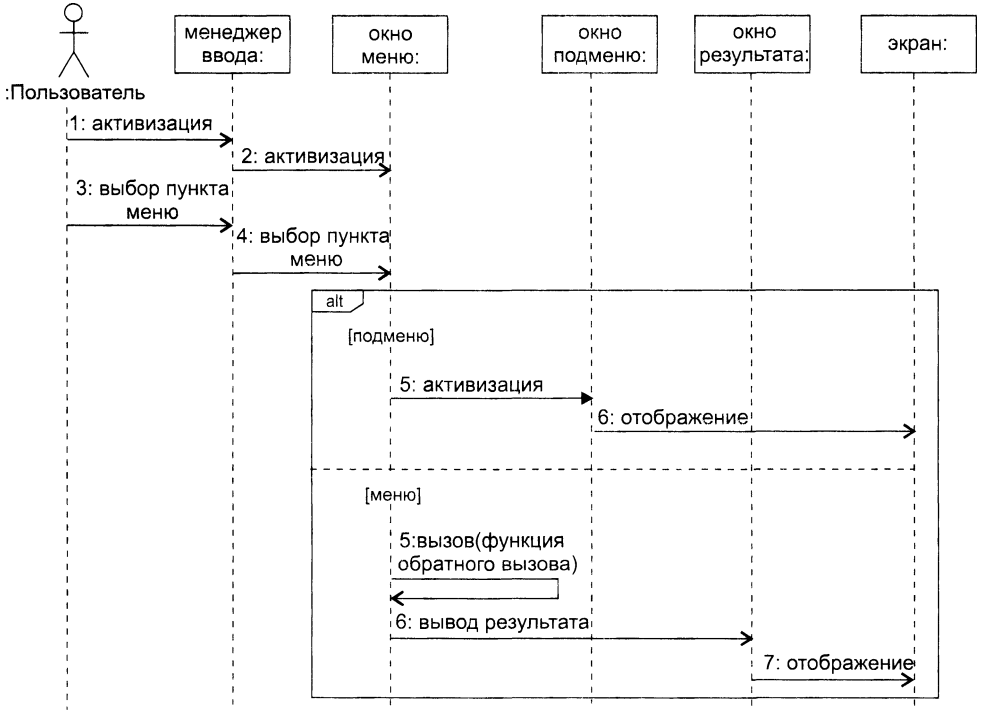


Рис. 13.14. Диаграмма последовательности Использование окна меню

Создание классов

С окончанием строительства диаграмм последовательности завершается рабочий поток анализа требований. Полученные диаграммы аналитик передает инженеру-проектировщику, и начинается рабочий поток проектирования. Суть проектирования заключается в выявлении такой структурной организации классов, которая обеспечивает выполнение всех требований.

Работа по созданию классов (и включению их в диаграмму классов) требует изучения содержания всех диаграмм последовательности. Проводится она в три этапа.

На первом этапе выявляются и именуются классы. Для этого просматривается каждая диаграмма последовательности. Любой обобщенный объект в этой диаграмме должен принадлежать конкретному классу, для которого надо придумать имя. Например, резонно предположить, что объекту менеджер окон должен соответствовать класс Window_Manager, поэтому класс Window_Manager следует ввести в диаграмму. Конечно, если в другой диаграмме последовательности опять появится подобный объект, то дополнительный класс не образуется.

На втором этапе выявляются операции классов. На диаграмме последовательности такая операция соответствует стрелке (и имени) сообщения, указывающей на линию жизни объекта класса. Например, если к линии жизни объекта менеджер окон подходит стрелка сообщения добавить окно, то в класс Window_Manager нужно ввести операцию add_to_list().

На третьем этапе определяются отношения ассоциации между классами — они обеспечивают пересылки сообщений между соответствующими объектами.

Первоначальная диаграмма классов системы, полученная простой обработкой диаграмм последовательности, представлена на рис. 13.15.

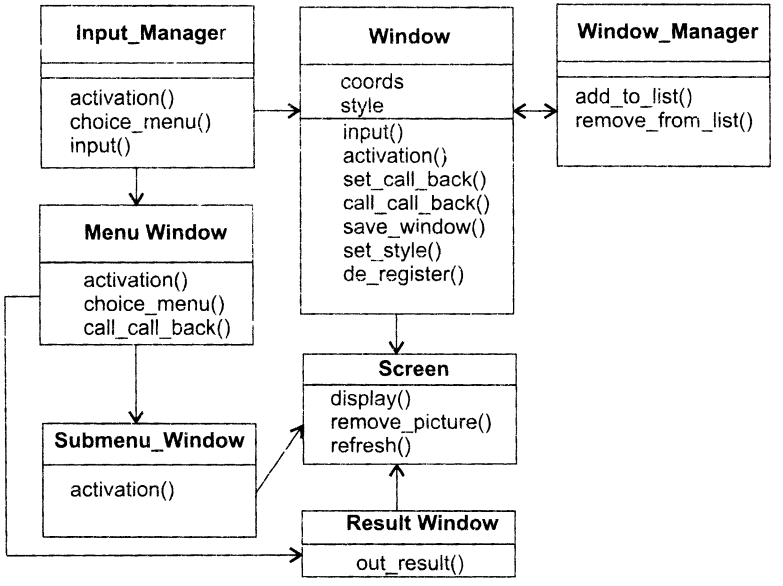


Рис. 13.15. Начальная диаграмма классов WUI

В ней только у одного класса существуют атрибуты. Эти атрибуты выявлены как параметры сообщения на диаграмме последовательности с рис. 13.10. Все остальные атрибуты придется придумывать инженеру-проектировщику. Собственно говоря, с обдумывания начальной диаграммы классов и начинается творческая работка проектировщика. Опираясь на собственный опыт, он ищет пути оптимизации

структуры, наполняет капсулы классов дополнительными подробностями, отражающими специфику и конкретику языка и среды программирования, которые будут задействованы в следующем рабочем потоке — потоке реализации.

В нашем примере инженер находит много общего в классах, соответствующих различным категориям окон, и принимает решение: трансформировать набор классов на основе механизма наследования. Итогом этой трансформации становятся следующие классы:

- *Root_Window* — абстрактный суперкласс. Он определяет минимальные обязанности, которые должен реализовать любой тип окна (посылка символа в окно, перевод окна в активное/пассивное состояние, перерисовка окна, возврат информации об окне). Все остальные классы окон являются его потомками.
- Window — класс, объектами которого являются простые окна;
- Menu — класс, объектами которого являются окна меню. Этот класс является потомком класса Window;

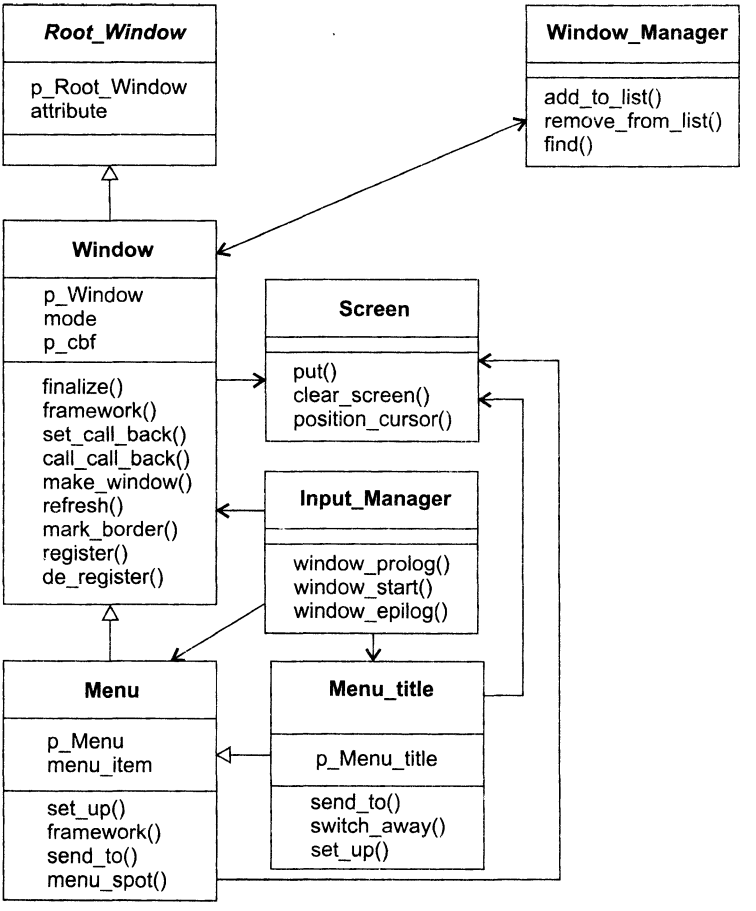


Рис. 13.16. Оптимизированная диаграмма классов WUI

- Menu_title — класс, объектом которого является окно главного меню. Класс является потомком класса Menu,
- Screen — класс, объектом которого является экран. Этот класс обеспечивает позиционирование курсора, вывод изображения на экран дисплея, очистку экрана;
- Input_Manager — объект этого класса управляет взаимодействием между пользователем и окнами интерфейса. Его обязанности: начальные установки среды WUI, запуск цикла обработки событий, закрытие среды WUI;
- Window_Manager — осуществляет общее управление окнами, отображаемыми на экране. Используется менеджером ввода для получения доступа к конкретному окну.

Оптимизированная диаграмма классов WUI показана на рис. 13.16. Нетрудно заметить, что существенно обновился состав как атрибутов, так и операций классов. Для учета изменений корректируют исходные диаграммы последовательности. Эта корректировка, кстати, косвенно проверяет правомочность проведенных изменений.

В этом месте уместно вспомнить, что проектирование считают фазой «взращивания» качества. Для подтверждения этого тезиса стартует механизм количественной оценки качества. Результаты начальной оценки качества итогов проектирования сведены в табл. 13.1.

Таблица 13.1. Результаты начальной оценки качества WUI

Метрика	Input_ Manager	Window_ Manager	Screen	Root_ Window	Window	Menu	Menu_title	Среднее значение
WMC	3	3	3	0	9	4	3	3,57
NOC	–	–	–	1	1	1	0	0,43
Метрики, вычисляемые для системы								
DIT	3							
NC	7							
NOM	25							

Планирование итераций конструирования

На данном шаге составляется план итераций, который определяет порядок действий на этапе конструирования. Цель каждой итерации — уменьшить риск разработки конечного продукта. Для создания начального плана анализируются элементы Use Case, их сценарии и диаграммы последовательности. Устанавливается приоритет их реализации. При завершении каждой итерации будет повторно вычисляться риск. Оценка риска может привести к необходимости обновления плана итераций.

Положим, что максимальный риск связан с реализацией элемента Use Case Управление окнами, причем наиболее опасна разработка сценария Создание окна, среднюю опасность несет сценарий Уничтожение окна и малую опасность — Изменение стиля рамки.

В связи с этими соображениями начальный план итераций принимает вид:

Итерация 1 — реализация сценариев элемента Use Case Управление окнами:

1. Создание окна.
2. Уничтожение окна.
3. Изменение стиля рамки.

Итерация 2 — реализация сценариев элемента Use Case Использование окон:

1. Использование простого окна.
2. Использование окна меню.

Этап КОНСТРУИРОВАНИЕ

Рассмотрим содержание итераций на этапе конструирования.

Итерация 1 — реализация сценариев элемента Use Case Управление окнами

Для реализации сценария Создание окна программируются следующие операции класса Window:

- framework() — создание каркаса окна;
- register() — регистрация окна;
- set_call_back() — установка функции обратного вызова;
- make_window() — задание видимости окна.

Далее реализуются операции общего управления окнами, методы класса Window_Manager:

- add_to_list() — добавление нового окна в массив управляемых окон;
- find() — поиск окна с заданным переключающим символом.

Программируются операции класса Input_Manager:

- window_prolog() — инициализация WUI;
- window_start() — запуск цикла обработки событий;
- window_epilog() — закрытие WUI.

В ходе реализации перечисленных операций выясняется необходимость и программируется содержание вспомогательных операций:

1) в классе Window_Manager:
   - write_to() — форматный вывод сообщения в указанное окно;
   - hide_win() — удаление окна с экрана;
   - switchAwayFromTop() — подготовка окна к переходу в пассивное состояние;
   - switch_to_top() — подготовка окна к переходу в активное состояние;
   - window_fatal() — формирование донесения об ошибке;
   - top() — переключение окна в активное состояние;
   - send_to_top() — посылка символа в активное окно.

2) в классе Window:

- put() — три реализации для записи в окно символьной, строковой и числовой информации;
- create() — создание макета окна (используется операцией framework);
- position() — изменение позиции курсора в окне;
- about() — возврат информации об окне;
- switch_to() — пометка активного окна;
- switch_away() — пометка пассивного окна;
- send_to() — посылка символа в окно для обработки.

Второй шаг первой итерации ориентирован на реализацию сценария Уничтожение окна. Основная операция — finalize() (метод класса Window), она выполняет разрушение окна. Для ее обеспечения создаются вспомогательные операции:

- de_register() — удаление окна из массива управляемых окон;
- remove_from_list() (метод класса Window_Manager) — вычеркивание окна из регистра.

Для реализации сценария Изменение стиля рамки создаются операции в классе Window:

- mark_border() — построение новой рамки окна;
- refresh() — перерисовка окна на экране.

В конце итерации создаются операции класса Screen:

- clear_screen() — очистка экрана;
- position_cursor() — позиционирование курсора;
- put() — вывод на экран дисплея строк, символов и чисел.

Результаты оценки качества первой итерации представлены в табл. 13.2.

Таблица 13.2. Оценки качества WUI после первой итерации

Метрика	Input_ Manager	Window_ Manager	Screen	Root_ Window	Window	Среднее значение
WMC	0,12	0,42	0,11	0	0,83	0,3
NOC	–	–	–	1	0	0,2
CBO	3	3	0	1	2	1,8
RFC	6	11	0	0	23	8
LCOM	3	0	5	0	0	1,6
CS	3/2	10/8	5/1	0/2	18/22	7,2/7
NOO	–	–	–	0	0	0
NOA	–	–	–	0	18	3,6
SI	–	–	–	0	0	0
OS_{use}	4	4,2	2,2	0	4,6	3

Метрика	Input_ Manager	Window_ Manager	Screen	Root_ Window	Window	Среднее значение
NP_{avg}	0	1,3	1	0	2,4	0,9
Метрики, вычисляемые для системы						
DIT	1					
NC	5					
NOM	35					
LOC_v	148					

Итерация 2 — реализация сценариев элемента Use Case Использование окон

На этой итерации реализуем методы классов Menu и Menu_title, а также добавим необходимые вспомогательные методы в класс Window.

Отметим, что операции, обеспечивающие сценарий Использование простого окна, в основном уже реализованы (на первой итерации). Осталось запрограммировать следующие операции — методы класса Window:

- call_call_back() — вызов функции обратного вызова;
- initialize() — управляемая инициализация окна;
- clear() — очистка окна с помощью пробелов;
- new_line() — перемещение на следующую строку окна.

Для обеспечения сценария Использование окна меню создаются следующие операции:

1) в классе Menu:
   - framework() — создание каркаса окна-меню;
   - send_to() — обработка пользовательского ввода в окно-меню;
   - menu_spot() — выделение выбранного элемента меню;
   - set_up() — заполнение окна-меню именами элементов;
   - get_menu_name() — возврат имени выбранного элемента меню;
   - get_cur_selected_details() — возврат указателя на выбранное окно и функцию обратного вызова.
2) в классе Menu_title:
   - send_to() — выделение новой строки меню или вызов функции обратного вызова ;
   - switch_away() — возврат в базовое окно-меню более высокого уровня;
   - set_up() — установки окна меню-заголовка.

Результаты оценки качества второй итерации представлены в табл. 13.3.

Сравним оценки качества первой и второй итераций.

1. Рост системных оценок LOC_{Σ}, *NOM*, а также средних значений метрик *WMC*, *RFC*, *CS*, *CBO* и *NOO* — свидетельство возрастания сложности продукта.
2. Увеличение значения *DIT* и среднего значения *NOC* говорит об увеличении возможности многократного использования классов.
3. На второй итерации в среднем была ослаблена абстракция классов, о чем свидетельствует увеличение средних значений *NOC*, *NOA*, *SI*.
4. Рост средних значений OS_{avg} и NP_{avg} говорит о том, что сотрудничество между объектами усложнилось.
5. Среднее значение *CBO* указывает на увеличение сцепления между классами (это нежелательно), зато снижение среднего значения *LCOM* свидетельствует, что связность внутри классов увеличилась (таким образом, снизилась вероятность ошибок в ходе разработки).

Вывод: качество разработки в среднем возросло, так как, несмотря на увеличение средних значений сложности и сцепления (за счет добавления в иерархию наследования новых классов), связность внутри классов была увеличена.

В практике проектирования достаточно типичны случаи, когда в процессе разработки меняются исходные требования или появляются дополнительные требования к продукту. Предположим, что в конце второй итерации появилось дополнительное требование — ввести в WUI новый тип окна — диалоговое окно. Диалоговое окно должно обеспечивать не только вывод, но и ввод данных, а также их обработку.

Таблица 13.3. Оценки качества WUI после второй итерации

Метрика	Input_Manager	Window_Manager	Screen	Root_Window	Window	Menu	Menu_title	Среднее значение
WMC	0,12	0,42	0,11	0	0,98	0,33	0,27	0,32
NOC	–	–	–	1	1	1	0	0,4
CBO	3	3	0	1	2	2	3	2
RFC	6	11	0	0	27	9	12	9,4
LCOM	3	0	5	0	0	0	0	1,1
CS	3/2	10/8	5/1	0/2	22/22	28/24	11/12	11,3/10,1
NOO	–	–	–	0	0	2	3	0,7
NOA	–	–	–	0	22	6	0	4
SI	–	–	–	0	0	0,23	0,46	0,1
OS_{avg}	4	4,2	2,2	0	4,45	4,13	9	4,0
NP_{avg}	0	1,3	1	0	2,18	4,63	1,67	1,5
Метрики, вычисляемые для системы								
DIT	3							
NC	7							
NOM	48							
LOC_{Σ}	223							

Для реализации этого требования вводится третья итерация конструирования.

Итерация 3 — разработка диалогового окна

Шаг 1. Спецификация представления диалогового окна.

На этом шаге фиксируется представление заказчика об обязанностях диалогового окна. Положим, что оно имеет следующий вид:

1. Диалоговое окно накапливает посылаемые в него символы, отображая их по мере получения.
2. При получении символа конца сообщения (ENTER) полная строка текста принимается в функцию обратного вызова, связанную с диалоговым окном.
3. Функция обратного вызова реализует обслуживание, требуемое пользователю.
4. Функция обратного вызова обеспечивается прикладным программистом.

Шаг 2. Модификация диаграммы Use Case для WUI.

Очевидно, что дополнительное требование приводит к появлению дополнительного элемента Use Case, который находится в отношении «расширяет» с базовым элементом Use Case Использование окон.

Диаграмма Use Case принимает вид, представленный на рис. 13.17.

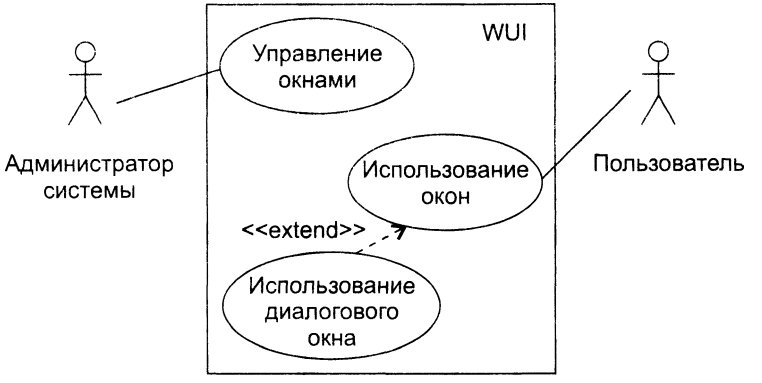


Рис. 13.17. Модифицированная диаграмма Use Case для WUI

Шаг 3. Описание элемента Use Case Использование диалогового окна.

```
Действия начинаются с ввода пользователем переключающего символа, активизирующего
данный тип окна. Символ воспринимается менеджером ввода. Далее пользователь вводит
данные, которые по мере поступления отображаются в диалоговом окне. После нажатия
пользователем символа окончания ввода (ENTER) данные передаются в функцию обратно-
го вызова как параметр. Выполняется функция обратного вызова, результат выводится
в простое окно результата.
```

Шаг 4. Диаграмма последовательности Использование диалогового окна.

Диаграмма последовательности для сценария Использование диалогового окна показана на рис. 13.18.

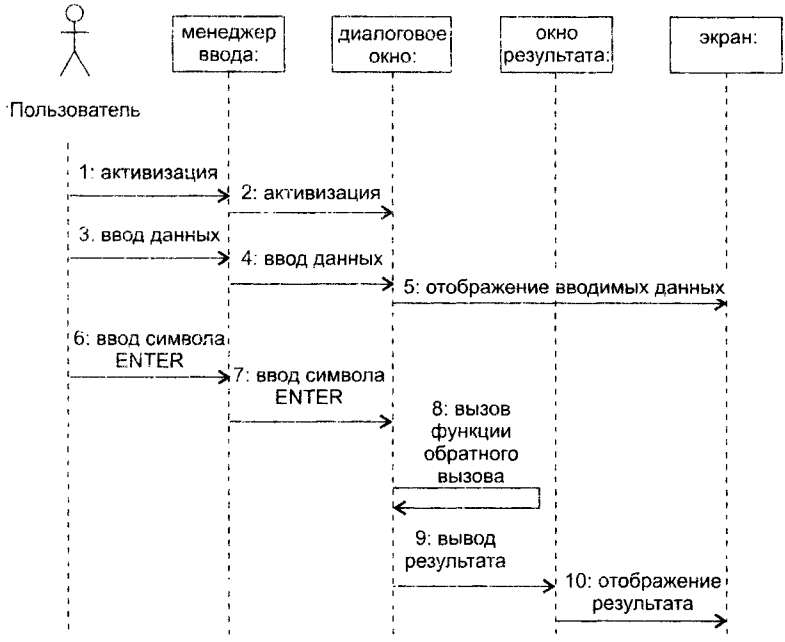


Рис. 13.18. Диаграмма последовательности Использование диалогового окна

Шаг 5. Создание класса

Для реализации сценария Использование диалогового окна создается новый класс Dialog, который является наследником класса Window. Объекты класса Dialog образуют диалоговые окна.

Класс Dialog переопределяет следующие операции, унаследованные от класса Window:

- framework() – формирование диалогового окна. Параметры операции: имя диалогового окна, координаты, ширина окна, заголовок окна и ссылка на функцию обратного вызова. Операция создает каркас окна, устанавливает для него функцию обратного вызова, делает окно видимым и регистрирует его в массиве управляемых окон;
- send_to() – обрабатывает пользовательский ввод, посылаемый в диалоговое окно. Окно запоминает символы, вводимые пользователем, а после нажатия пользователем клавиши ENTER вызывает функцию обратного вызова, обрабатывающую эти данные.

Конечное представление иерархии классов WUI показано на рис. 13.19. Результаты оценки качества проектного решения (в конце третьей итерации) сведены в табл. 13.4. Динамика изменения значений для метрик класса показана в табл. 13.5.

Сравним средние значения метрик второй и третьей итераций:

1. Общая сложность WUI возросла (увеличились значения LOC_C, *NOM* и *NC*), однако повысилось качество классов (уменьшились средние значения *WMC* и *RFC*).

2. Увеличились возможности многократного использования классов (о чем свидетельствует рост среднего значения *NOC* и уменьшение среднего значения *WMC*).

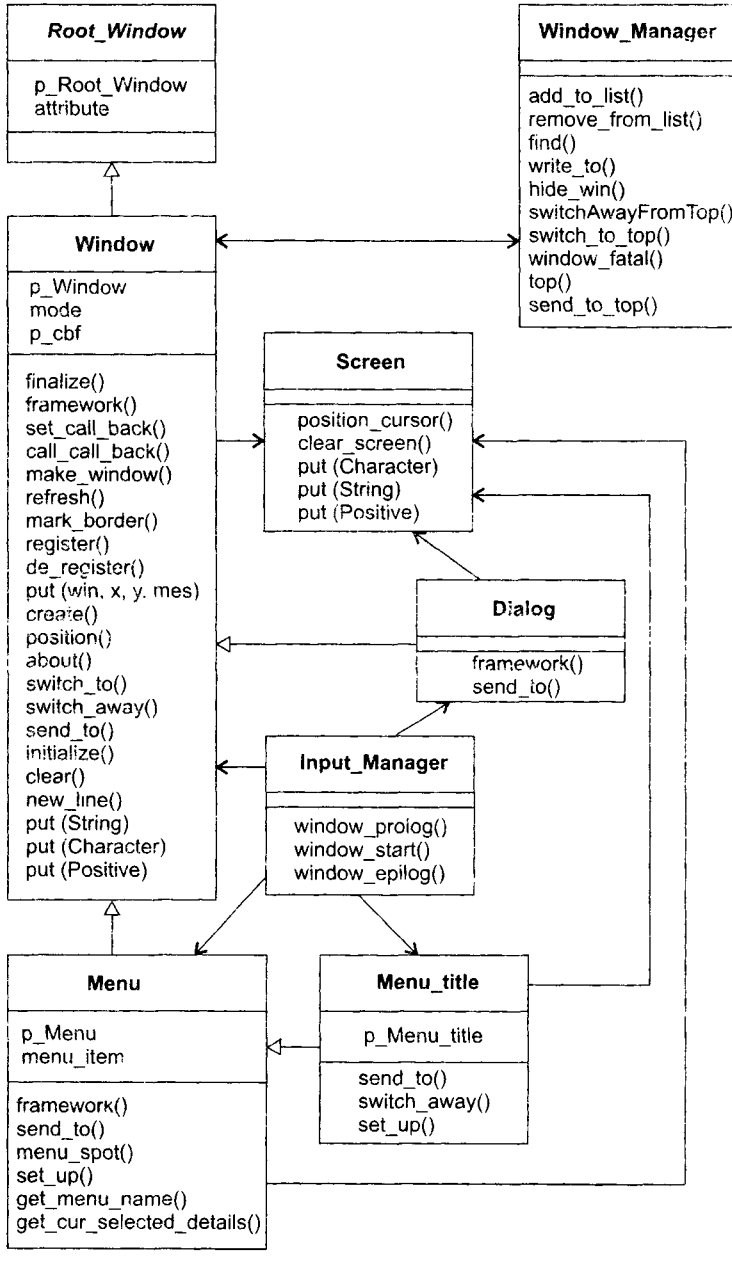


Рис. 13.19. Конечная диаграмма классов WUI

3. Возросла средняя связность класса (уменьшилось среднее значение метрики *LCOM*).
4. Уменьшилось среднее значение сцепления класса (сохранилось среднее значение *CBO* и уменьшилось среднее значение *RFC*).

Вывод: качество результатов проектирования стало выше.

Таблица 13.4. Оценки качества WUI после третьей итерации

Метрика	Input_ Manager	Window Manager	Screen	Root_ Window	Window	Menu	Menutitle	Dialog	Среднее значение
WMC	0,12	0,42	0,11	0	0,98	0,33	0,27	0,23	0,31
NOC	–	–	–	1	2	1	0	0	0,5
CBO	3	3	0	1	2	2	3	2	2
RFC	6	11	0	0	27	9	12	7	9,1
LCOM	3	0	5	0	0	0	0	0	1
CS	3/2	10/8	5/1	0/2	22/22	28/24	11/12	24/14	12,2/10,6
NOO	–	–	–	0	0	2	3	2	0,9
NOA	–	–	–	0	22	6	0	0	3,5
SI	–	–	–	0	0	0,23	0,46	0,27	0,14
OS_{avg}	4	4,2	2,2	0	4,45	4,13	9	11,5	4,9
NP_{avg}	0	1,3	1	0	2,18	4,63	1,67	4	1,8
Метрики, вычисляемые для системы									
DIT	3								
NC	8								
NOM	50								
LOC_{Σ}	246								

Таблица 13.5. Средние значения метрик класса на разных итерациях

Метрика	Итерация 1	Итерация 2	Итерация 3
WMC	0,3	0,32	0,31
NOC	0,2	0,4	0,5
CBO	1,8	2	2
RFC	8	9,4	9,1
LCOM	1,6	1,1	1
CS	7,2/7	11,3/10,1	12,2/10,6
NOO	0	0,7	0,9
NOA	3,6	4	3,5
SI	0	0,1	0,14
OS_{avg}	3	4,0	4,9

Метрика	Итерация 1	Итерация 2	Итерация 3
NP_{avg}	0,9	1,5	1,8
DIT	1	3	3
NC	5	7	8
NOM	35	48	50
LOC_{Σ}	148	223	246

Таблица 13.6. Значения метрик Абреу для WUI

Метрика	Значение
MHF	0.49
AHF	0.49
MIF	0 49
AIF	0.29
POF	0.69
COF	0.25

На последней итерации рассчитаны значения интегральных метрик Абреу, они представлены в табл. 13.6. Эти данные также характеризуют качество проектного решения и подтверждают наши выводы.

Метрические данные проекта помещают в метрический базис фирмы-разработчика, тем самым обеспечивается возможность их использования в последующих разработках.

Разработка системы управления торговым автоматом

В этом разделе приводится второй пример выполнения объектно-ориентированной разработки в рамках унифицированного процесса — мощного инструмента, предложенного программной индустрии отделением Rational корпорации IBM. Надеемся, что он будет способствовать формированию панорамной точки зрения читателя.

Этап НАЧАЛО

Цель этого этапа — создать начальную модель требований, описывающую желаемое поведение системы, а также оценить затраты на разработку, факторы риска.

Спецификация требований заказчика

Поставим задачу — разработать систему управления торговым автоматом, предназначенным для продажи продуктов питания. Система должна автоматизировать функции работы этого автомата, обеспечивать продажу продуктов, предусматривать ограничение доступа к сервисным функциям автомата, регистрировать выполняемые действия в системном журнале.

Автоматизированную систему управления предполагается устанавливать на торговые автоматы, размещаемые в общественных местах (институты, гостиницы,

клубы и др.). Предполагается наличие в автомате аппаратных средств, совместимых с x86, и наличие операционной системы *Windows NT*. Программное обеспечение должно быть реализовано на языке программирования *Object Pascal* в визуальной среде *Borland Delphi Studio*.

Для данной спецификации построим начальную диаграмму Use Case.

Идентификация актеров

Актерами для системы управления торговым автоматом являются:

- Клиент (Client) — тот, кто желает приобрести какой-нибудь продукт питания в автомате.
- Техник (Technican) — работник обслуживающей фирмы, выполняющий сервисные операции.

Идентификация и краткое описание элементов Use Case

Функциональный состав системы представляется следующими элементами Use Case:

- Buy — покупка товара — инициализируется актером Клиент (Client) и описывает действия, непосредственно связанные с сессией покупки продукта в автомате.
- Maintain — обслуживание — инициализируется актером Техник (Technican) и описывает действия, непосредственно связанные с проведением обслуживания автомата. Техник может выполнять такие сервисные операции, как ремонт автомата, инкассация — изъятие денег, загрузка новых продуктов и др.

Начальная диаграмма Use Case имеет вид, представленный на рис. 13.20.

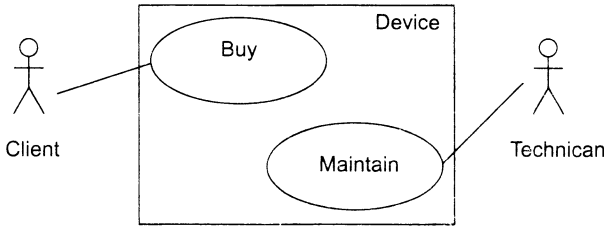


Рис. 13.20. Начальная диаграмма Use Case

Проведем предварительную оценку стоимости и затрат на разработку (табл. 13.7

Таблица 13.7. Предварительная оценка параметров разработки

Функция	Лучш.	Вероят.	Худш.	Ожид. [LOC]	Уд. стоимость [$/LOC]	Стоимость [$]	Произв. [LOC/чел.-мес.]	Затраты [чел.-мес.]
Модуль покупки	1500	1300	1100	1300	1.0	1300	385	3.4
Модуль обслуживания	1000	850	600	833	1.0	833	600	1.4
Итого				2133		$2133		4,8

Методика и примеры оценки затрат и стоимости проекта для всех трех подходов (*LOC*-оценки, *FP*-указатели и модель *COCOMO II*) описаны в третьей главе.

Как известно, любая разработка ведется в условиях риска. Очень важно не позволить факторам риска разрушить программный проект. Чтобы это не произошло, надо выявить степень опасности и защититься от нее. В качестве примера выполним начальную оценку факторов риска для нашего проекта (табл. 13.8).

Таблица 13.8. Начальная оценка факторов риска

Фактор риска	Вероятность, %	Потери	Влияние риска
Нестандартное аппаратное обеспечение заказчика	12	9	108
Удорожание стоимости проекта из-за ликвидации риска	10	10	100
Срыв плана из-за недостатка опыта в использовании новых технологий	15–20	5	75–100
Срыв плана из-за несоответствия документации на стандартные компоненты независимых разработчиков	14–17	5	70–85
Дальнейшая невостребованность ПО на рынке	5	8	40
Сбои в работе по вине аппаратуры	2	3	6

Мероприятия по защите от факторов риска представим в виде плана (таблица 13.9).

Таблица 13.9. План управления факторами риска

Фактор риска	Мероприятия
Нестандартное аппаратное обеспечение заказчика	ПО проектируется для стандартных моделей торговых автоматов. Если потребуется поддержка автоматов со специфичной архитектурой, возможно привлечение дополнительных ресурсов для поддержки нескольких версий
Удорожание стоимости проекта из-за ликвидации риска	При возникновении ситуации превышения бюджета возможно открытие кредитной линии в банке для погашения затрат. Впрочем, при превышении допустимых пределов затрат можно прекратить реализацию проекта
Срыв плана из-за недостатка опыта в использовании новых технологий	Возможно привлечение к разработке персонала, имеющего опыт в подобного рода технологиях
Срыв плана из-за несоответствия документации на стандартные компоненты независимых разработчиков	Для сокращения влияния данного фактора риска возможна организация поиска информации в различных конференциях. Возможно обращение к другим разработчикам, нашедшим решения для подобных проблем. Кроме того, возможна замена неудачного компонента на эквивалентные компоненты
Дальнейшая невостребованность ПО на рынке	Нужно постоянно контролировать спрос на рынке. В случае необходимости провести дополнительную рекламную кампанию. При особо заниженном спросе на ПО необходимо как можно скорее обнаружить этот факт, чтобы сократить потери от бессмысленного продолжения проекта
Сбои в работе по вине аппаратуры	Ввести электронный журнал для всех аппаратных событий, происходящих в системе

Этап РАЗВИТИЕ

На этом этапе уточняется состав элементов Use Case начальной диаграммы, создаются сценарии для элементов Use Case, разрабатываются диаграммы последовательности (формализующие текстовые представления сценариев), проектируются диаграммы классов и планируется содержание следующего этапа разработки.

Детализация содержания элемента Use Case Maintain приводит к необходимости выделения из него следующих элементов Use Case:

- Fix — описывает действия, выполняемые в случае ремонта автомата. Инициируется актером Техник (Technican).
- ChangePrice — изменить цену — инициируется актером Техник (Technican) и описывает действия, выполняемые при необходимости изменить цену на какой-либо продукт или продукты.
- TakeMoney — забрать деньги — инициируется актером Техник (Technican) и описывает действия, выполняемые в случае, когда Техник производит инкассацию автомата.
- InsertProduct — добавить продукт — инициируется актером Техник (Technican) и описывает действия, выполняемые при необходимости добавления продуктов в автомат.

Модифицированная диаграмма Use Case представлена на рис. 13.21.

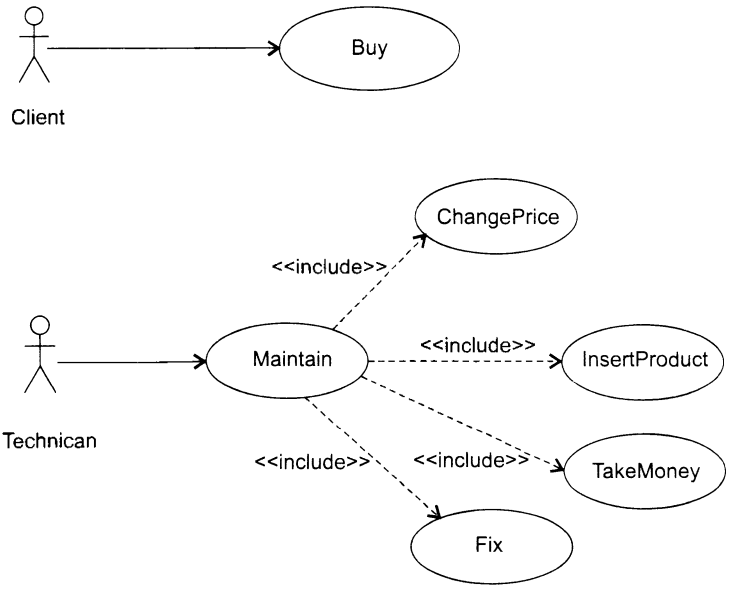


Рис. 13.21. Модифицированная диаграмма Use Case

Перед созданием сценариев для детального описания элементов Use Case целесообразно определиться с функциональной структурой будущего автомата

Без ограничения общности можно выделить в будущем автомате следующие части:

- MainInterface — обеспечивает взаимодействие со всеми остальными частями в системе.
- MoneyMachine — отвечает за операции, выполняемые с наличными деньгами.
- AutorizationUnit — обеспечивает операции авторизации техника в системе. Поддерживает интерфейс магнитных ключей.
- Extractor — отвечает за обеспечение приема и выдачи продуктов.
- Display — обеспечивает вывод информации на экран.
- ProductTable — содержит информацию о количестве и ценах продуктов.

Сценарии элемента Use Case Buy

Основной поток действий

Клиент вносит в отсек MoneyMachine автомата количество денег, необходимое для покупки продукта. В результате посылки сообщения из модуля MoneyMachine в модуль Display на дисплее отображается количество внесенных денег.
Начинается сессия работы с клиентом, о которой модуль MoneyMachine оповещает модуль MainInterface. Клиент выбирает продукт в меню модуля MainInterface. На дисплее отображается название выбранного продукта (модуль MainInterface посылает в модуль Display соответствующее сообщение).
MainInterface проверяет наличие необходимой суммы внесенных денег, и, если проверка оказалась успешной, производится выдача выбранного продукта (путем посылки определенной команды в модуль Extractor). Модуль MainInterface обновляет состояние модуля ProductTable (сведения о количестве продуктов), а также собственное состояние.
После проделанных операций дисплей возвращается в начальное состояние, то есть отображается приветствие (по сообщению, посылаемому модулем MainInterface в модуль Display).

Альтернативный поток 1

Если проверка наличия необходимой суммы внесенных денег оказалась неуспешной, клиенту выдается на экране сообщение, где предлагается либо забрать деньги, либо выбрать другой продукт. После сеанса работы экран возвращается в начальное состояние, то есть показывается приветствие (по сообщению, посылаемому модулем MainInterface в модуль Display).

Альтернативный поток 2

Если были внесены деньги в автомат, но клиент решает отказаться от покупки, то при нажатии соответствующей кнопки деньги возвращаются клиенту, а система возвращается в исходное состояние – на экране отображается приветствие (по сообщению, посылаемому модулем MainInterface в модуль Display).

Сценарий элемента Use Case TakeMoney

Данная последовательность действий обеспечивает проведение инкассации автомата. Действия начинаются с того, что техник подносит авторизационный ключ к гнезду, подключенному к модулю MainInterface. Модуль MainInterface предлагает модулю AuthorizationUnit провести авторизацию ключа. В случае успешной авторизации модуль MainInterface посылает в модуль Display соответствующее сообщение: на дисплее появляется приветствие, у техника запрашивается код требуемой операции.

После ввода кода операции для снятия денег модуль MainInterface дает команду модулю MoneyMachine на выдачу денежной суммы. Для снятия денег технику необходимо ввести отрицательное значение снимаемой суммы.
Счетчик наличных денег модуля MainInterface устанавливается в начальное значение. Для обеспечения контроля и безопасности в системный журнал модуля MainInterface заносится соответствующая запись. После проделанных операций экран возвращается в начальное состояние, то есть отображается приветствие (по сообщению, посылаемому модулем MainInterface в модуль Display).

Сценарий элемента Use Case InsertProduct

Данная последовательность действий производится для добавления в автомат новых (или отсутствующих) продуктов. Действия начинаются с того, что техник подносит авторизационный ключ к гнезду, подключенному к модулю MainInterface. Модуль MainInterface запрашивает у модуля AuthorizationUnit проведение авторизации ключа. В случае успешной авторизации модуль MainInterface посылает в модуль Display соответствующее сообщение: на дисплее появляется приветствие, и у техника запрашивается код требуемой операции. После ввода кода операции для добавления продуктов модуль MainInterface посылает сообщения: модулю Extractor для приема продуктов и модулю ProductTable для их учета. Модуль MainInterface обновляет сведения о продуктах (в его системный журнал заносится соответствующая запись). После проделанных операций экран возвращается в начальное состояние, то есть отображается приветствие (по сообщению, посылаемому модулем MainInterface в модуль Display).

Сценарий элемента Use Case ChangePrice

Данная последовательность действий производится для изменения цены какого-либо продукта. Действия начинаются с того, что техник подносит авторизационный ключ к гнезду, подключенному к модулю MainInterface. Модуль MainInterface запрашивает у модуля AuthorizationUnit проведение авторизации ключа. В случае успешной авторизации модуль MainInterface посылает в модуль Display соответствующее сообщение: на дисплее появляется приветствие, и у техника запрашивается код требуемой операции. После ввода кода операции для изменения цены продукта модуль MainInterface посылает сообщение модулю ProductTable, и цена товара изменяется. Модуль MainInterface обновляет сведения о продуктах (в его системный журнал заносится соответствующая запись). После проделанных операций экран возвращается в начальное состояние, то есть отображается приветствие (по сообщению, посылаемому модулем MainInterface в модуль Display).

На следующем шаге сценарии элементов Use Case преобразуются в диаграммы последовательности — за счет этого достигается формализация описаний, требуемая для построения диаграммы классов. Для построения диаграмм последовательности проводится грамматический разбор каждого сценария элемента Use Case: значащие существительные превращаются в обобщенные объекты, а значащие глаголы — в сообщения, пересылаемые между объектами.

Диаграммы последовательности

Диаграммы изображены на рис. 13.22–13.25.

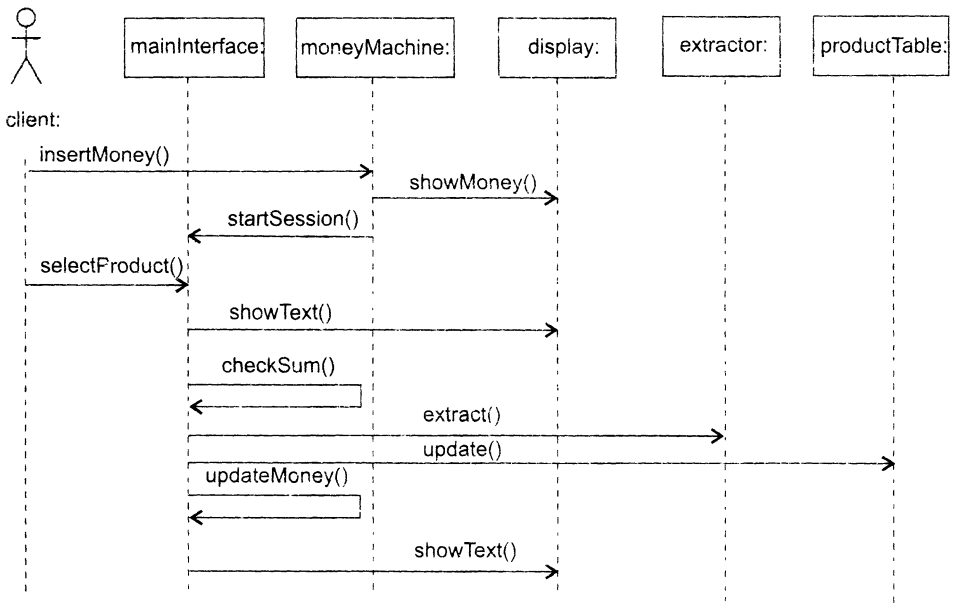


Рис. 13.22. Диаграмма последовательности для элемента Use Case Buy

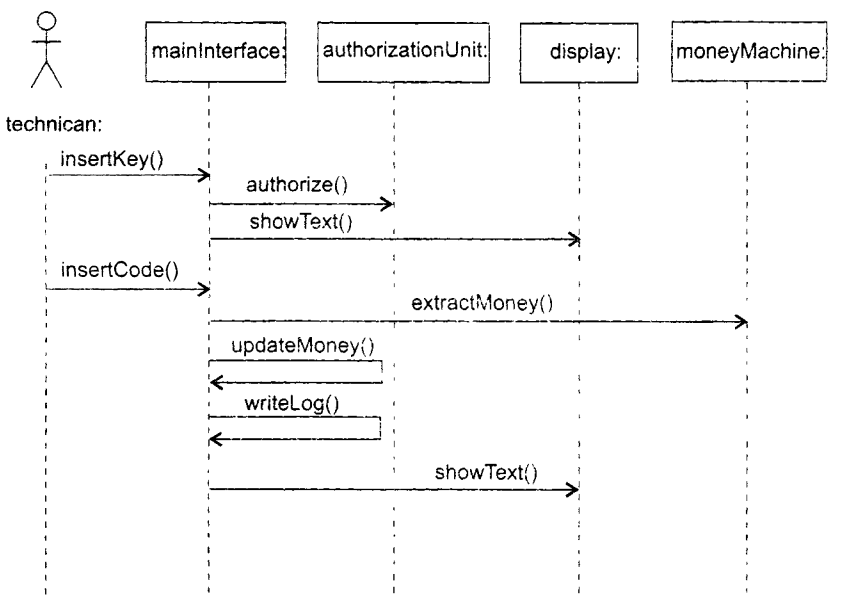


Рис. 13.23. Диаграмма последовательности для элемента Use Case TakeMoney

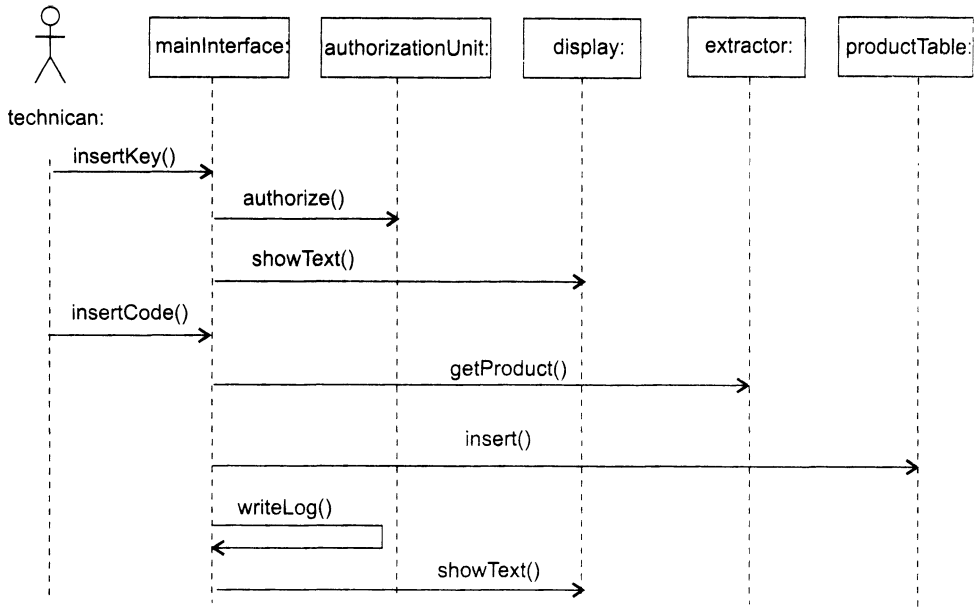


Рис. 13.24. Диаграмма последовательности для элемента Use Case InsertProduct

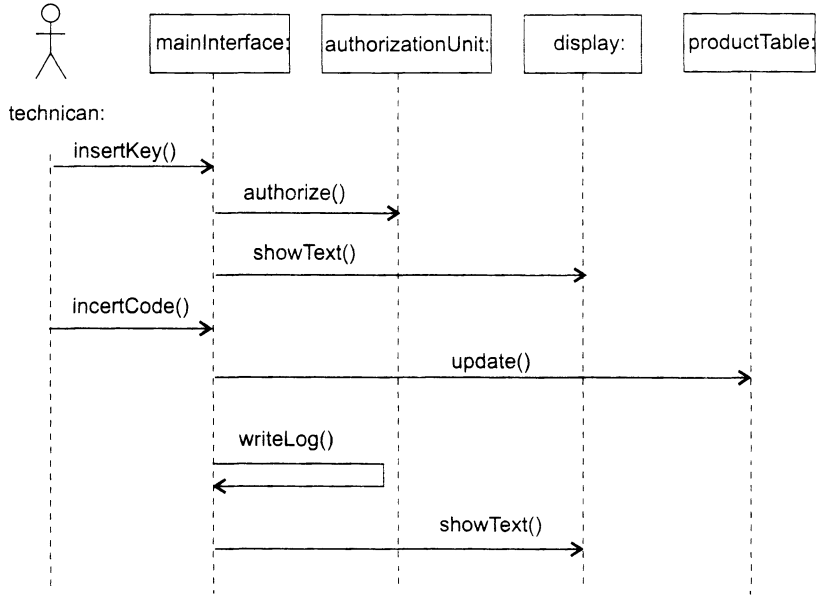


Рис. 13.25. Диаграмма последовательности для элемента Use Case ChangePrice

Создание классов

Работа по созданию классов требует изучения содержания всех диаграмм последовательности. Проводится она в три этапа.

На *первом этапе* выявляются и именуются классы. Для этого просматривается каждая диаграмма последовательности. Любой объект в этой диаграмме должен принадлежать конкретному классу, для которого надо придумать имя. Например, обобщенному объекту mainInterface должен соответствовать класс MainInterface, поэтому класс MainInterface следует ввести в диаграмму. Конечно, если в другой диаграмме последовательности опять появится подобный объект, то дополнительный класс не образуется.

На *втором этапе* выявляются операции классов. На диаграмме последовательности такая операция соответствует стрелке (и имени) сообщения, указывающей на линию жизни объекта класса. Например, если к линии жизни обобщенного объекта mainInterface подходит стрелка сообщения selectProduct(), то в класс MainInterface нужно ввести операцию selectProduct().

На *третьем этапе* определяются отношения ассоциации между классами — они обеспечивают пересылки сообщений между соответствующими обобщенными объектами.

В нашем примере анализ диаграмм последовательности позволяет выделить следующие классы: MainInterface, MoneyMachine, AutorizationUnit, Extractor, Display, ProductTable.

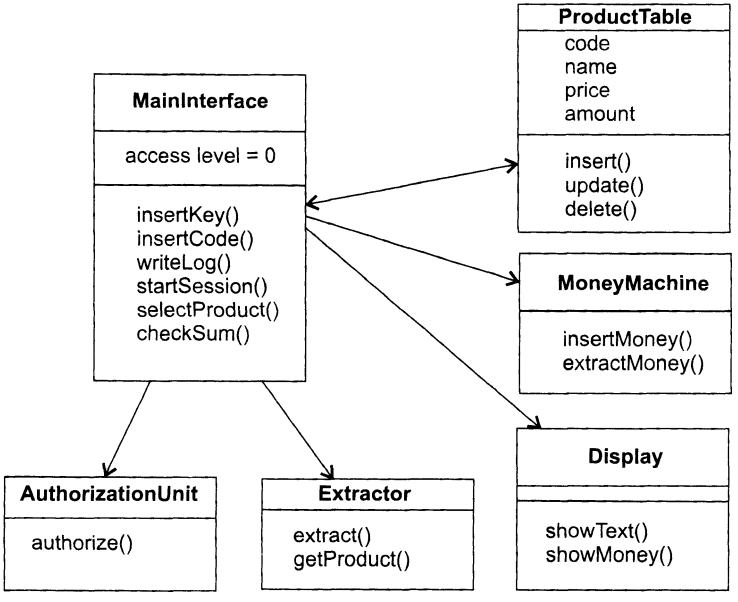


Рис. 13.26. Начальная диаграмма классов

Для реализации функций, определенных в сценариях, в классы добавляются атрибуты и операции. По результатам формирования атрибутов и операций классов обновляется содержание диаграмм последовательности.

Начальное представление диаграммы классов нашей системы показано на рис. 13.26.

Оценка качества логической структуры визуальной модели

Основным элементом логического представления визуальной модели является структура классов. Для оценки качества структуры классов удобно использовать метрики Чидамбера—Кемерера.

Результаты начальной оценки качества сведены в табл. 13.10. Методика и примеры расчета метрик приведены в главе 12.

Таблица 13.10. Результаты начальной оценки качества

Метрика	Main Interface	Money Machine	Autorization Unit	Extractor	Display	Product Table	Среднее значение
WMC	7	2	1	2	2	3	2,83
DIT	0	0	0	0	0	0	0
NOC	0	0	0	0	0	0	0
CBO	5	0	0	0	0	0	0,83
RFC	11	2	1	2	2	3	3,5
LCOM	2	0	0	1	0	0	0,5

Что можно сказать об этих оценках? Средние значения удовлетворительны. Правда, как и следовало ожидать, из общего ряда выделяются неблагоприятные оценки сложности, связности и сцепления класса MainInterface. По сути, этот класс является источником повышенной опасности для качества. Менеджерам разработчиков следует крепко подумать о его возможной модификации.

Планирование итераций конструирования

На данном шаге составляется план итераций, который определяет порядок действий на следующем этапе — этапе конструирования. Цель каждой итерации — уменьшить риск разработки конечного продукта. Для создания начального плана анализируются элементы Use Case, их сценарии и диаграммы последовательности. Устанавливается приоритет их реализации. При завершении каждой итерации будет повторно вычисляться риск. Оценка риска может привести к необходимости обновления плана итераций.

Положим, что максимальный риск связан с реализацией элемента Use Case Buy. Будем считать, что сценарии обслуживания менее опасны. Причем среди них наи-

более опасна разработка сценария InsertProduct, среднюю опасность несет сценарий TakeMoney и малую опасность — ChangePrice и Fix.

В связи с этими соображениями начальный план итераций принимает вид:

- Итерация 1 — реализация элемента Use Case Buy.
- Итерация 2 — реализация сценариев обслуживания:
  - элемента Use Case InsertProduct.
  - элемента Use Case TakeMoney.
  - элемента Use Case ChangePrice.
  - элемента Use Case Fix.

Этап КОНСТРУИРОВАНИЕ

Не будем останавливаться на подробностях содержания 1-й и 2-й итераций этапа конструирования. Отметим только, что на этих итерациях производится программирование операций классов (создаются их тела), которые задействованы при реализации соответствующих элементов Use Case (указанных в начальном плане итераций).

Кроме того, напомним, что программирование операций выполняется в среде программирования (для нашего примера это Borland Delphi Studio). Перед переходом в среду программирования классы системы упаковываются в компоненты — элементы компонентной диаграммы.

Будем считать, что программирование необходимых операций завершено. И тут появляются дополнительные требования:

- обеспечить возможность расчета клиента не только наличными деньгами, но и банковской карточкой;
- обеспечить возможность получения клиентом сдачи, если внесенная им сумма денег превышает необходимую сумму.

Для реализации этих требований вводится третья итерация конструирования.

Итерация 3

Шаг 1. Модификация диаграммы Use Case системы. Очевидно, что первое дополнительное требование приводит к появлению дополнительных элементов Use Case ByCash (покупка продукта за наличные деньги) и ByCard (покупка продукта с помощью кредитной карточки), которые находятся в отношении включения с базовым элементом Use Case Buy. Второе дополнительное требование приводит к появлению дополнительного элемента Use Case Change (выдача сдачи), который находится в отношении расширения с элементом Use Case ByCash. Этот элемент описывает действия, непосредственно связанные с подсчетом и выдачей сдачи, и задействуется только при необходимости.

Диаграмма Use Case принимает вид, представленный на рис. 13.27.

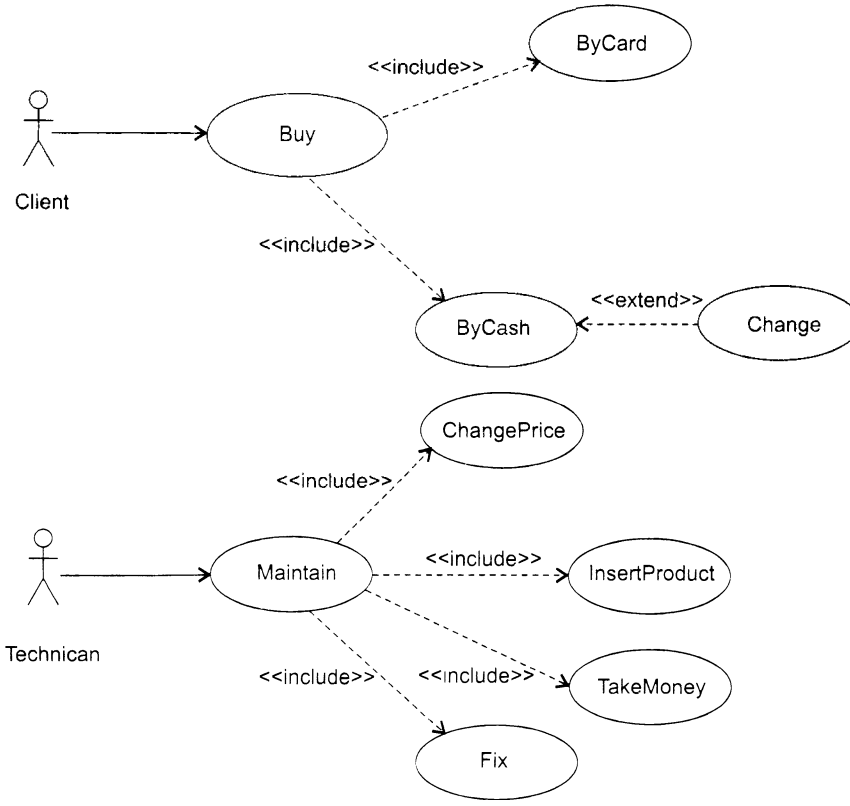


Рис. 13.27. Диаграмма Use Case с учетом дополнительной функциональности

Шаг 2. Описание дополнительных элементов Use Case. В качестве примера приведем описания двух новых элементов Use Case.

Сценарии элемента Use Case ByCash

Основной поток действий

Клиент вносит в отсек MoneyMachine автомата количество денег, необходимое для покупки продукта. В результате посылки сообщения из модуля MoneyMachine в модуль Display на дисплее отображается количество внесенных денег. Начинается сессия работы с клиентом, о которой модуль MoneyMachine оповещает модуль MainInterface. Клиент выбирает продукт в меню модуля MainInterface. На дисплее отображается название выбранного продукта (модуль MainInterface посылает в модуль Display соответствующее сообщение). MainInterface проверяет наличие достаточной суммы внесенных денег, и, если проверка оказалась успешной, производится выдача выбранного продукта (путем посылки определенной команды в модуль Extractor).
При необходимости клиенту выдается сдача (этот сценарий описывается расширяющим элементом Use Case Change). Модуль MainInterface обновляет состояние модуля ProductTable (сведения о количестве продуктов), а также собственное состояние.

После проделанных операций дисплей возвращается в начальное состояние, то есть отображается приветствие (по сообщению, посылаемому модулем MainInterface в модуль Display).

Альтернативный поток 1

Если проверка наличия достаточной суммы внесенных денег оказалась неуспешной, клиенту выдается на экране сообщение, где предлагается либо забрать деньги, либо выбрать другой продукт, либо выбрать другой способ оплаты (кредитной карточкой). После сеанса работы экран возвращается в начальное состояние, то есть показывается приветствие (по сообщению, посылаемому модулем MainInterface в модуль Display).

Альтернативный поток 2

Если были внесены деньги в автомат, но клиент решает отказаться от покупки, то при нажатии соответствующей кнопки на автомате деньги возвращаются клиенту, а система возвращается в исходное состояние – на экране отображается приветствие (по сообщению, посылаемому модулем MainInterface в модуль Display).

Сценарии элемента USE CASE ByCard

Основной поток действий

Клиент помещает кредитную карточку в соответствующий отсек MoneyMachine автомата. Модуль MoneyMachine генерирует запрос на оформление оплаты по кредитной карточке и отправляет его внешней службе авторизации кредитов. На дисплее отображается соответствующее сообщение о соединении. В случае успешного соединения начинается сессия работы с клиентом, о которой модуль MoneyMachine оповещает модуль MainInterface. Клиент выбирает продукт в меню модуля MainInterface. На дисплее отображается название выбранного продукта (модуль MainInterface посылает в модуль Display соответствующее сообщение).
MainInterface проверяет наличие необходимой суммы на счету клиента, и, если проверка оказалась успешной, производится выдача выбранного продукта (путем посылки определенной команды в модуль Extractor). Модуль MainInterface обновляет состояние модуля ProductTable (сведения о количестве продуктов), а также собственное состояние. После проделанных операций дисплей возвращается в начальное состояние, то есть отображается приветствие (по сообщению, посылаемому модулем MainInterface в модуль Display).

Альтернативный поток

Если проверка наличия необходимой суммы денег на счету клиента оказалась неуспешной, клиенту выдается на экране сообщение, где предлагается либо отказаться от покупки, либо выбрать другой продукт, либо выбрать другой способ оплаты (наличными). После сеанса работы экран возвращается в начальное состояние, то есть показывается приветствие (по сообщению, посылаемому модулем MainInterface в модуль Display).

Шаг 3. Диаграмма последовательности для дополнительных элементов Use Case. Диаграммы последовательности для основных потоков действий дополнительных элементов Use Case показаны на рис. 13.28–13.29.

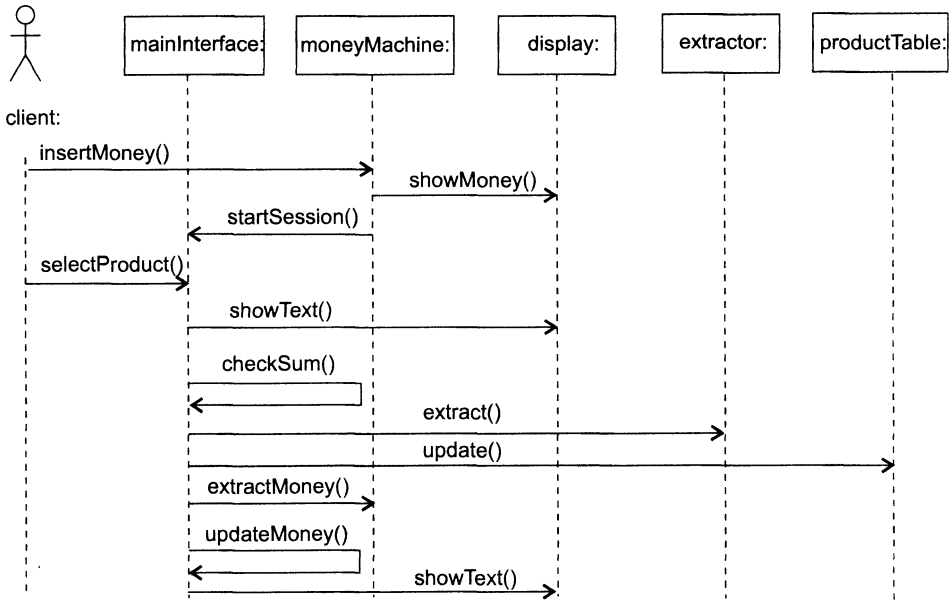


Рис. 13.28. Диаграмма последовательности для элемента Use Case ByCash

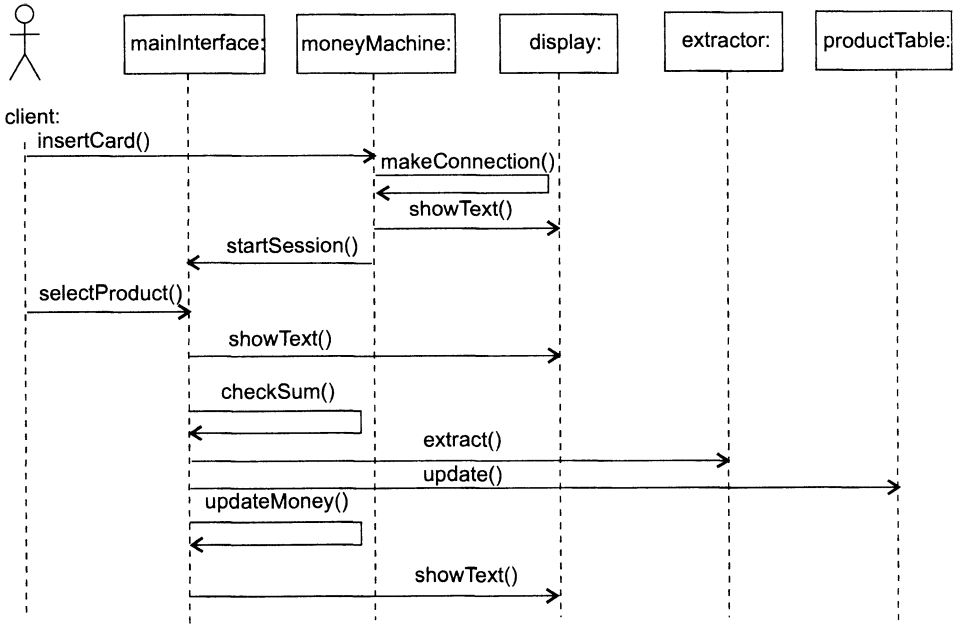


Рис. 13.29. Диаграмма последовательности для элемента Use Case ByCard

Шаг 4. Изменение диаграммы классов системы. Как и следовало ожидать, наращивание функциональности автомата отразилось на диаграмме классов.

В результате анализа новых диаграмм последовательности в класс MoneyMachine добавлены две новые операции insertCard() и makeConnection(), обеспечивающие оплату клиента с помощью кредитной карточки.

Измененная на данной итерации диаграмма классов показана на рис. 13.30.

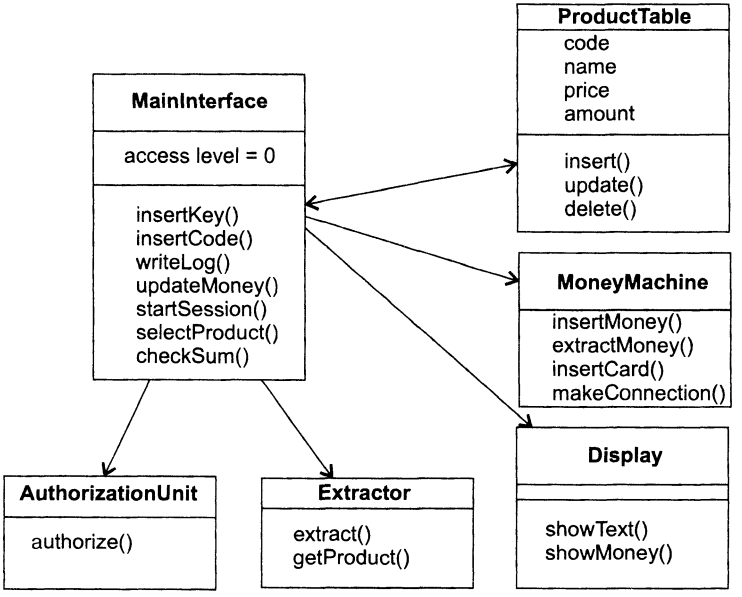


Рис. 13.30. Модифицированная диаграмма классов

Результаты оценки качества этой диаграммы сведены в табл. 13.11.

Таблица 13.11. Оценка качества системы на 3-й итерации конструирования

Метрика	Main Interface	Money Machine	Autorization Unit	Extractor	Display	Product Table	Среднее значение
WMC	7	4	1	2	2	3	3,17
DIT	0	0	0	0	0	0	0
NOC	0	0	0	0	0	0	0
CBO	5	0	0	0	0	0	0,83
RFC	11	4	1	2	2	3	3,83
LCOM	2	4	0	1	0	0	1,17

Сравним оценки начального качества и качества третьей итерации.

Рост средних значений метрик *WMC*, *RFC*, и *LCOM* — свидетельство возрастания сложности продукта.

По-прежнему самые высокие значения сложности и сцепления имеет класс MainInterface — он остается главным источником опасности.

Существенно ухудшилась связность внутри класса MoneyMachine. Причина — введение в него двух новых операций, которые не связаны по данным с предыдущими операциями класса.

Вывод: качество разработки понизилось. Это стало следствием неправильного размещения новых операций. Очевидно, что для улучшения ситуации необходимо перемещение операций insertCard() и makeConnection() в новый класс Card.

Шаг 5. Продолжение программирования операций для классов системы. Здесь программируются реализации операций insertCard() и makeConnection() в новом классе Card. Кроме того, добавляются вспомогательные и служебные операции в другие классы. Такие операции учитывают специфику среды программирования, библиотек этой среды.

Еще один вид деятельности связан с оптимизацией топологии диаграммы классов. Например, в классах ищутся общие характеристики (атрибуты и операции). Если они найдутся, то их выделяют в дополнительный суперкласс, который связывается с родственными классами отношениями наследования. Исследуется также возможность подчиненности классов, которая приводит к появлению отношений агрегации.

По результатам всей этой работы обновляется содержание диаграмм последовательности, а также диаграммы классов, пересчитываются оценки качества. Работа завершается при достижении приемлемого уровня качества и реализации всех требований к системе.

Заметим, что по мере продвижения вперед могут появляться новые требования, для реализации которых планируются новые итерации, приводящие к эволюционному развитию функциональности системы.

Возможен и альтернативный вариант, соответствующий инкрементному стилю разработки для зафиксированного набора требований. Тогда начальная порция требований реализуется в ходе i-й итерации, средняя порция — в ходе $(i + 1)$-й итерации, а конечная — в ходе $(i + 2)$-й итерации.

Разработка в стиле экстремального программирования

Базовые понятия и методы XP-процесса разработки обсуждались в разделе «XP-процесс» главы 1. Напомним, что основная область применения XP — небольшие проекты с постоянно изменяющимися требованиями заказчика [25, 26, 27, 28, 101]. Заказчик может не иметь точного представления о том, что должно быть сделано. Функциональность разрабатываемого продукта может изменяться каждые несколько месяцев. Именно в этих случаях XP позволяет достичь максимального успеха.

Основным структурным элементом XP-процесса является XP-реализация. Рассмотрим ее организацию.

XP-реализация

Структура XP-реализации показана на рис. 13.31.

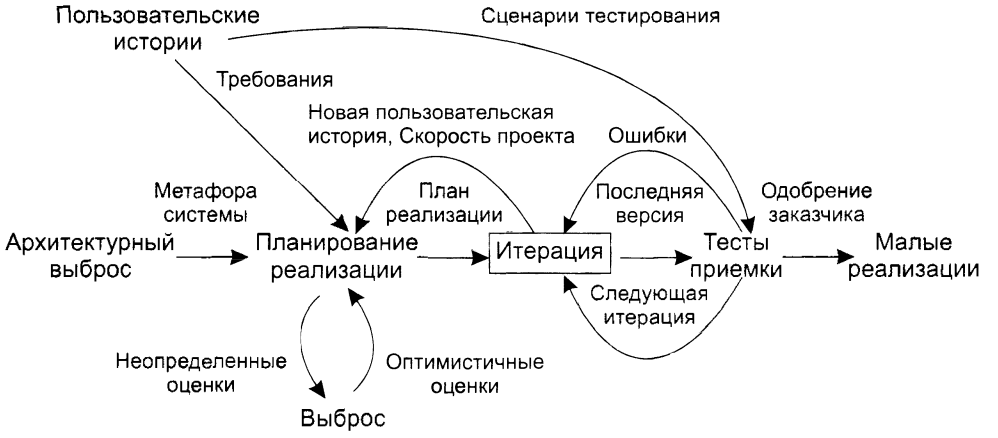


Рис. 13.31. Структура XP-реализации

Исходные требования к продукту фиксируются с помощью пользовательских историй. Истории позволяют оценить время, необходимое для разработки продукта. Они записываются заказчиком и задают действия, которые должна выполнять для него программная система. Каждая история занимает три-четыре текстовых предложения в терминах заказчика. Кроме того, истории служат для создания тестов приемки. Тесты приемки используют для проверки правильности реализации пользовательских историй.

Очень часто истории путают с традиционными требованиями к системе. Самое важное отличие состоит в уровне детализации. Истории обеспечивают только такие детали, которые необходимы для оценки времени их реализации (с минимальным риском). Когда наступит время реализовать историю, разработчики получат у заказчика более детальное описание требований.

Каждая история получает оценку идеальной длительности — одну, две или три недели разработки (время, за которое история может быть реализована при отсутствии других работ). Если срок превышает три недели, историю следует разбить на несколько частей. При длительности менее недели нужно объединить несколько историй.

Другое отличие истории от требований — во главу угла ставятся желания заказчика. В историях следует избегать описания технических подробностей, таких как организация баз данных или описания алгоритмов.

Если разработчики не знают, как оценить историю, они создают выброс — быстрое решение, содержащее ответ на трудные вопросы. Выброс — минимальное решение, выполняемое в черновом коде и впоследствии выбрасываемое. Результат выброса — знание, достаточное для оценивания.

Итогом архитектурного выброса является создание метафоры системы. Метафора системы определяет ее состав и именование элементов. Она позволяет удержать команду разработчиков в одних и тех же рамках при именовании классов,

методов, объектов. Это очень важно для понимания общего проекта системы и повторного использования кодов. Если разработчик правильно предугадывает, как может быть назван объект, это приводит к экономии времени. Следует применять такие правила именования объектов, которые каждый сможет понять без специальных знаний о системе.

Следующий шаг — планирование реализации. Планирование устанавливает правила того, каким образом вовлеченные в проект стороны (заказчики, разработчики и менеджеры) принимают соответствующие решения. Главным итогом является план выпуска версий, охватывающий всю реализацию. Далее этот план используется для создания планов каждой итерации. Итерации детально планируются непосредственно перед началом каждой из них. Важнейшая задача — правильно оценить сроки выполнения работ по каждой из пользовательских историй.

Часто при составлении плана заказчик пытается сократить сроки. Этого делать не стоит, чтобы не вызвать впоследствии проблем. Основная философия планирования строится на том, что имеются четыре параметра измерения проекта — объем, ресурсы, время и качество. Объем — сколько работы должно быть сделано, ресурсы — как много используется разработчиков, время — когда проект будет закончен, качество — насколько хорошо будет реализована и протестирована система. Можно, конечно, установить только три из четырех параметров, но равновесие всегда будет достигаться за счет оставшегося параметра.

План реализации определяет даты выпуска версий и пользовательские истории, которые будут воплощены в каждой из них. Исходя из этого, можно выбрать истории для очередной итерации. В течение итерации создаются тесты приемки, которые выполняются в пределах этой итерации и всех последующих, чтобы обеспечить правильную работу системы. План может быть пересмотрен в случае значительного отставания или опережения по итогам одной из итераций.

В каждую XP-реализацию многократно вкладывается базовый элемент — XP-итерация. Рассмотрим организацию XP-итерации.

XP-итерация

Структура XP-итерации показана на рис. 13.32.

Организация итерации придает XP-процессу динамизм, требуемый для обеспечения готовности разработчиков к постоянным изменениям в проекте. Не следует выполнять долгосрочное планирование. Вместо этого выполняется краткосрочное планирование в начале каждой итерации. Не стоит пытаться работать с незапланированными задачами — до них дойдет очередь в соответствии с планом реализации.

Привыкнув не добавлять функциональность заранее и использовать краткосрочное планирование, разработчики смогут легко приспосабливаться к изменению требований заказчика.

Цель планирования, с которого начинается итерация, — выработать план решения программных задач. Каждая итерация должна длиться от одной до трех недель. Пользовательские истории внутри итерации сортируются в порядке их значимости для заказчика. Кроме того, добавляются задачи, которые не смогли пройти предыдущие тесты приемки и требуют доработки.

Пользовательским историям и тестам с отказом в приемке сопоставляются задачи программирования. Задачи записываются на карточках, совокупность карточек образует детальный план итерации.

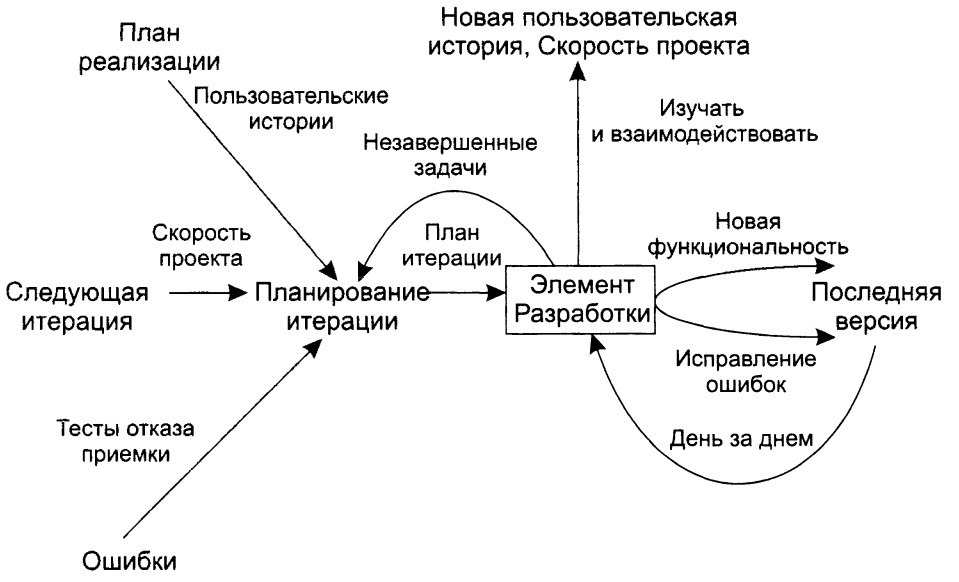


Рис. 13.32. Структура XP-итерации

Для решения каждой из задач требуется от одного до трех дней. Задачи, для которых нужно менее одного дня, группируются вместе, а большие задачи разбивают на более мелкие задачи.

Разработчики оценивают количество и длительность задач. Для определения фиксированной длительности итерации используют метрику «скорость проекта», вычисленную по предыдущей итерации (количество завершенных в ней задач/дней программирования). Если предполагаемая скорость превышает предыдущую скорость, заказчик выбирает истории, которые следует отложить до более поздней итерации. Если итерация слишком мала, к разработке принимается дополнительная история. Вполне допустимая практика — переоценка историй и пересмотр плана реализации после каждых трех или пяти итераций. Первоочередная реализация наиболее важных историй — гарантия того, что для клиента делается максимум возможного. Стиль разработки, основанный на последовательности итераций, улучшает подвижность процесса разработки.

В каждую XP-итерацию многократно вкладывается строительный элемент — элемент XP-разработки. Рассмотрим организацию элемента XP-разработки.

Элемент XP-разработки

Структура элемента XP-разработки показана на рис. 13.33.

День XP-разработчика начинается с установочной встречи. Ее цели: обсуждение проблем, нахождение решений и определение точки приложения усилий всей команды.

Участники утренней встречи стоят и располагаются по кругу, так можно избежать длинных дискуссий. Все остальные встречи проходят на рабочих местах, за компьютером, где можно просматривать код и обсуждать новые идеи.

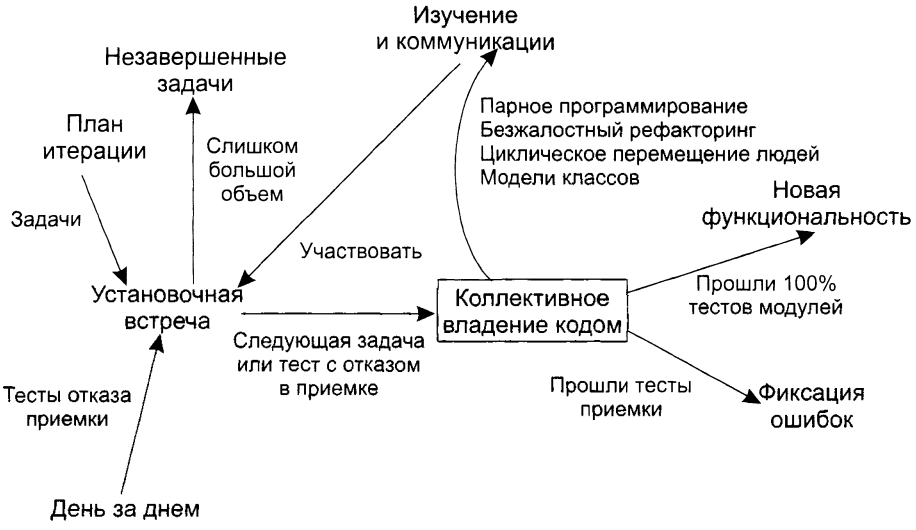


Рис. 13.33. Структура элемента XP-разработки

Весь день XP-разработчика проходит под лозунгом «коллективного владения кодом» программной системы. В результате этого происходит фиксация ошибок и добавление новой функциональности в систему.

Следует удерживаться от соблазна добавлять в продукт функциональность, которая будет востребована позже. Полагают, что только 10% такой функциональности будет когда-либо использовано, а потери составят 90% времени разработчика. В XP считают, что дополнительная функциональность только замедляет разработку и исчерпывает ресурсы. Предлагается подавлять такие творческие порывы и концентрироваться на текущих, запланированных задачах.

Коллективное владение кодом

Организацию коллективного владения кодом иллюстрирует рис. 13.34.

Коллективное владение кодом позволяет каждому разработчику выдвигать новые идеи в любой части проекта, изменять любую строку программы, добавлять функциональность, фиксировать ошибку и проводить рефакторинг. Один человек просто не в состоянии удержать в голове проект нетривиальной системы. Благодаря коллективному владению кодом снижается риск принятия неверного решения (главным разработчиком) и устраняется нежелательная зависимость проекта от одного человека.

Работа начинается с создания тестов модуля, она должна предшествовать программированию модуля. Тесты необходимо помещать в библиотеку кодов вместе с кодом, который они тестируют. Тесты делают возможным коллективное создание кода и защищают код от неожиданных изменений. В случае обнаружения ошибки также создается тест, чтобы предотвратить ее повторное появление.

Кроме тестов модулей создаются тесты приемки, они основываются на пользовательских историях. Эти тесты испытывают систему как «черный ящик» и ориентированы на требуемое поведение системы.

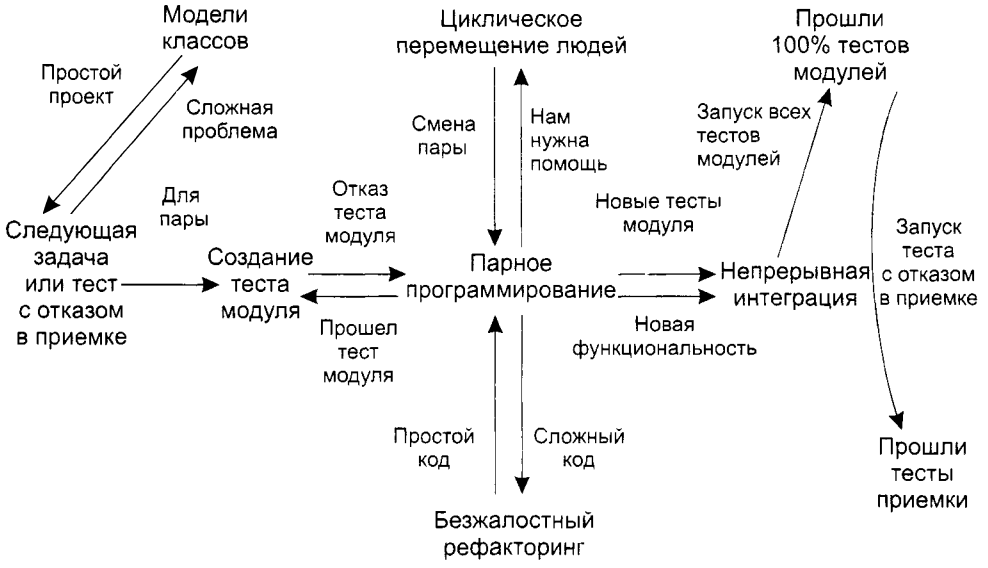


Рис. 13.34. Организация коллективного владения кодом

На основе результатов тестирования разработчики включают в очередную итерацию работу над ошибками. Вообще, следует помнить, что тестирование — один из краеугольных камней XP.

Все коды в проекте создаются парами программистов, работающими за одним компьютером. Парное программирование приводит к повышению качества без дополнительных затрат времени. А это, в свою очередь, уменьшает расходы на будущее сопровождение программной системы.

Оптимальный вариант для парной работы — одновременно сидеть за компьютером, передавая друг другу клавиатуру и мышь. Пока один человек набирает текст и думает (тактически) о создаваемом методе, второй думает (стратегически) о размещении метода в классе.

Во время очередной итерации всех сотрудников перемещают на новые участки работы. Такие перемещения помогают устранить изоляцию знаний и «узкие места». Особенно полезна смена одного из разработчиков при парном программировании.

Замечено, что программисты очень консервативны. Они продолжают использовать код, который трудно сопровождать, только потому, что он все еще работает. Боязнь модификации кода у них в крови. Это приводит к предельному понижению эффективности систем. В XP считают, что код нужно постоянно обновлять — удалять лишние части, убирать ненужную функциональность. Этот процесс называют рефакторингом кода (refactoring). Поощряется безжалостный рефакторинг, сохраняющий простоту проектных решений. Рефакторинг поддерживает прозрачность и целостность кода, обеспечивает его легкое понимание, исправление и расширение. На рефакторинг уходит значительно меньше времени, чем на сопровождение устаревшего кода. Увы, нет ничего вечного — когда-то отличный модуль теперь может быть совершенно не нужен.

И еще одна составляющая коллективного владения кодом — непрерывная интеграция.

Без последовательной и частой интеграции результатов в систему разработчики не могут быть уверены в правильности своих действий. Кроме того, трудно вовремя оценить качество выполненных фрагментов проекта и внести необходимые коррекции.

По возможности XP-разработчики должны интегрировать и публично отображать, демонстрировать код каждые несколько часов. Интеграция позволяет объединить усилия отдельных пар и стимулирует повторное использование кода.

Взаимодействие с заказчиком

Одно из требований XP — постоянное участие заказчика в проведении разработки. По сути, заказчик является одним из разработчиков.

Все этапы XP требуют непосредственного присутствия заказчика в команде разработчиков. Причем разработчикам нужен заказчик-эксперт. Он создает пользовательские истории, на основе которых оценивается время и назначаются приоритеты работ. В ходе планирования реализации заказчик указывает, какие истории следует включить в план реализации. Активное участие он принимает и при планировании итерации.

Заказчик должен как можно раньше увидеть программную систему в работе. Это позволит как можно раньше испытать систему и дать отзыв о ее работе. Поскольку при укрупненном планировании заказчик остается в стороне, разработчикам нужно постоянно общаться с заказчиком, чтобы получать как можно больше сведений при реализации задач программирования. Нужен заказчик и на этапе функционального тестирования при проведении тестов приемки.

Таким образом, активное участие заказчика не только предотвращает появление некачественной системы, но и является непременным условием выполнения разработки.

Стоимость изменения и проектирование

В основе организации прогнозирующих (тяжеловесных) процессов разработки лежит утверждение об экспоненциальной кривой стоимости изменения. Согласно этой кривой, по мере развития проекта стоимость внесения изменений экспоненциально возрастает (рис. 13.35) — то, что на этапе формирования требований стоит единицу, на этапе сопровождения будет стоить тысячу.

В основе адаптивного (облегченного) XP-процесса лежит предположение, что экспоненциальную кривую можно сгладить (рис. 13.36) [45, 50, 57, 58, 86]. Такое сглаживание, с одной стороны, возникает при применении методологии XP, а с другой — оно же в ней и применяется. Это еще раз подчеркивает тесную взаимосвязь между методами XP: нельзя использовать методы, которые опираются на сглаживание, не используя другие методы, которые это сглаживание производят.

К числу основных методов, осуществляющих сглаживание, относят:

- тотальное тестирование;
- непрерывную интеграцию;
- рефакторинг.

Рис. 13.35. Экспонента стоимости изменения в прогнозирующем процессе

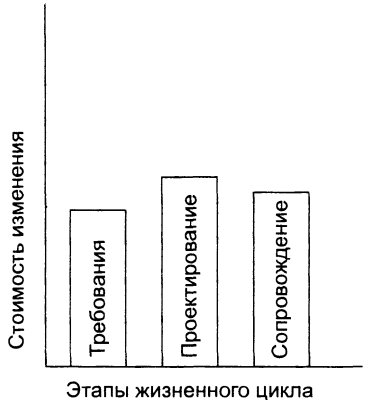


Рис. 13.36. Сглаживание стоимости изменения в адаптивном процессе

Повторим — надежность кода, которую обеспечивает сквозное тестирование, создает базис успеха, обеспечивает остальные возможности XP-разработки. Непрерывная интеграция необходима для синхронной работы всех сотрудников, чтобы любой разработчик мог вносить в систему свои изменения и не беспокоиться об интеграции с остальными членами команды. Совместные усилия этих двух методов оказывают существенное воздействие на кривую стоимости изменений в программной системе.

О влиянии рефакторинга очень интересно пишет Джим Хайсмит (Jim Highsmith) [58]. Он приводит аналогию с весами (рис. 13.37 и рис. 13.38).

На одной чаше весов лежит предварительное проектирование, на другой — рефакторинг. В прогнозирующих процессах разработки перевешивает предварительное проектирование, поскольку скорость изменений низкая (на рисунке скорость иллюстрируется положением точки равновесия). В адаптивных, облегченных процессах перевешивает рефакторинг, так как скорость изменений высокая. Это не означает отказа от предварительного проектирования. Однако теперь можно говорить о существовании баланса между двумя подходами к проектированию, из которых можно выбрать наиболее подходящий подход.

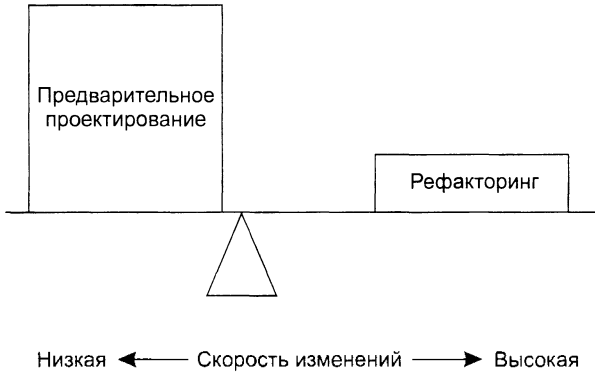


Рис. 13.37. Балансировка проектирования и рефакторинга при прогнозе

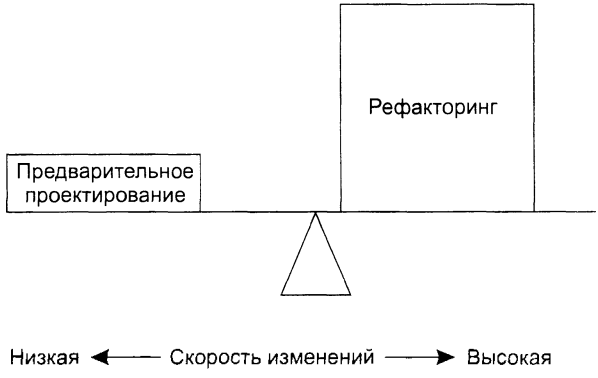


Рис. 13.38. Балансировка проектирования и рефакторинга при адаптации

Итак, в адаптивных процессах вообще и в XP-процессе в частности приветствуется простое проектирование.

Простое проектирование основывается на двух принципах:

- это вам не понадобится;
- ищите самое простое решение, которое может сработать.

Что означает первый принцип? Кажется, что все понятно — не надо сегодня писать код, который понадобится завтра. И все же сложности возникают, например, при создании гибких элементов (повторно используемых компонентов и паттернов) — ведь при этом вы смотрите в будущее, заранее добавляете к общей стоимости работ стоимость нужного проектирования и рассчитываете впоследствии вернуть эти деньги.

Тем не менее XP не советует заниматься созданием гибких элементов заранее. Лучше, если они будут добавляться по мере необходимости. Если сегодня нужен класс Арифметика, который выполняет только сложение, то сегодня я буду встраивать в этот класс именно сложение. Даже если я уверен, что уже на следующей итерации понадобится умножение и мне легко реализовать его сейчас, все равно

следует отложить эту работу до следующей итерации — когда в ней появится реальная необходимость.

Экономически такое поведение оправдано – незапланированная работа всегда крадет ресурсы у запланированной. Кроме того, отклонение от плана — это нарушение соглашений с заказчиком. К тому же появляется риск сорвать выполнение текущей работы. И даже если появилось свободное время, решение о его заполнении принимает заказчик («он сверху видит все, ты так и знай!»).

И еще одно оправдание — возможность ошибиться, ведь у нас еще нет подробных требований заказчика. А чем раньше мы введем в проект ошибочное решение, тем хуже.

Теперь о простоте решения. XP-идеолог Кент Бек приводит четыре критерия простой системы:

- система успешно проходит все тесты;
- код системы ясно раскрывает все изначальные замыслы;
- в ней отсутствует дублирование кода;
- используется минимально возможное количество классов и методов.

Успешное тестирование — довольно понятный критерий. Отсутствие дублирования кода, минимальное количество классов/методов — тоже ясные требования. А как расшифровать слова «раскрывает изначальные замыслы»?

XP всячески подчеркивает, что хороший код — это код, который можно легко прочесть и понять. Если вы хотите сделать комплимент XP-разработчику и скажете, что он пишет «умный код», будьте уверены — вы оскорбили человека.

Словом, сложную конструкцию труднее осмыслить. Понятно, что будущие модификации продукта приведут к его усложнению. Так зачем же усложнять заранее?

Такой стиль работы абсурден, если внедрять его в прогнозирующий, обычный процесс и игнорировать остальные методы XP. В комплексе с остальными XP-причудами он может стать действительно полезным.

Планирование в XP-разработке системы обслуживания банковских карт

В современных банках традиционно применяются комплексные системы, обеспечивающие широкий спектр услуг для физических и юридических лиц.

Одной из таких услуг является обслуживание банковских карт. Рассмотрим особенности планирования в среде экстремального программирования, используя фрагменты разработки подобной системы.

Спецификация заказчика

Некий банк испытывает потребность в автоматизации процесса обслуживания банковских карт.

Обслуживание банковских карт включает следующий набор услуг:

- Оформление карты:
  - Клиент заполняет заявление на открытие банковской карты, где указывает: персональные данные (имя, фамилию, адрес, контактный телефон); вид

открываемой карты (Visa, Maestro, MasterCard): данные счета, к которому привязывается карта (новый счет, существующий счет, секретный пароль); вид валюты (рубли, доллары, евро).

- Заявление клиента передается в card-отдел, где данные заносятся в журнал. Запись в журнале имеет поля: дата, номер заявления, имя, фамилия, адрес, телефон, вид карты, счет, вид валюты, статус (открытая, закрытая, заблокированная), информация о счете.

- Закрытие карты:
  - Клиент приносит заявление о закрытии банковской карточки, где указываются: имя, фамилия, номер счета, причина закрытия. Данные заявления заносятся в журнал card-отдела.
- Блокирование карты:
  - Клиент приносит заявление о блокировании карты, где указываются: имя, фамилия, номер счета, причина блокирования. Данные заявления заносятся в журнал card-отдела.
- Дополнительные операции:
  - Редактирование данных о клиентах, их счетах и картах. Отслеживание действий, совершаемых с картами, их журналирование.

Формирование пользовательских историй

Руководству фирмы-разработчика удалось убедить банк в целесообразности применения гибких технологий к разработке программной системы. В связи с требованиями XP-процесса банк выделил компетентного представителя заказчика, который включился в игру планирования и создал 11 пользовательских историй:

1. Оформление нового клиента.
2. Просмотр информации о клиенте.
3. Редактирование информации о клиенте.
4. Просмотр информации о счетах клиента.
5. Оформление нового счета.
6. Редактирование информации о счете.
7. Оформление новой карты.
8. Редактирование информации о карте.
9. Блокирование карты клиента.
10. Закрытие карты клиента.
11. Журнал card-отдела.

В свою очередь, команда разработчиков оценила длительность реализации каждой истории, использовав единицу измерения «полный день» (8 часов работы команды). Истории были записаны на отдельных бумажных карточках, сводная информация по историям представлена в табл. 13.12.

Таблица 13.12. Пользовательские истории

№	Название истории	Описание истории	Приоритет	Длительность, дней
1	Оформление нового клиента	Клиент предоставляет основную информацию о себе (имя, фамилия, адрес, контактный телефон), которая впоследствии сохраняется в БД банка	Максимальный	8
2	Просмотр информации о клиенте	Отображение основной информации о клиенте (имя, фамилия, адрес, контактный телефон). Данные о клиенте берутся из БД банка	Максимальный	7
3	Редактирование информации о клиенте	Редактирование основной информации о клиенте (имя, фамилия, адрес, контактный телефон). Данные после внесенных изменений сохраняются в БД банка	Средний	7
4	Просмотр информации о счетах клиента	Отображение информации о счетах (IBAN), оформленных на определенного клиента банка. Данные берутся из БД банка и отображаются в виде списка	Высокий	6
5	Оформление нового счета	Генерация нового IBAN и указание секретного пароля. Информация об оформляемом счете сохраняется в БД банка с привязкой к соответствующему клиенту	Максимальный	7
6	Редактирование информации о счете	Определение нового секретного пароля для указанного IBAN. Изменять IBAN для клиента нельзя. Данные после внесенных изменений сохраняются в БД банка	Средний	7
7	Оформление новой карты	Указывается информация о виде карты (Visa Classic, Visa Electron, Maestro, MasterCard), о виде валюты (RUR, EUR, USD). В случае необходимости создается новый счет. Данные после внесенных изменений сохраняются в БД банка, и статус карты помечается как «открытая»	Максимальный	8
8	Редактирование информации о карте	Редактирование информации о карте: вид карты (Visa Classic, Visa Electron, Maestro, MasterCard), вид валюты (RUR, USD, EUR), статус карты (открыта, закрыта, заблокирована). Данные после внесенных изменений сохраняются в БД банка	Максимальный	7
9	Блокирование карты клиента	Указать данные клиента (имя, фамилия, секретный пароль), номер счета и причину блокировки. Заблокировать банковскую карту. Статус карты отмечается как «заблокирована»	Высокий	6
10	Закрытие карты клиента	Указывается причина закрытия карты. После подтверждения информация сохраняется в БД банка, а статус карты помечается как «закрыта»	Средний	6
11	Журнал card-отдела	Отображение активности клиента по работе с банковскими картами: оформление, блокирование, закрытие. Информация об активности берется из БД банка и отображается в виде списка. Возможен поиск за период	Низкий	7

Планирование реализации

Анализ пользовательских историй выявил предполагаемую длительность разработки системы. Она составляет 76 дней, то есть 15 недель и один день. Длительность одной итерации определили как три недели. Таким образом, всего в проекте планируется шесть итераций.

По предложению заказчика очередность реализации историй решили выбирать в соответствии с их приоритетом: вначале обработать истории с максимальным приоритетом, затем — с высоким, после чего перейти к историям со средним приоритетом, а завершить проект проработкой истории с низким приоритетом. В итоге обсуждений получили полный план реализации системы (табл. 13.13).

Таблица 13.13. План реализации системы

Итерация	Номер истории	Название истории	Приоритет
1	1	Оформление нового клиента	Максимальный
	2	Просмотр информации о клиенте	Максимальный
2	5	Оформление нового счета	Максимальный
	7	Оформление новой карты	Максимальный
3	8	Редактирование информации о карте	Максимальный
	4	Просмотр информации о счетах клиента	Высокий
4	9	Блокирование карты клиента	Высокий
	3	Редактирование информации о клиенте	Средний
5	6	Редактирование информации о счете	Средний
	10	Закрытие карты клиента	Средний
6	11	Журнал card-отдела	Низкий

Этот план охватывает выпуск всех версий системы. Далее он применяется для создания планов каждой итерации.

Планирование итерации

Планирование выполняется в начале итерации. Основная информация берется из плана реализации. Например, основная цель третьей итерации — реализовать истории «Редактирование информации о карте» и «Просмотр информации о счетах клиента». К этому добавляются работы, незавершенные на предыдущей итерации.

Каждая история детализируется: разбивается на задачи программирования (длительностью от одного до трех дней). Типовая последовательность задач программирования для одной истории:

- Создать тест приемки истории.
- Создать тест функциональности 1.
- Запрограммировать функциональность 1.

- Протестировать функциональность 1.
- Интегрировать функциональность 1 в систему.
- Протестировать результаты интеграции.

...

- Создать тест функциональности *N*.
- Запрограммировать функциональность *N*.
- Протестировать функциональность *N*.
- Интегрировать функциональность *N* в систему.
- Протестировать результаты интеграции.
- Применить тест приемки истории.

Задачи записываются на бумажных карточках, образующих план итерации. По результатам планирования итерации могут вноситься коррективы в план реализации. Эти коррективы отражают реальное положение дел.

Scrum-процесс гибкой разработки ПО

Scrum — это еще одна методика гибкой разработки программного обеспечения, которая был задумана Джеффом Сазэрлэндом и его командой в начале девяностых прошлого века. В последние годы дальнейшее развитие Scrum-методов связывают со Швабером (Schwaber) и Бидлом (Beedle) [40].

ПРИМЕЧАНИЕ

Термин Scrum принят из-за аналогии с типовой ситуацией, возникающей в игре регби при нарушении правил и остановке игры. Игроки противоборствующих команд образуют круг, в центре которого находится мяч, после чего, яростно толкаясь, стараются сдвинуть соперников так, чтобы мяч оказался вне этого круга. Эта ситуация называется «схваткой» (scrum).

Принципы Scrum хорошо согласуются с провозглашаемой концепцией гибкости или подвижности (agile) и используются как руководство в ходе разработки, включающей следующие виды деятельности: формирование требований, анализ, проектирование, развитие и поставка. Все эти виды деятельности встраиваются в паттерн процесса по имени «спринт» (sprint). Содержание спринта привязано к решаемой проблеме, оно определяется (а часто и модифицируется) Scrum-командой. Количество спринтов, требуемых для полномасштабной разработки, может меняться в зависимости от сложности и размера конечного продукта. Общий ход Scrum-процесса иллюстрируется с помощью рис. 13.39.

Scrum делает упор на использование набора паттернов процесса [88], которые доказали свою эффективность в проектах с плотными графиками работы и изменяемыми требованиями. Каждый из этих паттернов процесса определяет ряд понятий.

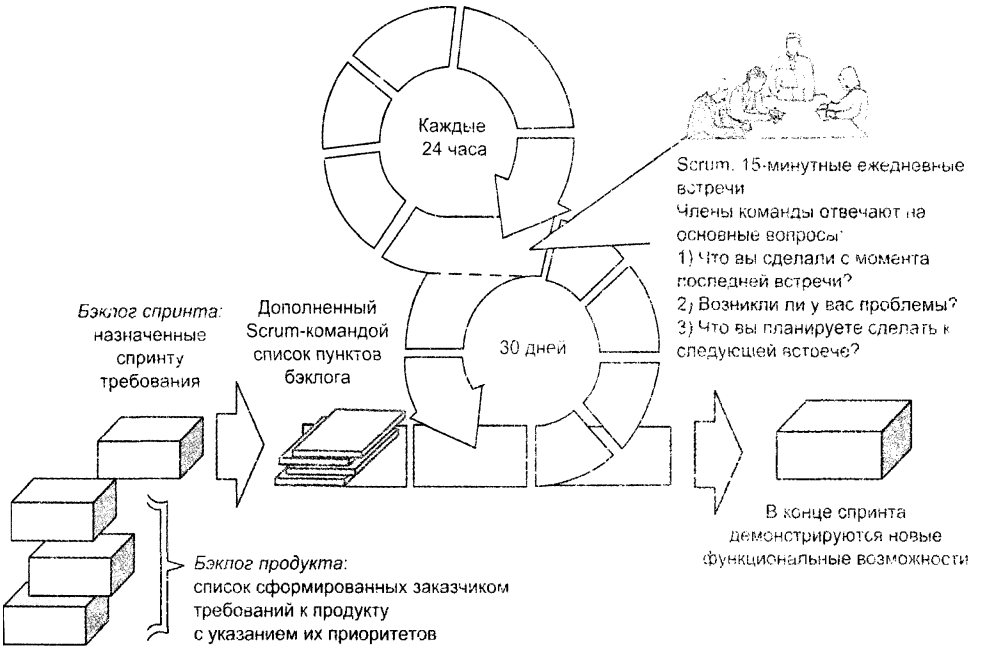


Рис. 13.39. Общий ход Scrum-процесса

1. *Бэклог* (Backlog)[1] — список требований или характеристик проекта, упорядоченных по приоритету и имеющих важное значение для заказчика. Бэклог в любое время может быть расширен (так вводятся изменения требований). Менеджер проекта оценивает бэклог и по необходимости меняет приоритеты.
2. *Спринты* — состоят из единиц работы, которые нужно выполнить для обеспечения требований бэклога, причем за предопределенный квант времени (обычно за 30 дней). Изменения в пунктах бэклога во время спринта запрещены. Следовательно, спринт позволяет членам группы работать в стабильной среде. Правда, это кратковременная стабильность.
3. *Scrum-обсуждения* (Scrum meetings) — короткие встречи (обычно 15 минут) членов Scrum-команды, проводятся ежедневно. На встрече руководитель задает всем членам команды три ключевых вопроса:
   - Что вы сделали с момента последней встречи команды?
   - С какими препятствиями вы столкнулись?
   - Чего вы планируете достичь к следующей встрече команды?

   Руководитель команды, называемый Scrum-мастером (Scrum Master), ведет встречу и оценивает ответы каждого члена группы. Scrum-обсуждение спо-

[1] Английский термин *Backlog* переводится дословно как «невыполненная работа». В последнее время стал употребляться русский перевод «журнал требований», фиксируемый в сокращенной форме, применяемой к продукту и спринту, как «журнал продукта» и «журнал спринта» соответственно.

собствует раннему обнаружению потенциальных проблем. Кроме того, эти ежедневные встречи приводят к распространению индивидуальных знаний на всю команду.

4. Демонстрационные версии – предоставляют заказчику расширенные варианты функциональной организации системы, тем самым демонстрируя реализацию новых функций. Эту реализацию заказчик может оценить. Важно отметить, что демонстрационная версия обычно не содержит всю запланированную функциональность, а включает только те функции, которые действительно были созданы в рамках прошедшего кванта времени.

Каждый спринт начинается с планирования, а заканчивается обсуждением, за которым следует мероприятие под названием «ретроспектива». Основная задача планирования состоит в выборке из бэклога всего продукта тех требований, реализации которых посвящен спринт (они помещаются в бэклог спринта). В ходе обсуждения владельцу продукта (представителю заказчика) демонстрируется созданная версия продукта, рассматриваются итоги завершенной и перспективы будущей работы. Главной темой ретроспективы является улучшение работы в следующем спринте.

Говорят, что Scrum-методика априори допускает существование хаоса и дает возможность команде успешно работать в мире, где устранение неопределенности просто невозможно. Внимательный читатель заметит несомненное сходство понятий Scrum-процесса и XP-процесса. Просто одни и те же понятия по-разному называются. И это неудивительно, ведь они «одной крови»! Хотя, если присмотреться к деталям, различия, конечно, есть.

Контрольные вопросы и упражнения

1. Что является критерием управления унифицированным процессом разработки? Как он применяется?
2. Какую структуру имеет унифицированный процесс разработки?
3. Какие этапы входят в унифицированный процесс разработки? Поясните назначение этих этапов.
4. Какие рабочие потоки имеются в унифицированном процессе разработки? Поясните назначение этих потоков.
5. Какие модели предусмотрены в унифицированном процессе разработки? Поясните назначение этих моделей.
6. Какие технические артефакты определены в унифицированном процессе разработки? Поясните назначение этих артефактов.
7. Дайте характеристику целей, действий и результатов этапа НАЧАЛО.
8. Дайте характеристику целей, действий и результатов этапа РАЗВИТИЕ.
9. Дайте характеристику целей, действий и результатов этапа КОНСТРУИРОВАНИЕ.
10. Дайте характеристику целей, действий и результатов этапа ПЕРЕХОД.

11. Какие метрики используют для оценки качества унифицированного процесса разработки?
12. Завершите описанный в главе процесс разработки системы управления торговым автоматом, а именно: детализируйте содержание пятого шага последней итерации (модифицируйте диаграмму классов с учетом класса Card и пересчитайте метрики для оценки качества системы).
13. Охарактеризуйте содержание XP-реализации.
14. В чем разница между пользовательскими историями и обычными требованиями к системе?
15. Что такое выброс?
16. Как создаются тесты приемки?
17. Поясните содержание XP-итерации.
18. В чем заключается планирование XP-итерации?
19. Что такое скорость проекта?
20. Поясните структуру элемента XP-разработки.
21. В чем заключается коллективное владение кодом? Охарактеризуйте содержание такого владения.
22. Как организуется взаимодействие с XP-заказчиком?
23. Прокомментируйте стоимость XP-изменения.
24. Поясните особенности XP-проектирования.
25. Выполните первую XP-итерацию для системы обслуживания банковских карт.
26. Сравните XP-подход со Scrum-подходом. Чем они схожи и в чем отличаются?

Глава 14

Структурное тестирование программного обеспечения

В этой главе определяются общие понятия и принципы тестирования ПО (принцип «черного ящика» и принцип «белого ящика»). Читатель знакомится с содержанием процесса тестирования, после этого его внимание концентрируется на особенностях структурного тестирования программного обеспечения (по принципу «белого ящика»), описываются его достоинства и недостатки. Далее рассматриваются наиболее популярные способы структурного тестирования: тестирование базового пути, тестирование ветвей и операторов отношений, тестирование потоков данных, тестирование циклов.

Основные понятия и принципы тестирования ПО

Тестирование — это процесс выполнения программы с целью обнаружения ошибок. Шаги процесса задаются тестами.

Каждый тест определяет:

- свой набор исходных данных и условий для запуска программы;
- набор ожидаемых результатов работы программы.

Другое название теста — тестовый вариант. Полную проверку программы гарантирует *исчерпывающее тестирование*. Оно требует проверить все наборы исходных данных, все варианты их обработки и включает большое количество тестовых вариантов. Увы, но исчерпывающее тестирование во многих случаях остается только мечтой — срабатывают ресурсные ограничения (прежде всего, ограничения по времени).

Хорошим считают тестовый вариант с высокой вероятностью обнаружения еще не раскрытой ошибки. Успешным называют тест, который обнаруживает до сих пор не раскрытую ошибку.

Целью проектирования тестовых вариантов является систематическое обнаружение различных классов ошибок при минимальных затратах времени и стоимости.

Важен ответ на вопрос — что может тестирование?

Тестирование обеспечивает:

- обнаружение ошибок;
- демонстрацию соответствия функций программы ее назначению;
- демонстрацию реализации требований к характеристикам программы;
- отображение надежности как индикатора качества программы.

А чего не может тестирование? Тестирование не может показать отсутствия дефектов (оно может показывать только присутствие дефектов). Важно помнить это (скорее печальное) утверждение при проведении тестирования.

Рассмотрим информационные потоки процесса тестирования. Они показаны на рис. 14.1.

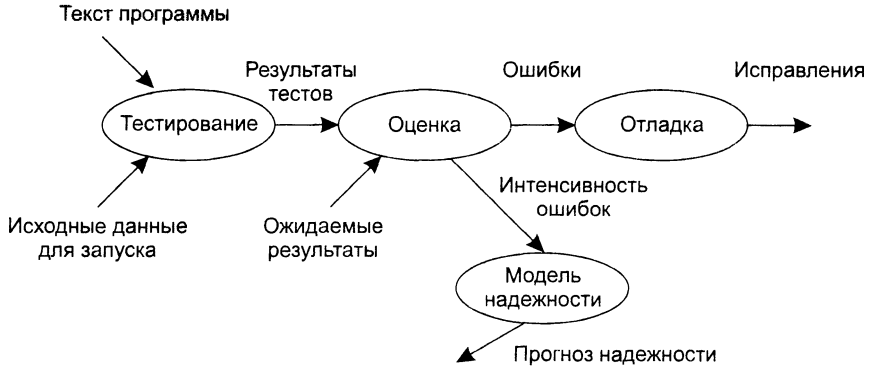


Рис. 14.1. Информационные потоки процесса тестирования

На входе процесса тестирования три потока:

- текст программы;
- исходные данные для запуска программы;
- ожидаемые результаты.

Выполняются тесты, все полученные результаты оцениваются. Это значит, что реальные результаты тестов сравниваются с ожидаемыми результатами. Когда обнаруживается несовпадение, фиксируется ошибка — начинается отладка. Процесс отладки непредсказуем по времени. На поиск места дефекта и исправление может потребоваться час, день, месяц. Неопределенность в отладке приводит к большим трудностям в планировании действий.

После сбора и оценки результатов тестирования начинается отображение качества и надежности ПО. Если регулярно встречаются серьезные ошибки, требующие проектных изменений, то качество и надежность ПО подозрительны, констатируется необходимость усиления тестирования. С другой стороны, если функции ПО реализованы правильно, а обнаруженные ошибки легко исправляются, может быть сделан один из двух выводов:

- качество и надежность ПО удовлетворительны;
- тесты не способны обнаруживать серьезные ошибки.

В конечном счете, если тесты не обнаруживают ошибок, появляется сомнение в том, что тестовые варианты достаточно продуманы и что в ПО нет скрытых ошибок. Такие ошибки будут, в конечном итоге, обнаруживаться пользователями и корректироваться разработчиком на этапе сопровождения (когда стоимость исправления возрастает в 60–100 раз по сравнению с этапом разработки).

Результаты, накопленные в ходе тестирования, могут оцениваться и более формальным способом. Для этого используют модели надежности ПО, выполняющие прогноз надежности по реальным данным об интенсивности ошибок.

Существует два принципа тестирования программы:

- функциональное тестирование (тестирование «черного ящика»);
- структурное тестирование (тестирование «белого ящика»).

Тестирование «черного ящика»

Известны: функции программы.

Исследуется: работа каждой функции на всей области определения.

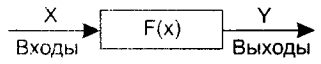


Рис. 14.2. Тестирование «черного ящика»

Как показано на рис. 14.2, основное место приложения тестов «черного ящика» – интерфейс ПО.

Эти тесты демонстрируют:

- как выполняются функции программ;
- как принимаются исходные данные;
- как вырабатываются результаты;
- как сохраняется целостность внешней информации.

При тестировании «черного ящика» рассматриваются системные характеристики программ, игнорируется их внутренняя логическая структура. Исчерпывающее тестирование, как правило, невозможно. Например, если в программе 10 входных величин и каждая принимает по 10 значений, то потребуется 10^{10} тестовых вариантов. Отметим также, что тестирование «черного ящика» не реагирует на многие особенности программных ошибок.

Тестирование «белого ящика»

Известна: внутренняя структура программы.

Исследуются: внутренние элементы программы и связи между ними (рисунок 14.3).

Объектом тестирования здесь является не внешнее, а внутреннее поведение программы. Проверяется корректность построения всех элементов программы и правильность их взаимодействия друг с другом. Обычно анализируются управляющие связи элементов, реже — информационные связи. Тестирование по принципу «белого ящика» характеризуется степенью, в какой тесты выполняют или

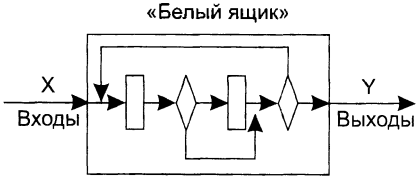


Рис. 14.3. Тестирование «белого ящика»

покрывают логику (исходный текст) программы. Исчерпывающее тестирование также затруднительно. Особенности этого принципа тестирования рассмотрим отдельно.

Особенности тестирования «белого ящика»

Обычно тестирование «белого ящика» основано на анализе управляющей структуры программы [8, 29]. Программа считается полностью проверенной, если проведено исчерпывающее тестирование маршрутов (путей) ее графа управления.

В этом случае формируются тестовые варианты, в которых:

- гарантируется проверка всех независимых маршрутов программы;
- проходятся ветви True, False для всех логических решений;
- выполняются все циклы (в пределах их границ и диапазонов);
- анализируется правильность внутренних структур данных.

Недостатки тестирования «белого ящика»:

1. Количество независимых маршрутов может быть очень велико. Например, если цикл в программе выполняется k раз, а внутри цикла имеется n ветвлений, то количество маршрутов вычисляется по формуле:

$$m = \sum_{i=1}^{k} n^i.$$

   При $n = 5$ и $k = 20$ количество маршрутов $m = 1014$. Примем, что на разработку, выполнение и оценку теста по одному маршруту расходуется 1 мс. Тогда, при работе 24 часа в сутки, 365 дней в году на тестирование уйдет 3170 лет.
2. Исчерпывающее тестирование маршрутов не гарантирует соответствия программы исходным требованиям к ней.
3. В программе могут быть пропущены некоторые маршруты.
4. Нельзя обнаружить ошибки, появление которых зависит от обрабатываемых данных (это ошибки, обусловленные выражениями типа: `if abs (a–b) < eps...`, `if (a+b+c)/3=a...`).

Достоинства тестирования «белого ящика» связаны с тем, что принцип «белого ящика» позволяет учесть особенности программных ошибок:

1. Количество ошибок минимально в «центре» и максимально на «периферии» программы.

2. Предварительные предположения о вероятности потока управления или данных в программе часто бывают некорректны. В результате типовым может стать маршрут, модель вычислений по которому проработана слабо.
3. При записи алгоритма ПО в виде текста на языке программирования возможно внесение типовых ошибок трансляции (синтаксических и семантических).
4. Некоторые результаты в программе зависят не от исходных данных, а от внутренних состояний программы.

Каждая из этих причин является аргументом для проведения тестирования по принципу «белого ящика». Тесты «черного ящика» не смогут реагировать на ошибки таких типов.

Способ тестирования базового пути

Тестирование базового пути — это способ, который основан на принципе «белого ящика». Автор этого способа — Том МакКейб (1976) [73].

Способ тестирования базового пути дает возможность:

- получить оценку комплексной сложности программы;
- использовать эту оценку для определения необходимого количества тестовых вариантов.

Тестовые варианты разрабатываются для проверки базового множества путей (маршрутов) в программе. Они гарантируют однократное выполнение каждого оператора программы при тестировании.

Потоковый граф

Для представления программы используется потоковый граф. Перечислим его особенности.

1. Граф строится отображением управляющей структуры программы. В ходе отображения закрывающие скобки условных операторов и операторов циклов (end if; end loop) рассматриваются как отдельные (фиктивные) операторы.
2. Узлы (вершины) потокового графа соответствуют линейным участкам программы, включают один или несколько операторов программы.
3. Дуги потокового графа отображают поток управления в программе (передачи управления между операторами). Дуга — это ориентированное ребро.
4. Различают операторные и предикатные узлы. Из операторного узла выходит одна дуга, а из предикатного — две дуги.
5. Предикатные узлы соответствуют простым условиям в программе. Составное условие программы отображается в несколько предикатных узлов. Составным называют условие, в котором используется одна или несколько булевых операций (OR, AND).

   Например, фрагмент программы

```
if a OR b
        then x
        else y
end if;
```

вместо прямого отображения в потоковый граф вида, показанного на рис. 14.4,

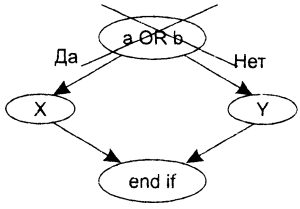


Рис. 14.4. Прямое отображение в потоковый граф

отображается в преобразованный потоковый граф (рис. 14.5)

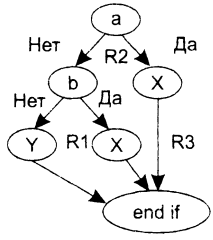


Рис. 14.5. Преобразованный потоковый граф

5. Замкнутые области, образованные дугами и узлами, называют регионами.
6. Окружающая граф среда рассматривается как дополнительный регион. Например, показанный здесь граф имеет три региона — R1, R2, R3.

Пример 14.1. Рассмотрим процедуру сжатия:

```
        процедура сжатие
1        выполнять пока нет EOF
1            читать запись;
2            если запись пуста
3                то удалить запись;
4                иначе если поле a >= поля b
5                             то удалить b;
6                             иначе удалить a;
7a                       конец если;
7a               конец если;
7b           конец выполнять;
8    конец сжатие;
```

Она отображается в потоковый граф, представленный на рис. 14.6.
Видим, что этот потоковый граф имеет четыре региона.

Цикломатическая сложность

Цикломатическая сложность — метрика ПО, которая обеспечивает количественную оценку логической сложности программы. В способе тестирования базового пути цикломатическая сложность определяет:

- количество независимых путей в базовом множестве путей программы;
- верхнюю оценку количества тестов, которое гарантирует однократное выполнение всех операторов.

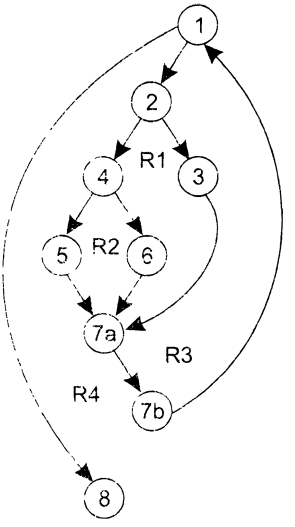


Рис. 14.6. Преобразованный потоковый граф процедуры сжатия

Независимым называется любой путь, который вводит новый оператор обработки или новое условие. В терминах потокового графа независимый путь должен содержать дугу, не входящую в ранее определенные пути.

ПРИМЕЧАНИЕ

Путь начинается в начальном узле, а заканчивается в конечном узле графа. Независимые пути формируются в порядке от самого короткого к самому длинному.

Перечислим независимые пути для потокового графа из примера 14.1:

Путь 1: 1–8;

Путь 2: 1–2–3–7a–7b–1–8;

Путь 3: 1–2–4–5–7a–7b–1–8;

Путь 4: 1–2–4–6–7a–7b–1–8.

Заметим, что каждый новый путь включает новую дугу.

Все независимые пути графа образуют базовое множество путей.

Свойства базового множества путей:

1) тесты, обеспечивающие его проверку, гарантируют:
   - однократное выполнение каждого оператора;
   - выполнение каждого условия по True-ветви и по False-ветви;
2) мощность базового множества равна цикломатической сложности потокового графа.

Значение 2-го свойства трудно переоценить — оно дает априорную оценку количества независимых путей, которое имеет смысл искать в графе.

Цикломатическая сложность вычисляется одним из трех способов:

1) цикломатическая сложность равна количеству регионов потокового графа;

2) цикломатическая сложность определяется по формуле:

$$V(G) = E - N + 2,$$

где E — количество дуг, N — количество узлов потокового графа;

3) цикломатическая сложность формируется по выражению $V(G) = p + 1$, где p — количество предикатных узлов в потоковом графе G.

Вычислим цикломатическую сложность графа из примера 14.1 каждым из трех способов:

1) потоковый граф имеет 4 региона;
2) $V(G)$ = 11 дуг — 9 узлов + 2 = 4;
3) $V(G)$ = 3 предикатных узла + 1 = 4.

Таким образом, цикломатическая сложность потокового графа из примера 14.1 равна четырем.

Шаги способа тестирования базового пути

Для иллюстрации шагов данного способа используем конкретную программу — процедуру вычисления среднего значения:

```
    процедура сред;
1    i := 1;
1    введено := 0;
1    колич := 0;
1    сум := 0;
     вып пока 2 – [вел( i ) <> stop] и [введено <= 500] – 3
4        введено:= введено + 1;
        если 5 – [вел( i ) >= мин] и [вел( i ) <= макс] – 6
7            то колич := колич + 1;
7                 сум := сум + вел( i );
8        конец если;
8        i := i + 1;
9    конец вып;
10    если колич > 0
11       то сред := сум / колич;
12    иначе сред := stop;
13    конец если;
13    конец сред;
```

Заметим, что процедура содержит составные условия (в заголовке цикла и условном операторе). Элементы составных условий для наглядности помещены в рамки.

Шаг 1. На основе текста программы формируется потоковый граф:

- нумеруются операторы текста (номера операторов показаны в тексте процедуры);
- производится отображение пронумерованного текста программы в узлы и вершины потокового графа (рис. 14.7).

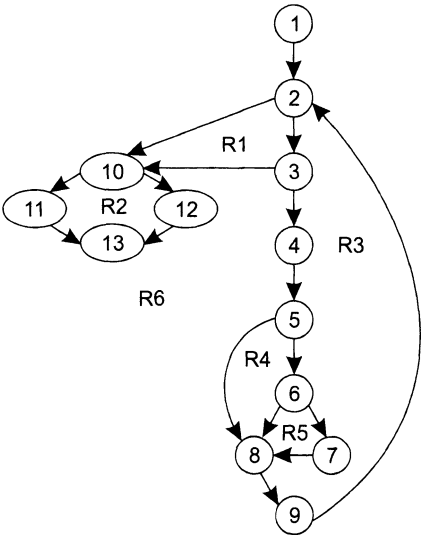


Рис. 14.7. Потоковый граф процедуры вычисления среднего значения

Шаг 2. Определяется цикломатическая сложность потокового графа — по каждой из трех формул:

1. $V(G)$ = 6 регионов;
2. $V(G)$ = 17 дуг − 13 узлов + 2 = 6;
3. $V(G)$ = 5 предикатных узлов + 1 = 6.

Шаг 3. Определяется базовое множество независимых линейных путей:

Путь 1: 1–2–10–11–13; /вел=stop, колич>0.

Путь 2: 1–2–10–12–13; /вел=stop, колич=0.

Путь 3: 1–2–3–10–11–13; /попытка обработки 501-й величины.

Путь 4: 1–2–3–4–5–8–9–2–... /вел<мин.

Путь 5: 1–2–3–4–5–6–8–9–2–... /вел>макс.

Путь 6: 1–2–3–4–5–6–7–8–9–2–... /режим нормальной обработки.

ПРИМЕЧАНИЕ

Для удобства дальнейшего анализа по каждому пути указаны условия запуска. Точки в конце путей 4–6 указывают, что допускается любое продолжение через остаток управляющей структуры графа.

Шаг 4. Подготавливаются тестовые варианты, инициирующие выполнение каждого пути.

Каждый тестовый вариант формируется в следующем виде:

Исходные данные ИД:

Ожидаемые результаты ОЖ.РЕЗ.:

Исходные данные должны выбираться так, чтобы предикатные вершины обеспечивали нужные переключения — запуск только тех операторов, которые перечислены в конкретном пути, причем в требуемом порядке.

Определим тестовые варианты, удовлетворяющие выявленному множеству независимых путей.

Тестовый вариант для пути 1 **ТВ1**:

ИД: вел(k) = допустимое значение, где $k < i$; вел(i) = stop, где $2 \le i \le 500$.

ОЖ.РЕЗ.: корректное усреднение основывается на k величинах и правильном подсчете.

ПРИМЕЧАНИЕ

Путь не может тестироваться самостоятельно, а должен тестироваться как часть путей 4–6 (трудности проверки 11-го оператора).

Тестовый вариант для пути 2 **ТВ2**:

ИД: вел(1) = stop.

ОЖ.РЕЗ.: сред = stop, другие величины имеют начальные значения.

Тестовый вариант для пути 3 **ТВ3**:

ИД: попытка обработки 501-й величины, первые 500 величин должны быть правильными.

ОЖ.РЕЗ.: корректное усреднение основывается на k величинах и правильном подсчете.

Тестовый вариант для пути 4 **ТВ4**:

ИД: вел(i) = допустимое значение, где $i \le 500$; вел(k) < мин, где $k < i$.

ОЖ.РЕЗ.: корректное усреднение основывается на k величинах и правильном подсчете.

Тестовый вариант для пути 5 **ТВ5:**

ИД: вел(i) = допустимое значение, где $I \le 500$; вел(k) > макс, где $k \le i$.

ОЖ.РЕЗ.: корректное усреднение основывается на n величинах и правильном подсчете.

Тестовый вариант для пути 6 **ТВ6**:

ИД: вел(i) = допустимое значение, где $i \le 500$.

ОЖ.РЕЗ.: корректное усреднение основывается на n величинах и правильном подсчете.

Реальные результаты каждого тестового варианта сравниваются с ожидаемыми результатами. После выполнения всех тестовых вариантов гарантируется, что все операторы программы выполнены по меньшей мере один раз.

Важно отметить, что некоторые независимые пути не могут проверяться изолированно. Такие пути должны проверяться при тестировании другого пути (как часть другого тестового варианта).

Пример 14.2. Дана непустая последовательность различных натуральных чисел. Признаком окончания последовательности служит ноль. Процедура определяет порядковый номер и значение минимального числа, количество чисел в последовательности (листинг 14.1).

Листинг 14.1. Первый объект тестирования

```
void test1() {
    int x, i, min, n_min;//1
    scanf("введите число %d\n", &min);//1
    n_min=1;//1
    scanf("введите число %d\n", &x);//1
    i=2;//1
    while (x!=0) {//2
        if (x<min) {//3
            min=x;//4
            n_min=i;//4
        }//4
        scanf("введите число %d\n", &x);//5
        i++;//5
    }//6
    printf("минимальное число \n", min);//7
    printf("номер числа \n", n_min);//7
    printf("количество чисел \n", i-1);//7
}//7
```

Шаг 1. Потоковый граф первого объекта тестирования показан на рис. 14.8.

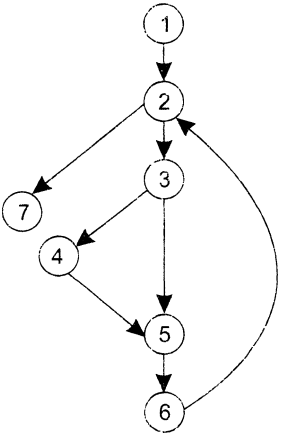


Рис. 14.8. Потоковый граф первого объекта тестирования

Шаг 2. Цикломатическая сложность потокового графа:

1. $V(G)$ = 3 регионов;
2. $V(G)$ = 8 дуг – 7 узлов + 2 = 3;
3. $V(G)$ = 2 предикатных узла + 1 = 3.

Шаг 3. Базовое множество независимых линейных путей:

Путь 1: 1–2–7. // в последовательности одно число.

Путь 2: 1–2–3–5–6–2–7. // в последовательности два числа, первое число меньше второго (или равно ему).

Путь 3: 1–2–3–4–5–6–2–7. // в последовательности два числа, первое число больше второго.

Шаг 4. Тестовые варианты.

Тестовый вариант для пути 1 **ТВ1**:

ИД: [a, 0].

ОЖ.РЕЗ.: n_min=1, min=a, количество=1.

Тестовый вариант для пути 2 **ТВ2**:

ИД: [a, b, 0], a<=b.

ОЖ.РЕЗ.: n_min=1, min=a, количество=2.

Тестовый вариант для пути 3 **ТВ3**:

ИД: [a, b, 0], a>b.

ОЖ.РЕЗ.: n_min=2, min=b, количество=2.

Пример 14.3. Протестируем одну из операций системы управления торговым автоматом (листинг 14.2).

Листинг 14.2. Второй объект тестирования

```
void selectProduct.btnClick(Object sender) {
    String quantity;
    Double number;
    if (mainForm.authorization.technican())//1
        close;//2
    else {//3
        quantity="1";//3
        if (inputQuery(mainForm.productTableName, "Введите количество",
                                                  quantity))//4
            {try//5
                number=strToFloat(quantity);//5
            except//6
                messageDlg("Ошибка ввода", mtError, [mbOK], 0);//6
                exit;//6
            }//7
            if (number>mainForm.productTableOST) {//8
                messageDlg("Не хватает продуктов", mtError, [mbOK], 0);//9
                exit;//9
            }//9
            if (not mainForm.checkSum(number)) {//10
                messageDlg("Не хватает денег", mtError, [mbOK], 0);//11
                exit;//11
            }//11
            mainForm.extractProduct(number);//12
        }//13
    }//14
}//15
```

Шаг 1. Потоковый граф второго объекта тестирования показан на рис. 14.9.

Шаг 2. Цикломатическая сложность потокового графа:

1) $V(G)$ = 6 регионов;
2) $V(G)$ = 19 дуг – 15 узлов + 2 = 6;
3) $V(G)$ = 5 предикатных узлов + 1 = 6.

Шаг 3. Базовое множество независимых линейных путей:

Путь 1: 1–2–15.

Путь 2: 1–3–4–5–6–15.
Путь 3: 1–3–4–13–14–15.
Путь 4: 1–3–4–5–7–8–9–15.
Путь 5: 1–3–4–5–7–8–10–11–15.
Путь 6: 1–3–4–5–7–8–10–12–13–14–15.

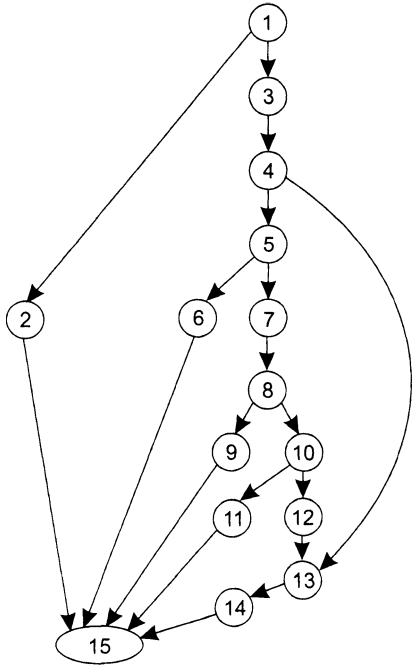


Рис. 14.9. Потоковый граф второго объекта тестирования

Шаг 4. Тестовые варианты.

Тестовый вариант для пути 1 **ТВ1**:

ИД: Функция technican() возвращает значение TRUE (с системой взаимодействует техник, а не покупатель).

ОЖ.РЕЗ.: Закрытие текущей формы и завершение сеанса работы с автоматом.

Тестовый вариант для пути 2 **ТВ2**:

ИД: Функция technican() возвращает значение FALSE (с системой взаимодействует покупатель). Количество покупаемого продукта введено некорректно (ошибка типа), например введена строка, содержащая символы, не являющиеся цифрой или десятичной точкой.

ОЖ.РЕЗ.: Вывод диалогового окна с сообщением: «Ошибка ввода». Завершение сеанса работы.

Тестовый вариант для пути 3 **ТВ3**:

ИД: Функция technican() возвращает значение FALSE (с системой взаимодействует покупатель). Покупатель отказался от ввода количества приобретаемого продукта (нажал Cancel).

ОЖ.РЕЗ.: Завершение сеанса работы.

Тестовый вариант для пути 4 **ТВ4**:

ИД: Функция technican() возвращает значение FALSE (с системой взаимодействует покупатель). Количество покупаемого продукта введено корректно. Но введенное количество превышает количество товара, имеющегося в автомате.

ОЖ.РЕЗ.: Вывод диалогового окна с сообщением: «Не хватает продуктов». Завершение сеанса работы.

Тестовый вариант для пути 5 **ТВ5**:

ИД: Функция technican() возвращает значение FALSE (с системой взаимодействует покупатель). Количество покупаемого продукта введено корректно. Введенное количество меньше или равно количеству товара в автомате. Функция проверки достаточности денежной суммы, предоставляемой покупателем для приобретения товара, возвращает значение FALSE (недостаточно денежных средств).

ОЖ.РЕЗ.: Вывод диалогового окна с сообщением: «Не хватает денег». Завершение сеанса работы.

Тестовый вариант для пути 6 **ТВ6**:

ИД: Функция technican() возвращает значение FALSE (с системой взаимодействует покупатель). Количество покупаемого продукта введено корректно. Введенное количество меньше или равно количеству товара в автомате. Функция проверки достаточности денежной суммы, предоставляемой покупателем для приобретения товара, возвращает значение TRUE (денежных средств достаточно).

ОЖ.РЕЗ.: Выполнение функции extractProduct() — выдача товара в указанном количестве. Завершение сеанса работы.

Способы тестирования условий

Цель этого семейства способов тестирования — строить тестовые варианты для проверки логических условий программы. При этом желательно обеспечить охват операторов из всех ветвей программы.

Рассмотрим используемую здесь терминологию.

Простое условие — булева переменная или выражение отношения.

Выражение отношения имеет вид:

E1 <операция отношения> E2,

где E1, E2 — арифметические выражения, а в качестве операции отношения используется одна из следующих операций $<$, $>$, $=$, $\neq$, $\leq$, $\geq$.

Составное условие состоит из нескольких простых условий, булевых операций и круглых скобок. Будем применять булевы операции OR, AND (&), NOT. Условия, не содержащие выражений отношения, называют булевыми выражениями.

Таким образом, элементами условия являются: булева операция, булева переменная, пара скобок (заключающая простое или составное условие), операция отношения, арифметическое выражение. Эти элементы определяют типы ошибок в условиях.

Если условие некорректно, то некорректен по меньшей мере один из элементов условия. Следовательно, в условии возможны следующие типы ошибок:

- ошибка булевой операции (наличие некорректных/отсутствующих/избыточных булевых операций);
- ошибка булевой переменной;
- ошибка булевой скобки;
- ошибка операции отношения;
- ошибка арифметического выражения.

Способ тестирования условий ориентирован на тестирование каждого условия в программе. Методики тестирования условий имеют два достоинства. Во-первых, достаточно просто выполнить измерение тестового покрытия условия. Во-вторых, тестовое покрытие условий в программе – это фундамент для генерации дополнительных тестов программы.

Целью тестирования условий является определение не только ошибок в условиях, но и других ошибок в программах. Если набор тестов для программы A эффективен для обнаружения ошибок в условиях, содержащихся в A, то вероятно, что этот набор также эффективен для обнаружения других ошибок в A. Кроме того, если методика тестирования эффективна для обнаружения ошибок в условии, то вероятно, что эта методика будет эффективна для обнаружения ошибок в программе.

Существует несколько методик тестирования условий.

Простейшая методика — *тестирование ветвей*. Здесь для составного условия *C* проверяется:

- каждое простое условие (входящее в него);
- True-ветвь;
- False-ветвь.

Другая методика — *тестирование области определения*. В ней для выражения отношения требуется генерация 3–4 тестов. Выражение вида

E1 <операция отношения> E2

проверяется 3 тестами, которые формируют значение E1 большим, чем E2, равным E2 и меньшим, чем E2.

Если операция отношения неправильна, а E1 и E2 корректны, то эти три теста гарантируют обнаружение ошибки операции отношения.

Для определения ошибок в E1 и E2 тест должен сформировать значение E1 большим или меньшим, чем E2, причем обеспечить как можно меньшую разницу между этими значениями.

Для булевых выражений с n переменными требуется набор из 2^n тестов. Этот набор позволяет обнаружить ошибки булевых операций, переменных и скобок, но практичен только при малом n. Впрочем, если в булево выражение каждая булева переменная входит только один раз, то количество тестов легко уменьшается.

Обсудим способ тестирования условий, базирующийся на приведенных выше методиках.

Тестирование ветвей и операций отношений

Способ тестирования ветвей и операций отношений (автор К. Таи, 1989) обнаруживает ошибки ветвления и операций отношения в условии, для которого выполняются следующие ограничения [97]:

- все булевы переменные и операции отношения входят в условие только по одному разу;
- в условии нет общих переменных.

В данном способе используются естественные ограничения условий (ограничения на результат). Для составного условия C, включающего n простых условий, формируется ограничение условия:

$$ОУ_c = (d_1, d_2, d_3, \ldots d_n),$$

где d_i — ограничение на результат i-го простого условия.

Ограничение на результат фиксирует возможные значения аргумента (переменной) простого условия (если он один) или соотношения между значениями аргументов (если их несколько).

Если i-е простое условие является булевой переменной, то его ограничение на результат состоит из двух значений и имеет вид:

$$d_i = (\text{true}, \text{false}).$$

Если j-е простое условие является выражением отношения, то его ограничение на результат состоит из трех значений и имеет следующий вид:

$$d_j = (>, <, =).$$

Говорят, что ограничение условия $ОУ_c$ (для условия C) покрывается выполнением C, если в ходе этого выполнения результат каждого простого условия в C удовлетворяет соответствующему ограничению в $ОУ_c$.

На основе ограничения условия $ОУ$ создается ограничивающее множество $ОМ$, элементы которого являются сочетаниями всех возможных значений d_1, d_2, $d_3, \ldots d_n$.

Ограничивающее множество — удобный инструмент для записи задания на тестирование, ведь оно составляется из сведений о значениях переменных, которые влияют на значение проверяемого условия. Поясним это на примере. Положим, надо проверить условие, составленное из трех простых условий:

$$b \& (x > y) \& a.$$

Условие принимает истинное значение, если все простые условия истинны. В терминах значений простых условий это соответствует записи

(true, true, true),

а в терминах ограничений на значения аргументов простых условий — записи

(true, >, true).

Ясно, что вторая запись является прямым руководством для написания теста. Она указывает, что переменная b должна иметь истинное значение, значение переменной x должно быть больше значения переменной y, и, наконец, переменная a должна иметь истинное значение.

Итак, каждый элемент OM задает отдельный тестовый вариант. Исходные данные тестового варианта должны обеспечить соответствующую комбинацию значений простых условий, а ожидаемый результат равен значению составного условия.

Пример 14.4. В качестве примера рассмотрим два типовых составных условия:

$$C_{\&} = a \ \& \ b, \ C_{or} = a \ or \ b,$$

где a и b – булевы переменные.

Соответствующие ограничения условий принимают вид:

$$ОУ_{\&} = (d_1, d_2), ОУ_{or} = (d_1, d_2),$$

где $d_1 = d_2$ = (true, false).

Ограничивающие множества удобно строить с помощью таблицы истинности (табл. 14.1).

Таблица 14.1. Таблица истинности логических операций

Вариант	a	b	a **&** b	a **or** b
1	false	false	false	false
2	false	true	false	true
3	true	false	false	true
4	true	true	true	true

Видим, что табл. 14.1 задает в OM четыре элемента (и соответственно четыре тестовых варианта). Зададим вопрос – каковы возможности минимизации? Можно ли уменьшить количество элементов в OM?

С точки зрения тестирования нам надо оценить влияние составного условия на программу. Составное условие может принимать только два значения, но каждое из значений зависит от большого количества простых условий. Стоит задача – избавиться от влияния избыточных сочетаний значений простых условий.

Воспользуемся идеей сокращенной схемы вычисления – элементы выражения вычисляются до тех пор, пока они влияют на значение выражения. При тестировании необходимо выявить ошибки переключения, то есть ошибки из-за булевой операции, оперируя значениями простых условий (булевых переменных). При таком инженерном подходе справедливы следующие выводы:

- для условия типа И (a & b) варианты 2 и 3 поглощают вариант 1. Поэтому ограничивающее множество имеет вид:

$$OM_{\&} = \{(\text{false}, \text{true}), (\text{true}, \text{false}), (\text{true}, \text{true})\};$$

- для условия типа ИЛИ (*a or b*) варианты 2 и 3 поглощают вариант 4. Поэтому ограничивающее множество имеет вид:

$$OM_{or} = \{(\text{false}, \text{false}), (\text{false}, \text{true}), (\text{true}, \text{false})\}.$$

Рассмотрим шаги **способа тестирования ветвей и операций отношений**.

Для каждого условия в программе выполняются следующие действия:

1) строится ограничение условий *ОУ*;
2) выявляются ограничения результата по каждому простому условию;
3) строится ограничивающее множество *ОМ*. Построение выполняется путем подстановки в константные формулы $OM_{\&}$ или OM_{or} выявленных ограничений результата;
4) для каждого элемента *ОМ* разрабатывается тестовый вариант.

Пример 14.5. Рассмотрим составное условие C_1 вида:

$$B_1 \,\&\, (E_1 = E_2),$$

где B_1 — булево выражение, E_1, E_2 — арифметические выражения.

Ограничение составного условия имеет вид:

$$OY_{C_1} = (d_1, d_2),$$

где ограничения простых условий равны

$$d_1 = (\text{true}, \text{false}); d_2 = (=, <, >).$$

Проводя аналогию между C_1 и $C_{\&}$ (разница лишь в том, что в C_1 второе простое условие — это выражение отношения), мы можем построить ограничивающее множество для C_1 модификацией

$$OM_{\&} = \{(\text{false}, \text{true}), (\text{true}, \text{false}), (\text{true}, \text{true})\}.$$

Заметим, что true для $(E_1 = E_2)$ означает =, а false для $(E_1 = E_2)$ означает или <, или >. Заменяя (true, true) и (false, true), ограничениями (true, =) и (false, =) соответственно, а (true, false) — ограничениями (true, <) и (true, >), получаем ограничивающее множество для C_1:

$$OM_{C_1} = \{(\text{false}, =), (\text{true}, <), (\text{true}, >), (\text{true}, =)\}.$$

Покрытие этого множества гарантирует обнаружение ошибок булевых операций и операций отношения в C_1.

Пример 14.6. Рассмотрим составное условие C_2 вида:

$$(E_3 > E_4) \,\&\, (E_1 = E_2),$$

где E_1, E_2, E_3, E_4 — арифметические выражения.

Ограничение составного условия имеет вид:

$$OY_{C_2} = (d_1, d_2),$$

где ограничения простых условий равны

$$d_1 = (=, <, >); d_2 = (=, <, >).$$

Проводя аналогию между C_2 и C_1 (разница лишь в том, что в C_2 первое простое условие – это выражение отношения), мы можем построить ограничивающее множество для C_2 модификацией OM_{C_1}:

$$OM_{C_2} = \{(=, =), (<, =), (>, <), (>, >), (>, =)\}.$$

Покрытие этого ограничивающего множества гарантирует обнаружение ошибок операций отношения в C_2.

Пример 14.7. Протестируем операцию, в которой имеются как составное, так и простое условие (листинг 14.3).

Листинг 14.3. Третий объект тестирования

```
void test3() {
    int pol, vozrast;
    scanf("введите ваш пол: 1 – если мужской, 0 – если женский \n", &pol);
    scanf("введите ваш возраст \n", &vozrast);
    if (pol && vozrast>20)//составное условие C1
        print("вы мужчина\n");
    else if (pol)//простое условие C2
            printf("вы юноша\n");
        else
            printf("вы девушка\n");
}
```

Для составного условия C_1:

- Ограничение условия $OУ_{C_1} = (d_1, d_2)$.
- Ограничения на результат простых условий d_1 = (true, false), d_2 = (>, <, =).

Для простого условия C_2:

- Ограничение условия $OУ_{C_2} = (d_3)$.
- Ограничение на результат d_3 = (true, false).

Ограничивающее множество для C_1 удобно строить на основе $OM_{\&}$:

$$OM_{\&} = \{(\text{false}, \text{true}), (\text{true}, \text{false}), (\text{true}, \text{true})\},$$

в котором следует провести замену правой части каждого элемента в круглых скобках. Для тестируемого условия правила замены имеют вид: false заменяется парой значений «меньше» и «равно», а true – значением «больше».

В итоге получаем:

$$OM_{C_1} = \{(\text{false}, >), (\text{true}, <), (\text{true}, =), (\text{true}, >)\}.$$

Простое условие C_2 покрывается ограничивающим множеством:

$$OM_{C_2} = \{(\text{false}), (\text{true})\}.$$

Для создания тестовых вариантов последовательно просматриваем созданные множества.

Тестовый вариант **ТВ1**:

ИД: Поскольку первый элемент множества для условия C_1 имеет вид (false, >), где первый компонент является значением переменной pol, а второй компонент указывает на значение переменной vozrast (оно должно быть больше 20), исходные данные образуют: пол=женский, возраст=30 (лет).

ОЖ.РЕЗ.: Девушка.

Тестовый вариант **ТВ2**:

ИД: пол=мужской, возраст=15.

ОЖ.РЕЗ.: Юноша.

Тестовый вариант **ТВ3**:

ИД:. пол=мужской, возраст=20.

ОЖ.РЕЗ.: Юноша.

Тестовый вариант **ТВ4**:

ИД: пол=мужской, возраст=30.

ОЖ.РЕЗ.: Мужчина.

Тестовый вариант для пути 5 **ТВ5**:

ИД: Определяются первым элементом множества для C_2: пол=женский. возраст=любой.

ОЖ.РЕЗ.: Девушка.

Замечание. При соотношении «возраст>20» пятый вариант поглощается первым тестовым вариантом. При соотношении «возраст<=20» пятый вариант имеет самостоятельное значение.

Тестовый вариант **ТВ6**:

ИД: пол=мужской, возраст<=20.

ОЖ.РЕЗ.: Юноша.

Замечание. Этот вариант поглощается вариантами 2 и 3.

Вывод: для тестирования операции нужно использовать пять тестов.

Пример 14.8. Протестируем операцию для работы с торговым автоматом, содержание которой иллюстрирует листинг 14.4.

Листинг 14.4. Четвертый объект тестирования

```
void test4(const Double summa){
    if (summa>0)//условие C1
        moneyMachine.getMoney(summa);
    else if (summa<0 && authorization.technican()) {//условие C2
        moneyMachine.putMoney(summa);
        printf("Выдана следующая сумма денег \n", summa);
    }
}
```

Для простого условия C_1:

- Ограничение условия $ОУ_{C_1} = (d_1)$.
- Ограничение на результат $d_1 = (>, <, =)$.

Простое условие C_1 покрывается ограничивающим множеством:

$$OM_{C_1} = \{(>), (<), (=)\}.$$

Для составного условия C_2:

- Ограничение условия $OУ_{C_2} = (d_2, d_3)$.
- Ограничения на результат простых условий $d_2 = (>, <, =)$, $d_3 = (\text{true}, \text{false})$.

Ограничивающее множество для C_2 удобно строить на основе $OM_{\&}$:

$$OM_{\&} = \{(\text{false}, \text{true}), (\text{true}. \text{false}), (\text{true}, \text{true})\},$$

в котором следует произвести замену левой части каждого элемента в круглых скобках. Для тестируемого условия правила замены имеют вид: *false* заменяется парой значений «равно» и «больше», а *true* – значением «меньше».

В итоге получаем:

$$OM_{C_2} = \{(=, \text{true}), (>, \text{true}), (<, \text{false}), (<, \text{true})\}.$$

Для создания тестовых вариантов последовательно просматриваем созданные множества.

Начинаем с множества для C_1.

Тестовый вариант **ТВ1**:

ИД: Поскольку первый элемент множества для условия C_1 имеет вид (>) и указывает на значение переменной summa (оно должно быть больше ноля), исходные данные образуют: summa>0.

ОЖ.РЕЗ.: Выполнение операции moneyMachine.putMoney(summa) — прием денег от покупателя.

Тестовый вариант **ТВ2**:

ИД: summa<0.

ОЖ.РЕЗ.: Результат не определен, так как неизвестно кто работает с торговым автоматом.

Тестовый вариант **ТВ3**:

ИД:. summa=0.

ОЖ.РЕЗ.: Ничего не происходит.

Теперь обрабатываем элементы ограничивающего множества для C_2.

Тестовый вариант **ТВ4**:

ИД: summa=0, с автоматом работает техник.

ОЖ.РЕЗ.: Ничего не происходит, поскольку не введена отрицательная сумма денег.

Тестовый вариант для пути 5 **ТВ5**:

ИД: summa>0, с автоматом работает техник.

ОЖ.РЕЗ.: Тест реализовать нельзя, поскольку техник по определению не может ввести в автомат положительную сумму денег.

Тестовый вариант **ТВ6**:

ИД: summa<0, с автоматом работает покупатель.

ОЖ.РЕЗ.: Ничего не происходит, поскольку покупателю запрещено изымать деньги из автомата.

Тестовый вариант **ТВ7**:

ИД: summa<0, с автоматом работает техник.

ОЖ.РЕЗ.: Техник изымает из автомата введенную отрицательную сумму денег (работает операция moneyMachine.putMoney(summa)). Автомат оповещает техника о выдаче конкретной суммы. Резонно предположить, что это действие заносится в журнал.

Очевидно, что варианты 6 и 7 поглощают вариант 2. Кроме того, вариант 5 тоже надо исключить (как нереализуемый). И наконец, имеет смысл доопределить вариант 3 так: «с автоматом работает покупатель», поскольку ввод нулевой суммы техником предусматривается вариантом 4.

Вывод: для тестирования операции нужно использовать пять тестов с номерами 1, 3, 4, 6 и 7.

Способ тестирования потоков данных

В предыдущих способах тесты строились на основе анализа управляющей структуры программы. В данном способе анализу подвергается информационная структура программы.

Работу любой программы можно рассматривать как обработку потока данных, передаваемых от входа в программу к ее выходу.

Рассмотрим пример.

Пусть потоковый граф программы имеет вид, представленный на рис. 14.10. В нем сплошные дуги — это связи по управлению между операторами в программе. Пунктирные дуги отмечают информационные связи (связи по потокам данных). Обозначенные здесь информационные связи соответствуют следующим допущениям:

- в вершине 1 определяются значения переменных *a*, *b*;
- значение переменной *a* используется в вершине 4;
- значение переменной *b* используется в вершинах 3, 6;
- в вершине 4 определяется значение переменной *c*, которая используется в вершине 14.

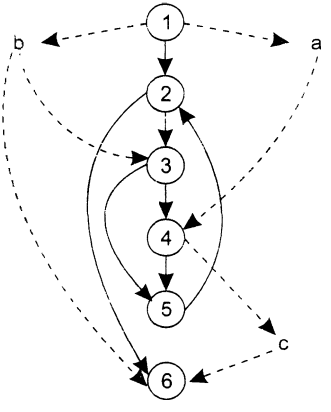


Рис. 14.10. Граф программы с управляющими и информационными связями

В общем случае для каждой вершины графа можно записать:

- множество определений данных

  DEF(i) = { x | i-я вершина содержит определение x};

- множество использований данных:

  USE (i) = { x | i-я вершина использует x}.

Под *определением данных* понимают действия, изменяющие элемент данных. Признак определения — имя элемента стоит в левой части оператора присваивания:

$$x:=f(...).$$

Использование данных — это применение элемента в выражении, где происходит обращение к элементу данных, но не изменение элемента. Признак использования — имя элемента стоит в правой части оператора присваивания:

$$\square:=f(x).$$

Здесь место подстановки другого имени отмечено прямоугольником (прямоугольник играет роль метки-заполнителя).

Назовем *DU-цепочкой* (*цепочкой определения-использования*) конструкцию $[x, i, j]$, где i, j — имена вершин; x определена в i-й вершине ($x \in \mathrm{DEF}(i)$) и используется в j-й вершине ($x \in \mathrm{USE}(j)$).

В нашем примере существуют следующие *DU*-цепочки:

$$[a, 1, 4], [b, 1, 3], [b, 1, 6], [c, 4, 6].$$

Способ *DU-тестирования* требует охвата всех *DU*-цепочек программы. Таким образом, разработка тестов здесь проводится на основе анализа жизни всех данных программы.

Очевидно, что для подготовки тестов требуется выделение маршрутов — путей выполнения программы на управляющем графе. Критерий для выбора пути — покрытие максимального количества *DU*-цепочек.

Шаги способа DU-тестирования:

1) построение управляющего графа (*УГ*) программы;
2) построение информационного графа (*ИГ*);
3) формирование полного набора *DU*-цепочек;
4) формирование полного набора отрезков путей в управляющем графе (отображением набора *DU*-цепочек информационного графа, рис. 14.11);
5) построение маршрутов — полных путей на управляющем графе, покрывающих набор отрезков путей управляющего графа;
6) подготовка тестовых вариантов.

Достоинства DU-тестирования:

- простота необходимого анализа операционно-управляющей структуры программы;
- простота автоматизации.

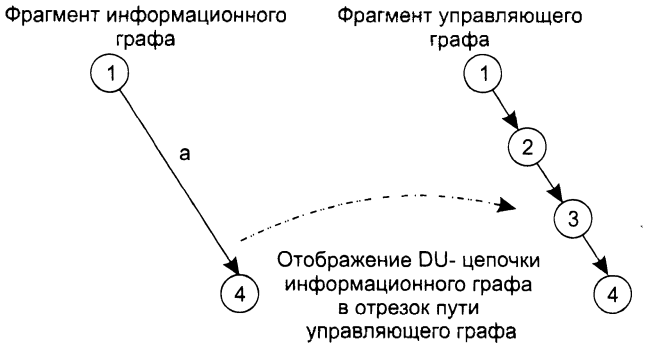


Рис. 14.11. Отображение DU-цепочки в отрезок пути

Недостаток DU-тестирования: трудности в выборе минимального количества максимально эффективных тестов.

Область использования DU-тестирования: программы с вложенными условными операторами и операторами цикла.

Пример 14.9. Протестируем операцию, граф которой представлен на рис. 14.12. На этом графе показаны преобразования переменных, выполняемые в его узлах. Стрелки передачи управления помечены условиями переходов.

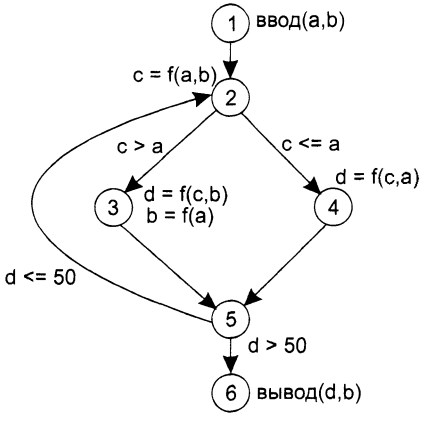


Рис. 14.12. Управляющий граф пятого объекта тестирования

Построим информационный граф операции (рис. 14.13). В нем информационные связи подписаны именами пересылаемых переменных.

Обозначенные здесь информационные связи соответствуют следующим допущениям:

- в узле 1 определяются значения переменных *a* и *b*;
- в узле 3 повторно определяется значение переменной *b*;
- в узле 2 определяется значение переменной *c*;
- в узлах 3 и 4 определяется значение переменной *d*;

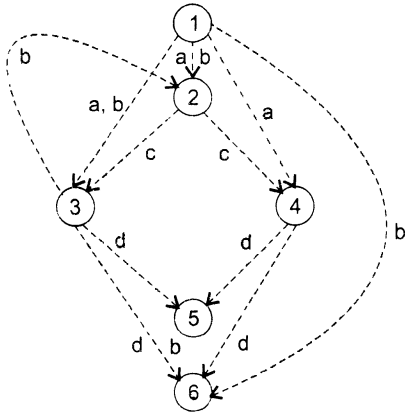


Рис. 14.13. Информационный граф пятого объекта тестирования

- значение переменной *a* используется в узлах 2, 3 и 4;
- значение переменной *b* используется в узлах 2, 3 и 6;
- значение переменной *c* используется в узлах 3 и 4;
- значение переменной *d* используется в узлах 5 и 6.

Сформируем полный набор *DU*-цепочек:
[a, 1, 2], [a, 1, 3], [a, 1, 4].
[b, 1, 2], [b, 1, 3] [b, 1, 6], [b, 3, 2], [b, 3, 6].
[c, 2, 3], [c, 2, 4].
[d, 3, 5]. [d, 3, 6] [d, 4, 5], [d, 4, 6].

Отобразим все *DU*-цепочки информационного графа на управляющий граф, получим отрезки путей (табл. 14.2).

Таблица 14.2. Отображение информационного графа на управляющий граф

Элемент данных	Отрезки путей на управляющем графе
a	{1-2}, {1-2-3}, {1-2-4}
b	{1-2}, {1-2-3}, {1-2-4-5-6}, {3-5-6, {3-5-2}
c	{2-3}, {2-4}
d	{3-5}, {3-5-6}, {4-5}, {4-5-6}

Теперь из отрезков путей надо построить маршруты — полные пути на управляющем графе, каждый из которых начинается в начальном узле графа, а завершается в конечном узле графа. Это даст нам возможность запускать программу в точке старта, ожидая ее завершение в точке финиша.

Алгоритм строительства прост: вынимай отрезки из табл. 14.2 и, глядя на управляющий граф, соединяй их в маршрут. Строительство заканчивается, когда таблица опустеет.

В нашем случае получаем три маршрута:

Путь 1: 1-2-4-5-6.

Путь 2: 1–2–3–5–6.

Путь 3: 1–2–3–5–2–4–5–6.

При строительстве третьего пути получилась заминка: мы вынули из таблицы последний отрезок {3–5–2} и ... пристроили к нему входной сегмент, протянувшийся к стартовому узлу, и выходной сегмент, достигший финального узла.

Последний шаг работы показывать не будем. Он «традиционен» – создание тестовых вариантов, соответствующих маршрутам. Правда, возможны неожиданности. Например, нет такого набора значений (переменных), которые обеспечивают прохождение какого-то маршрута. Тогда приходится откатываться назад и повторять процедуру строительства маршрутов.

Пример 14.10. Применим данный способ тестирования к операции из примера 14.3. Содержание операции было представлено на листинге 14.2.

Шаг 1. Построение управляющего графа. Управляющий потоковый граф тестируемой операции был показан на рис. 14.9.

Шаг 2. Построение информационного графа. Отметим на управляющем графе пунктирными линиями информационные связи (связи по потокам данных). Граф с управляющими и информационными связями показан на рис. 14.14.

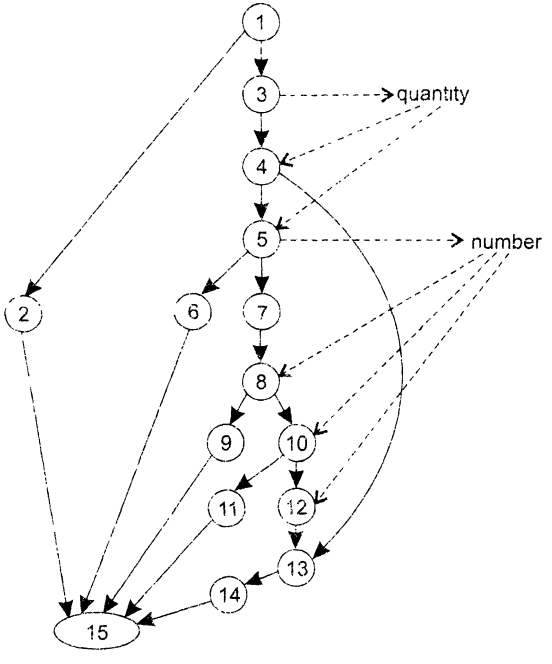


Рис. 14.14. Граф второго объекта тестирования с управляющими и информационными связями

Обозначенные здесь информационные связи соответствуют следующим допущениям:

- в узле 3 определяется значение переменной quantity;
- значение переменной quantity используется в узлах 4 и 5;

- в узле 5 определяется значение переменной number;
- значение переменной number используется в узлах 8, 10 и 12.

Шаг 3. Формирование полного набора *DU*-цепочек:

[quantity, 3, 4], [quantity, 3, 5], [number, 5, 8], [number, 5, 10], [number, 5, 12].

Шаг 4. Формирование полного набора отрезков путей на управляющем графе.

Отобразим набор *DU*-цепочек информационного графа на управляющий граф, получим следующие отрезки:

3–4,
3–4–5,
5–7–8,
5–7–8–10,
5–7–8–10–12.

Шаг 5. Построение маршрутов — полных путей на управляющем графе, покрывающих набор отрезков путей. Для нашей операции все отрезки путей можно покрыть с помощью одного маршрута:

1–3–4–5–7–8–10–12–13–14–15.

Шаг 6. Построение тестового варианта, соответствующего маршруту:

ИД: Функция technican() возвращает значение FALSE (с системой взаимодействует покупатель). Количество покупаемого продукта введено корректно. Введенное количество меньше или равно количеству товара в автомате. Функция проверки достаточности денежной суммы, предоставляемой покупателем для приобретения товара, возвращает значение TRUE (денежных средств достаточно).

ОЖ.РЕЗ.: Выполнение функции extractProduct() — выдача товара в указанном количестве. Завершение сеанса работы.

Тестирование циклов

Цикл — наиболее распространенная конструкция алгоритмов, реализуемых в ПО. Тестирование циклов производится по принципу «белого ящика», при проверке циклов основное внимание обращается на правильность конструкций циклов.

Различают 4 типа циклов: простые, вложенные, объединенные, неструктурированные. Структура циклов приведена на рис. 14.15.

Простые циклы

Для проверки простых циклов с количеством повторений n может использоваться один из следующих наборов тестов:

1) прогон всего цикла;
2) только один проход цикла;
3) два прохода цикла;
4) m проходов цикла, где $m < n$;
5) $n - 1$, n, $n + 1$ проходов цикла.

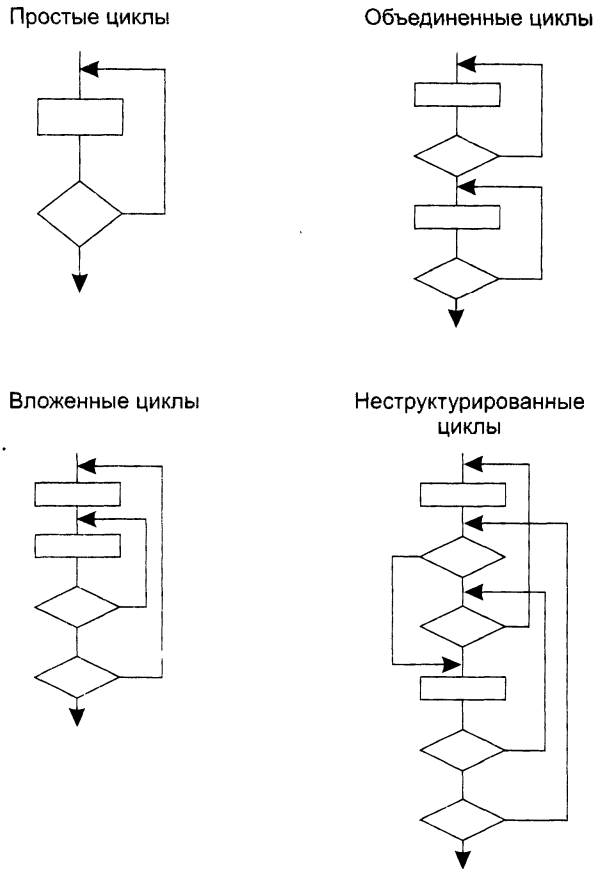


Рис. 14.15. Типовые структуры циклов

Вложенные циклы

С увеличением уровня вложенности циклов количество возможных путей резко возрастает. Это приводит к нереализуемому количеству тестов [29]. Для сокращения количества тестов применяется специальная методика, в которой используются такие понятия, как объемлющий и вложенный цикл (рис. 14.16).

Рис. 14.16. Объемлющий и вложенный циклы

Порядок тестирования вложенных циклов иллюстрирует рис. 14.17.

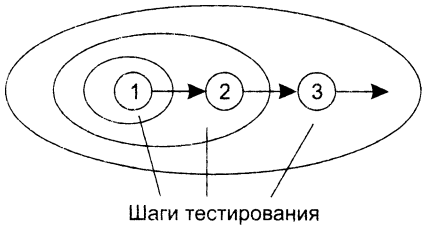


Рис. 14.17. Шаги тестирования вложенных циклов

Шаги тестирования:

1. Выбирается самый внутренний цикл. Устанавливаются минимальные значения параметров всех остальных циклов.
2. Для внутреннего цикла проводятся тесты простого цикла. Добавляются тесты для исключенных значений и значений, выходящих за пределы рабочего диапазона.
3. Переходят в следующий по порядку объемлющий цикл. Выполняют его тестирование. При этом сохраняются минимальные значения параметров для всех внешних (объемлющих) циклов и типовые значения для всех вложенных циклов.
4. Работа продолжается до тех пор, пока не будут протестированы все циклы.

Объединенные циклы

Если каждый из циклов независим от других, то используется техника тестирования простых циклов. При наличии зависимости (например, конечное значение счетчика первого цикла используется как начальное значение счетчика второго цикла) используется методика для вложенных циклов.

Неструктурированные циклы

Неструктурированные циклы тестированию не подлежат. Этот тип циклов должен быть переделан с помощью структурированных программных конструкций.

Пример 14.10. Протестируем операцию с вложенными циклами (листинг 14.5). Здесь речь идет об обработке трехмерного массива $10 \times 10 \times 10$.

Листинг 14.5. Шестой объект тестирования

```
int test6(const int m[] [10] [10] {
    int a, b, c, i, j, k, sum;
    scanf("1-й параметр %d \n", &a);
    scanf("2-й параметр %d \n", &b);
    scanf("3-й параметр %d \n", &c);
    sum=0;
    for (i=0; i<a; i++)
        for (j=0; j<b; j++)
            for (k=0; k<c; k++)
                sum += m[i] [j] [k];
    return sum;
}
```

Используем методику тестирования вложенных циклов. Простой цикл будем тестировать на минимальном и максимальном значениях параметра цикла.

Работу начнем с внутреннего цикла (параметр цикла — k), затем перейдем к среднему (параметр цикла — j), а завершим — внешним циклом (параметр цикла — i).

Тестовый вариант **ТВ1**:

$$ИД: a = 1, b = 1, c = 1.$$
$$ОЖ.РЕЗ.: \text{sum} = m[0][0][0].$$

Тестовый вариант **ТВ2**:

$$ИД: a = 1, b = 1, c = 10.$$
$$ОЖ.РЕЗ.: \text{sum} = \sum_{k=0}^{9} m[0][0][k].$$

Тестовый вариант **ТВ3**:

$$ИД: a = 1, b = 10, c = 5.$$
$$ОЖ.РЕЗ.: \text{sum} = \sum_{j=0}^{9}\sum_{k=0}^{4} m[0][j][k].$$

Тестовый вариант **ТВ4**:

$$ИД: a = 10, b = 5, c = 2.$$
$$ОЖ.РЕЗ.: \text{sum} = \sum_{i=0}^{9}\sum_{j=0}^{4}\sum_{k=0}^{1} m[i][j][k].$$

Контрольные вопросы и упражнения

1. Определите понятие тестирования.
2. Что такое тест? Поясните содержание процесса тестирования.
3. Что такое исчерпывающее тестирование?
4. Какие задачи решает тестирование?
5. Каких задач не решает тестирование?
6. Какие принципы тестирования вы знаете? В чем их отличие друг от друга?
7. В чем состоит суть тестирования «черного ящика»?
8. В чем состоит суть тестирования «белого ящика»?
9. Каковы особенности тестирования «белого ящика»?
10. Какие недостатки имеет тестирование «белого ящика»?
11. Какие достоинства имеет тестирование «белого ящика»?
12. Дайте характеристику способа тестирования базового пути.
13. Какие особенности имеет потоковый граф?
14. Поясните понятие независимого пути.

15. Поясните понятие цикломатической сложности.
16. Что такое базовое множество?
17. Какие свойства имеет базовое множество?
18. Какие способы вычисления цикломатической сложности вы знаете?
19. Поясните шаги способа тестирования базового пути.
20. Поясните достоинства, недостатки и область применения способа тестирования базового пути.
21. Дайте общую характеристику способов тестирования условий.
22. Какие типы ошибок в условиях вы знаете?
23. Какие методики тестирования условий вы знаете?
24. Поясните суть способа тестирования ветвей и операций отношений. Какие он имеет ограничения?
25. Что такое ограничение на результат?
26. Что такое ограничение условия?
27. Что такое ограничивающее множество? Чем удобно его применение?
28. Поясните шаги способа тестирования ветвей и операций отношений.
29. Поясните достоинства, недостатки и область применения способа тестирования ветвей и операций отношений.
30. Поясните суть способа тестирования потоков данных.
31. Что такое множество определений данных?
32. Что такое множество использований данных?
33. Что такое цепочка определения-использования?
34. Поясните шаги способа тестирования потоков данных.
35. Поясните достоинства, недостатки и область применения способа тестирования потоков данных.
36. Запрограммируйте и протестируйте операции класса MoneyMachine системы управления торговым автоматом, описанной в главе 13. Используйте способы тестирования базового пути, тестирования ветвей и операций отношения.
37. Запрограммируйте и протестируйте операции класса Display системы управления торговым автоматом, описанной в главе 13. Используйте способ тестирования базового пути и способ тестирования потоков данных.
38. Поясните особенности тестирования циклов.
39. Какие методики тестирования простых циклов вы знаете?
40. Охарактеризуйте шаги тестирования вложенных циклов. Чем обусловлена их последовательность?

Глава 15

Функциональное тестирование программного обеспечения

В этой главе продолжается обсуждение вопросов тестирования ПО на уровне программных модулей. Впрочем, рассматриваемое здесь функциональное тестирование, основанное на принципе «черного ящика», может применяться и на уровне программной системы. После определения особенностей тестирования «черного ящика» в главе описываются популярные способы тестирования: разбиение по классам эквивалентности, анализ граничных значений, тестирование на основе диаграмм причин–следствий.

Особенности тестирования «черного ящика»

Тестирование «черного ящика» (функциональное тестирование) позволяет получить комбинации входных данных, обеспечивающих полную проверку всех функциональных требований к программе [30]. Программное изделие здесь рассматривается как «черный ящик», чье поведение можно определить только исследованием его входов и соответствующих выходов. При таком подходе желательно иметь:

- набор, образуемый такими входными данными, которые приводят к аномалиям поведения программы (назовем его *IT*);
- набор, образуемый такими выходными данными, которые демонстрируют дефекты программы (назовем его *OT*).

Как показано на рис. 15.1, любой способ тестирования «черного ящика» должен:

- выявить такие входные данные, которые с высокой вероятностью принадлежат набору *IT*;
- сформулировать такие ожидаемые результаты, которые с высокой вероятностью являются элементами набора *OT*.

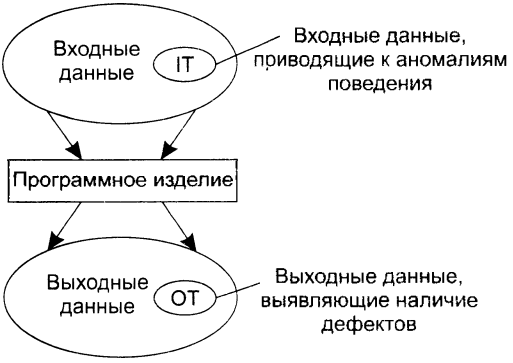


Рис. 15.1. Тестирование «черного ящика»

Во многих случаях определение таких тестовых вариантов основывается на предыдущем опыте инженеров тестирования. Они используют свое знание и понимание области определения для идентификации тестовых вариантов, которые эффективно обнаруживают дефекты. Тем не менее систематический подход к выявлению тестовых данных, обсуждаемый в данной главе, может использоваться как полезное дополнение к эвристическому знанию.

Принцип «черного ящика» не альтернативен принципу «белого ящика». Скорее это дополняющий подход, который обнаруживает другой класс ошибок.

Тестирование «черного ящика» выявляет следующие категории ошибок:

1) некорректные или отсутствующие функции;
2) ошибки интерфейса;
3) ошибки во внешних структурах данных или в доступе к внешней базе данных;
4) ошибки характеристик (необходимая емкость памяти и т. д.);
5) ошибки инициализации и завершения.

Подобные категории ошибок способами «белого ящика» не обнаруживаются.

В отличие от тестирования «белого ящика», которое выполняется на ранней стадии процесса тестирования, тестирование «черного ящика» применяют на поздних стадиях тестирования. При тестировании «черного ящика» пренебрегают управляющей структурой программы. Здесь внимание концентрируется на информационной области определения программной системы.

Техника «черного ящика» ориентирована на решение следующих задач:

- сокращение необходимого количества тестовых вариантов (из-за проверки не статических, а динамических аспектов системы);
- выявление классов ошибок, а не отдельных ошибок.

Способ разбиения по эквивалентности

Разбиение по эквивалентности — самый популярный способ тестирования «черного ящика» [10, 30].

В этом способе входная область данных программы делится на классы эквивалентности. Для каждого класса эквивалентности разрабатывается один тестовый вариант.

Класс эквивалентности — это набор данных с общими свойствами. Обрабатывая разные элементы класса, программа должна вести себя одинаково. Иначе говоря, при обработке любого набора из класса эквивалентности в программе задействуется один и тот же набор операторов (и связей между ними).

На рис. 15.2 каждый класс эквивалентности показан как эллипс. Здесь выделены входные классы эквивалентности допустимых и недопустимых исходных данных, а также классы результатов.

Классы эквивалентности могут быть определены по спецификации на программу.

Рис. 15.2. Разбиение по эквивалентности

Например, если спецификация задает в качестве допустимых входных величин 5-разрядные целые числа в диапазоне 15 000...70 000, то класс эквивалентности допустимых ИД (исходных данных) включает величины от 15 000 до 70 000, а два класса эквивалентности недопустимых ИД составляют:

- числа меньшие, чем 15 000;
- числа большие, чем 70 000.

Класс эквивалентности включает множество значений данных, допустимых или недопустимых по условиям ввода.

Условие ввода может задавать:

1) определенное значение;
2) диапазон значений;
3) множество конкретных величин;
4) булево условие.

Сформулируем *правила формирования классов эквивалентности.*

1. Если условие ввода задает диапазон *n...m*, то определяется один допустимый и два недопустимых класса эквивалентности:
   - V_Class = {*n...m*} — допустимый класс эквивалентности;
   - Inv_Class1 = {*x*|для любого *x*: $x < n$} — первый недопустимый класс эквивалентности;
   - Ivn_Class2 = {*y*|для любого *y*: $y > m$} — второй недопустимый класс эквивалентности.
2. Если условие ввода задает конкретное значение *a*, то определяется один допустимый и два недопустимых класса эквивалентности:
   - V_Class = {*a*};
   - Inv_Class1 = {*x*|для любого *x*: $x < a$};
   - Ivn_Class2 = {*y*|для любого *y*: $y > a$}.
3. Если условие ввода задает множество значений {*a*, *b*, *c*}, то определяется один допустимый и один недопустимый класс эквивалентности:
   - V_Class = {*a*, *b*, *c*};
   - Inv_Class = {*x*|для любого *x*: $(x \neq a)\&(x \neq b)\&(x \neq c)$}.
4. Если условие ввода задает булево значение, например *true*, то определяется один допустимый и один недопустимый класс эквивалентности:
   - V_Class = {true};
   - Inv_Class = {false}.

После построения классов эквивалентности разрабатываются тестовые варианты. Тестовый вариант выбирается так, чтобы проверить сразу наибольшее количество свойств класса эквивалентности.

Способ анализа граничных значений

Как правило, большая часть ошибок происходит на границах области ввода, а не в центре. Анализ граничных значений заключается в получении тестовых вариантов, которые анализируют граничные значения [10, 30, 15]. Данный способ тестирования дополняет способ разбиения по эквивалентности.

Основные отличия анализа граничных значений от разбиения по эквивалентности:

1) тестовые варианты создаются для проверки только ребер классов эквивалентности;
2) при создании тестовых вариантов учитывают не только условия ввода, но и область вывода.

Сформулируем **правила анализа граничных значений**.

1. Если условие ввода задает диапазон *n...m*, то тестовые варианты должны быть построены:
   - для значений *n* и *m*;

- для значений чуть левее *n* и чуть правее *m* на числовой оси.

Например, если задан входной диапазон –1,0...+1,0, то создаются тесты для значений –1,0, +1,0, –1,001, +1,001.

2. Если условие ввода задает дискретное множество значений, то создаются тестовые варианты:
   - для проверки минимального и максимального из значений;
   - для значений чуть меньше минимума и чуть больше максимума.

   Так, если входной файл может содержать от 1 до 255 записей, то создаются тесты для 0, 1, 255, 256 записей.
3. Правила 1 и 2 применяются к условиям области вывода.

   Рассмотрим пример, когда в программе требуется выводить таблицу значений. Количество строк и столбцов в таблице меняется. Задается тестовый вариант для минимального вывода (по объему таблицы), а также тестовый вариант для максимального вывода (по объему таблицы).
4. Если внутренние структуры данных программы имеют предписанные границы, то разрабатываются тестовые варианты, проверяющие эти структуры на их границах.
5. Если входные или выходные данные программы являются упорядоченными множествами (например, последовательным файлом, линейным списком, таблицей), то надо тестировать обработку первого и последнего элементов этих множеств.

Большинство разработчиков используют этот способ интуитивно. При применении описанных правил тестирование границ будет более полным, в связи с чем возрастет вероятность обнаружения ошибок.

Пример 15.1. Рассмотрим применение способов разбиения по эквивалентности и анализа граничных значений на конкретном примере. Положим, что нужно протестировать программу бинарного поиска. Нам известна *спецификация* этой программы. Поиск выполняется в массиве элементов M, возвращается индекс I элемента массива, значение которого соответствует ключу поиска Key.

Предусловия:

1) массив должен быть упорядочен;
2) массив должен иметь не менее одного элемента;
3) нижняя граница массива (индекс) должна быть меньше или равна его верхней границе.

Постусловия:

1) если элемент найден, то флаг Result=True, значение I — номер элемента;
2) если элемент не найден, то флаг Result=False, значение I не определено.

Для формирования классов эквивалентности (и их ребер) надо произвести разбиение области ИД — построить дерево разбиений. Листья дерева разбиений дадут нам искомые классы эквивалентности. Определим стратегию разбиения. На первом уровне будем анализировать выполнимость предусловий, на втором уровне — выполнимость постусловий. На третьем уровне можно анализировать

специальные требования, полученные из практики разработчика. В нашем примере мы знаем, что входной массив должен быть упорядочен. Обработка упорядоченных наборов из четного и нечетного количества элементов может выполняться по-разному. Кроме того, принято выделять специальный случай одноэлементного массива. Следовательно, на уровне специальных требований возможны следующие эквивалентные разбиения:

1) массив из одного элемента;
2) массив из четного количества элементов;
3) массив из нечетного количества элементов, большего единицы.

Наконец на последнем, четвертом уровне критерием разбиения может быть анализ ребер классов эквивалентности. Очевидно, возможны следующие варианты:

1) работа с первым элементом массива;
2) работа с последним элементом массива;
3) работа с промежуточным (ни с первым, ни с последним) элементом массива.

Структура дерева разбиений приведена на рис. 15.3.

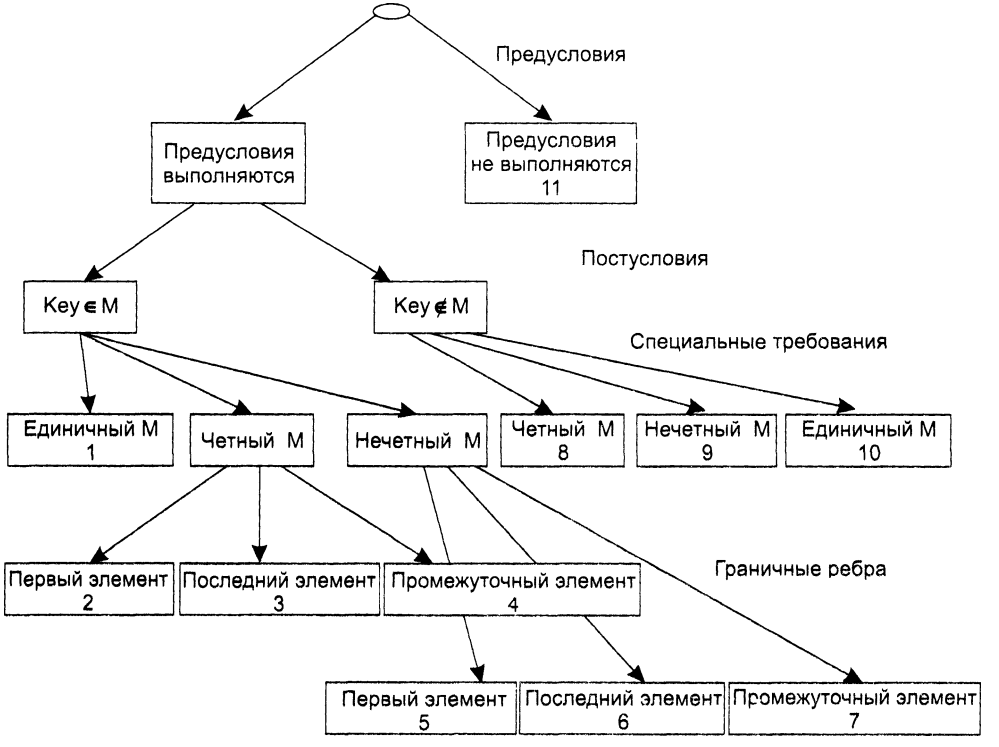


Рис. 15.3. Дерево разбиений области исходных данных бинарного поиска

Это дерево имеет 11 листьев. Каждый лист задает отдельный тестовый вариант. Покажем тестовые варианты, основанные на проведенных разбиениях.

Тестовый вариант 1 (*единичный массив, элемент найден*) **ТВ1**:
ИД: M = 15; Key = 15.
ОЖ.РЕЗ.: Result=True; I = 1.
Тестовый вариант 2 (*четный массив, найден 1-й элемент*) **ТВ2**:
ИД: M = 15, 20, 25, 30, 35, 40; Key = 15.
ОЖ.РЕЗ.: Result=True; I = 1.
Тестовый вариант 3 (*четный массив, найден последний элемент*) **ТВ3**:
ИД: M = 15, 20, 25, 30, 35, 40; Key = 40.
ОЖ.РЕЗ.: Result=True; I = 6.
Тестовый вариант 4 (*четный массив, найден промежуточный элемент*) **ТВ4**:
ИД: M = 15, 20, 25, 30, 35, 40; Key = 25.
ОЖ.РЕЗ.: Result=True; I = 3.
Тестовый вариант 5 (*нечетный массив, найден 1-й элемент*) **ТВ5**:
ИД: M = 15, 20, 25, 30, 35, 40, 45; Key = 15.
ОЖ.РЕЗ.: Result=True; I = 1.
Тестовый вариант 6 (*нечетный массив, найден последний элемент*) **ТВ6**:
ИД: M = 15, 20, 25, 30, 35, 40, 45; Key = 45.
ОЖ.РЕЗ.: Result=True; I = 15.
Тестовый вариант 7 (*нечетный массив, найден промежуточный элемент*) **ТВ7**:
ИД: M = 15, 20, 25, 30, 35, 40, 45; Key = 30.
ОЖ.РЕЗ.: Result=True; I = 4.
Тестовый вариант 8 (*четный массив, не найден элемент*) **ТВ8**:
ИД: M = 15, 20, 25, 30, 35, 40; Key = 23.
ОЖ.РЕЗ.: Result=False; I = ?
Тестовый вариант 9 (*нечетный массив, не найден элемент*) **ТВ9**:
ИД: M = 15, 20, 25, 30, 35, 40, 45; Key = 24.
ОЖ.РЕЗ.: Result=False; I = ?
Тестовый вариант 10 (*единичный массив, не найден элемент*) **ТВ10**:
ИД: M = 15; Key = 0.
ОЖ.РЕЗ.: Result=False; I = ?
Тестовый вариант 11 (*нарушены предусловия*) **ТВ11**:
ИД: M = 15, 10, 5, 25, 20, 40, 35; Key = 35.
ОЖ.РЕЗ.: Аварийное донесение: Массив не упорядочен.

Пример 15.2. Протестируем операцию покупки товара в торговом автомате за наличные деньги.

Предусловия операции:

- покупатель вносит деньги;
- количество запрашиваемого товара введено правильно.

Постусловия операции:

- если нет проблем с деньгами и товаром, то выдается запрошенное количество товара;
- если есть проблемы с деньгами или товаром, то они устраняются.

Для формирования классов эквивалентности применим четырехуровневую стратегию разбиения области исходных данных: на первом уровне работаем

с предусловиями, на втором — с постусловиями, на третьем — со специальными требованиями, на четвертом — с граничными ребрами.

Специальные требования применяются при наличии проблемы постусловий. Они дают два варианта:

1) источник проблемы — деньги;
2) источник проблемы — товар.

Уровень граничных ребер выделяет три варианта:

1) внесена избыточная сумма денег;
2) внесена недостаточная сумма денег;
3) нет необходимого количества товара.

Структура дерева разбиений показана на рис. 15.4.

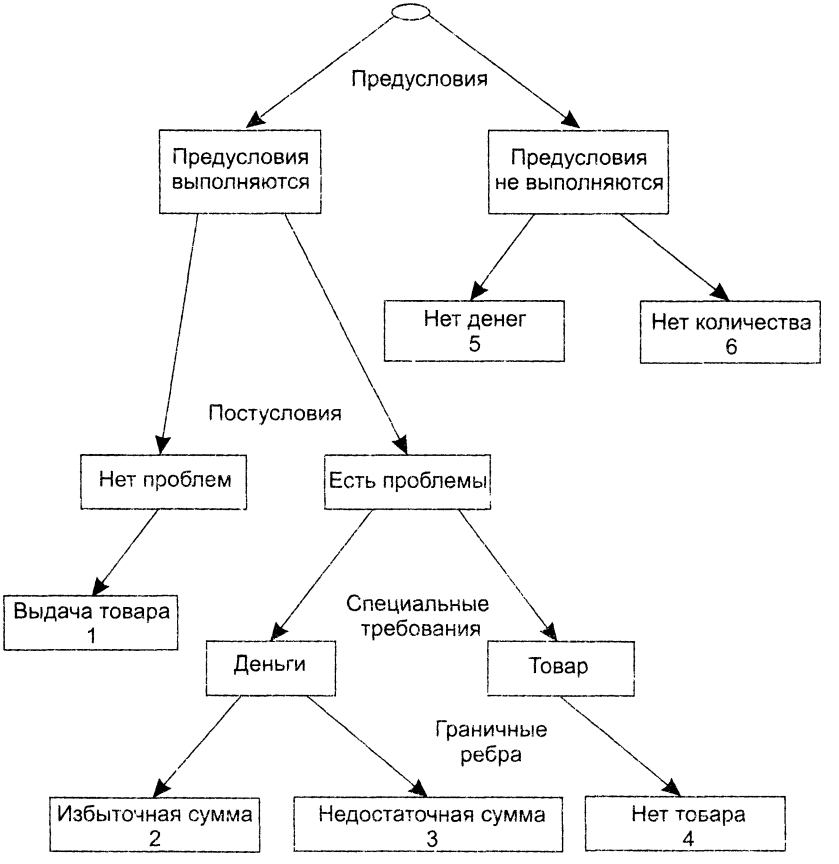


Рис. 15.4. Дерево разбиений области исходных данных для операции покупки товара

Данное дерево имеет 6 листьев, каждый из которых соответствует отдельному тестовому варианту.

Покажем тестовые варианты, основанные на проведенных разбиениях.

Тестовый вариант 1 (*нет проблем, выдача товара*) **ТВ1:**

ИД: Количество приобретаемого товара введено корректно и не превышает количества товара, имеющегося в автомате. Покупатель вносит сумму, равную стоимости покупки.

ОЖ.РЕЗ.: Выдача запрошенного количества товара.

Тестовый вариант 2 (*проблема — деньги, избыточная сумма*) **ТВ2:**

ИД: Количество приобретаемого товара введено корректно и не превышает количества товара, имеющегося в автомате. Покупатель вносит сумму, превышающую стоимость покупки.

ОЖ.РЕЗ.: Выдача запрошенного количества товара и сдачи.

Тестовый вариант 3 (*проблема — деньги, недостаточная сумма*) **ТВ3:**

ИД: Количество приобретаемого товара введено корректно и не превышает количества товара, имеющегося в автомате. Покупатель вносит сумму, меньшую стоимости покупки.

ОЖ.РЕЗ.: Возврат внесенных в автомат денег и вывод сообщения «Не хватает денег».

Тестовый вариант 4 (*проблема — товар, нет товара*) **ТВ4:**

ИД: Количество приобретаемого товара введено корректно, но превышает количество товара, имеющегося в автомате. Покупатель вносит некоторую сумму денег.

ОЖ.РЕЗ.: Возврат внесенных в автомат денег и вывод сообщения «Не хватает продуктов».

Тестовый вариант 5 (*нарушены предусловия, нет денег*) **ТВ5:**

ИД: Количество приобретаемого товара введено корректно, но деньги не внесены.

ОЖ.РЕЗ.: Вывод сообщения «Не хватает денег».

Тестовый вариант 6 (*нарушены предусловия, нет количества*) **ТВ6:**

ИД: Количество приобретаемого товара введено некорректно (например, введено нулевое значение или символ, не являющийся цифрой).

ОЖ.РЕЗ.: Вывод сообщений «Ошибка ввода» и «Введите количество продуктов». Покупателю разрешаются три попытки ввода.

Способ диаграмм причин-следствий

Диаграммы причинно-следственных связей — способ проектирования тестовых вариантов, который обеспечивает формальную запись логических условий и соответствующих действий [10, 88]. Используется автоматный подход к решению задачи.

Шаги способа:

1) для каждого модуля перечисляются причины (условия ввода или классы эквивалентности условий ввода) и следствия (действия или условия вывода). Каждой причине и следствию присваивается свой идентификатор;
2) разрабатывается граф причинно-следственных связей;
3) граф преобразуется в таблицу решений;
4) столбцы таблицы решений преобразуются в тестовые варианты.

Изобразим базовые символы для записи графов причин и следствий (cause-effect graphs).

Сделаем предварительные *замечания*:

1) причины будем обозначать символами c_i, а следствия — символами e_i;
2) каждый узел графа может находиться в состоянии 0 или 1 (0 — состояние отсутствует, 1 — состояние присутствует).

Функция тождество (рис. 15.5) устанавливает, что если значение c_1 есть 1, то и значение e_1 есть 1; в противном случае значение e_1 есть 0.

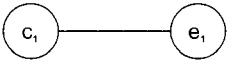


Рис. 15.5. Функция тождество

Функция не (рис. 15.6) устанавливает, что если значение c_1 есть 1, то значение e_1 есть 0; в противном случае значение e_1 есть 1.

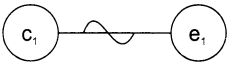


Рис. 15.6. Функция не

Функция или (рис. 15.7) устанавливает, что если c_1 или c_2 есть 1, то e_1 есть 1, в противном случае e_1 есть 0.

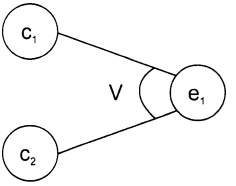


Рис. 15.7. Функция или

Функция и (рис. 15.8) устанавливает, что если и c_1 и c_2 есть 1, то e_1 есть 1, в противном случае e_1 есть 0.

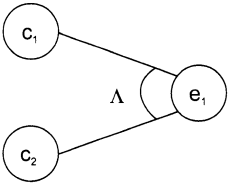


Рис. 15.8. Функция и

Часто определенные комбинации причин невозможны из-за синтаксических или внешних ограничений. Используются перечисленные ниже обозначения ограничений.

Ограничение *E* (исключает, Exclusive, рис. 15.9) устанавливает, что *E* должно быть истинным, если причины нулевые, или только одна из причин — *a* или *b* — принимает значение 1 (*a* и *b* не могут принимать значение 1 одновременно, табл. 15.1). Иными словами, чтобы *E* сохраняло истинное значение, лишь одна из причин *может* принять значение 1.

Таблица 15.1. Таблица истинности для ограничения E

Ограничение E	a	b
true	0	0
true	0	1
true	1	0
false	1	1

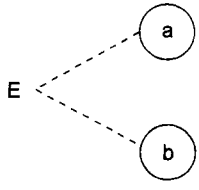


Рис. 15.9. Ограничение E (исключает, Exclusive)

Ограничение I (включает, Inclusive, рис. 15.10) устанавливает, что по крайней мере одна из величин *a*, *b* или *c* всегда должна быть равной 1 (*a*, *b* и *c* не могут принимать значение 0 одновременно, табл.15.2).

Таблица 15.2. Таблица истинности для ограничения I

Ограничение I	a	b	c
false	0	0	0
true	0	0	1
	0	1	0
	0	1	1
	1	0	0
	1	0	1
	1	1	0
	1	1	1

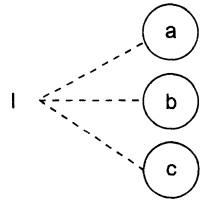


Рис. 15.10. Ограничение I (включает, Inclusive)

Ограничение *O* (одно и только одно, Only one, рис. 15.11) устанавливает, что одна и только одна из величин *a* или *b* должна быть равна 1. Это показано в табл. 15.3.

Таблица 15.3. Таблица истинности для ограничения O

Ограничение O	a	b
false	0	0
true	0	1
true	1	0
false	1	1

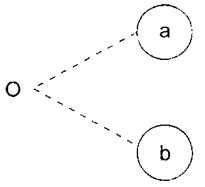


Рис. 15.11. Ограничение O (одно и только одно, Only one)

Ограничение *R* (требует, Requires, рис. 15.12) устанавливает, что если *a* принимает значение 1, то и *b* должна принимать значение 1 (нельзя, чтобы *a* было равно 1, а $b - 0$).

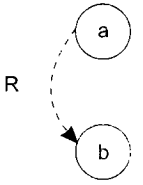


Рис. 15.12. Ограничение R (требует, Requires)

Часто возникает необходимость в ограничениях для следствий.

Ограничение *M* (скрывает, Masks, рис. 15.13) устанавливает, что если следствие *a* имеет значение 1, то следствие *b* должно принять значение 0.

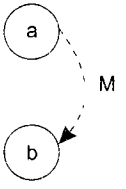


Рис. 15.13. Ограничение M (скрывает, Masks)

Пример 15.3. Для иллюстрации использования способа рассмотрим пример, когда программа выполняет расчет оплаты за электричество по среднему или переменному тарифу.

При расчете по среднему тарифу:

- при месячном потреблении энергии меньшем, чем 100 кВт/ч, выставляется фиксированная сумма;
- при потреблении энергии большем или равном 100 кВт/ч применяется процедура *A* планирования расчета.

При расчете по переменному тарифу:

- при месячном потреблении энергии меньшем, чем 100 кВт/ч, применяется процедура *A* планирования расчета;
- при потреблении энергии большем или равном 100 кВт/ч применяется процедура *B* планирования расчета.

Шаг 1. Причинами являются:

1) расчет по среднему тарифу;
2) расчет по переменному тарифу;
3) месячное потребление электроэнергии меньшее, чем 100 кВт/ч;
4) месячное потребление электроэнергии большее или равное 100 кВт/ч.

На основе различных комбинаций причин можно перечислить следующие следствия:

- 101 — минимальная месячная стоимость;
- 102 — процедура *A* планирования расчета;
- 103 — процедура *B* планирования расчета.

Шаг 2. Разработка графа причинно-следственных связей (рис. 15.14).

Узлы причин перечислим по вертикали у левого края рисунка, а узлы следствий — у правого края рисунка. Для следствия 102 возникает необходимость введения вторичных причин — 11 и 12 — их размещаем в центральной части рисунка.

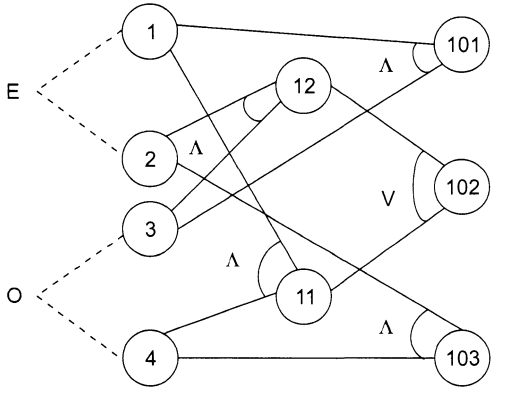


Рис. 15.14. Граф причинно-следственных связей

Шаг 3. Генерация таблицы решений. При генерации причины рассматриваются как условия, а следствия — как действия.

Порядок генерации.

1. Выбирается некоторое следствие, которое должно быть в состоянии «1».
2. Находятся все комбинации причин (с учетом ограничений), которые устанавливают это следствие в состояние «1». Для этого из следствия прокладывается обратная трасса через граф.
3. Для каждой комбинации причин, приводящих следствие в состояние «1», строится один столбец.
4. Для каждой комбинации причин доопределяются состояния всех других следствий. Они помещаются в тот же столбец таблицы решений.
5. Действия 1–4 повторяются для всех следствий графа.

Таблица решений для нашего примера показана в табл. 15.4.

Таблица 15.4. Таблица решений для расчета оплаты за электричество

Номера столбцов			1	2	3	4
Условия	Причины	1	1	0	1	0
		2	0	1	0	1
		3	1	1	0	0
		4	0	0	1	1
	Вторичные причины	11	0	0	1	0
		12	0	1	0	0
Действия	Следствия	101	1	0	0	0
		102	0	1	1	0
		103	0	0	0	1

Шаг 4. Преобразование каждого столбца таблицы в тестовый вариант. В нашем примере таких вариантов четыре.

Тестовый вариант 1 (*столбец 1*) **ТВ1**:

ИД: расчет по среднему тарифу; месячное потребление электроэнергии 75 кВт/ч.

ОЖ.РЕЗ.: минимальная месячная стоимость.

Тестовый вариант 2 (*столбец 2*) **ТВ2**:

ИД: расчет по переменному тарифу; месячное потребление электроэнергии 90 кВт/ч.

ОЖ.РЕЗ.: процедура *A* планирования расчета.

Тестовый вариант 3 (*столбец 3*) **ТВ3**:

ИД: расчет по среднему тарифу; месячное потребление электроэнергии 100 кВт/ч.

ОЖ.РЕЗ.: процедура *A* планирования расчета.

Тестовый вариант 4 (*столбец 4*) **ТВ4**:

ИД: расчет по переменному тарифу; месячное потребление электроэнергии 100 кВт/ч.

ОЖ.РЕЗ.: процедура *B* планирования расчета.

Контрольные вопросы и упражнения

1. Каковы особенности тестирования «черного ящика»?
2. Какие категории ошибок выявляет тестирование «черного ящика»?
3. Какие достоинства имеет тестирование «черного ящика»?
4. Поясните суть способа разбиения по эквивалентности.
5. Что такое класс эквивалентности?
6. Что может задавать условие ввода?
7. Какие правила формирования классов эквивалентности вы знаете?
8. Как выбирается тестовый вариант при тестировании по способу разбиения по эквивалентности?
9. Поясните суть способа анализа граничных значений.
10. Чем способ анализа граничных значений отличается от разбиения по эквивалентности?
11. Поясните правила анализа граничных значений.
12. Что такое дерево разбиений? Каковы его особенности?
13. В чем суть способа диаграмм причин-следствий?
14. Что такое причина?
15. Что такое следствие?
16. Дайте общую характеристику графа причинно-следственных связей.
17. Какие функции используются в графе причин и следствий?
18. Какие ограничения используются в графе причин и следствий?
19. Поясните шаги способа диаграмм причин-следствий.
20. Какую структуру имеет таблица решений в способе диаграмм причин-следствий?
21. Как таблица решений преобразуется в тестовые варианты?
22. Создайте граф причинно-следственных связей торгового автомата, для которого известны *двадцать причин*:
    1) выбрана функция изменения цены товара;
    2) выбрана функция добавления товара в автомат;
    3) выбрана функция изъятия денег из автомата;
    4) пароль техника введен корректно;
    5) пароль техника введен некорректно;
    6) с автоматом работает техник;
    7) количество запрашиваемых продуктов введено некорректно;
    8) PIN-код кредитной карточки введен неправильно;
    9) с автоматом работает покупатель;
    10) выбрана функция покупки продукта за наличные;
    11) выбрана функция покупки продукта с помощью кредитной карты;

12) сумма внесенных денег превышает сумму покупки;
13) сумма внесенных денег соответствует сумме покупки;
14) сумма внесенных денег недостаточна для покупки;
15) количество покупаемых продуктов введено корректно;
16) количество покупаемых продуктов не превышает количества, имеющегося в автомате;
17) количество покупаемых продуктов превышает количество, имеющееся в автомате;
18) PIN-код кредитной карточки введен правильно;
19) денег на счете достаточно;
20) денег на счете недостаточно;

и *десять следствий:*

1) выполнение функции изменения цены на продукт;
2) выполнение функции добавления продукта в автомат;
3) выполнение функции изъятия денег из автомата;
4) выдача сообщения об ошибке доступа к системе;
5) выдача сообщения о некорректном вводе количества товара;
6) выдача сообщения об ошибке ввода PIN-кода;
7) выполнение функции выдачи купленных продуктов и причитающейся сдачи;
8) выполнение функции выдачи купленных продуктов;
9) выдача сообщения о недостаточности денежных средств;
10) выдача сообщения о недостаточности продуктов в автомате;

Протестируйте этот автомат способом диаграмм причин-следствий.

Глава 16

Организация процесса тестирования программного обеспечения

В этой главе излагаются вопросы, связанные с проведением тестирования на всех этапах разработки программной системы. Классический процесс тестирования обеспечивает проверку результатов, полученных на каждом этапе разработки. Как правило, он начинается с тестирования в малом, когда проверяются программные модули, продолжается при проверке объединения модулей в систему и завершается тестированием в большом, при котором проверяется соответствие программного продукта требованиям заказчика и его взаимодействие с другими компонентами компьютерной системы. Данная глава последовательно описывает содержание каждого шага тестирования. Здесь же рассматривается организация отладки ПО, которая проводится для устранения выявленных при тестировании ошибок.

Методика тестирования программных систем

Процесс тестирования объединяет различные способы тестирования в спланированную последовательность шагов, которые приводят к успешному построению программной системы (ПС) [10, 29, 88, 94]. Методика тестирования ПС может быть представлена в виде разворачивающейся спирали (рис. 16.1).

В начале осуществляется *тестирование элементов* (*модулей*), проверяющее результаты этапа *кодирования* ПС. На втором шаге выполняется *тестирование интеграции*, ориентированное на выявление ошибок этапа *проектирования* ПС. На третьем обороте спирали производится *тестирование правильности*, проверяющее корректность этапа *анализа требований к* ПС. На заключительном витке спирали проводится *системное тестирование*, выявляющее дефекты этапа *сбора требований* ПС.

Рис. 16.1. Спираль процесса тестирования ПС

Охарактеризуем каждый шаг процесса тестирования.

1. *Тестирование элементов.* Цель — индивидуальная проверка каждого модуля. Используются способы тестирования «белого ящика».
2. *Тестирование интеграции.* Цель — тестирование сборки модулей в программную систему. В основном применяют способы тестирования «черного ящика».
3. *Тестирование правильности.* Цель — проверить реализацию в программной системе всех функциональных и поведенческих требований, а также требования эффективности. Используются исключительно способы тестирования «черного ящика».
4. *Системное тестирование.* Цель — проверка правильности объединения и взаимодействия всех элементов компьютерной системы, реализации всех системных функций.

Организация процесса тестирования в виде эволюционной разворачивающейся спирали обеспечивает максимальную эффективность поиска ошибок. Однако возникает вопрос — когда заканчивать тестирование?

Ответ практика обычно основан на статистическом критерии: «Можно с 95%-й уверенностью сказать, что провели достаточное тестирование, если вероятность безотказной работы ЦП с программным изделием в течение 1000 часов составляет по меньшей мере 0,995».

Научный подход при ответе на этот вопрос состоит в применении математической модели отказов. Например, для логарифмической модели Пуассона формула расчета текущей интенсивности отказов имеет вид:

$$\lambda(t) = \frac{\lambda_0}{\lambda_0 \times p \times t + 1}, \qquad (16.1)$$

где $\lambda(t)$ — текущая интенсивность программных отказов (количество отказов в единицу времени); λ_0 — начальная интенсивность отказов (в начале тестирования); p — экспоненциальное уменьшение интенсивности отказов за счет обнаруживаемых и устраняемых ошибок; t — время тестирования.

С помощью уравнения (16.1) можно предсказать снижение ошибок в ходе тестирования, а также время, требуемое для достижения допустимо низкой интенсивности отказов.

Тестирование элементов

Объектом тестирования элементов является наименьшая единица проектирования ПС — модуль. Для обнаружения ошибок в рамках модуля тестируются его важнейшие управляющие пути. Относительная сложность тестов и ошибок определяется как результат ограничений области тестирования элементов. Принцип тестирования — «белый ящик», шаг может выполняться для набора модулей параллельно.

Тестированию подвергаются:

- интерфейс модуля;
- внутренние структуры данных;
- независимые пути;
- пути обработки ошибок;
- граничные условия.

Интерфейс модуля тестируется для проверки правильности ввода-вывода тестовой информации. Если нет уверенности в правильном вводе-выводе данных, нет смысла проводить другие тесты.

Исследование внутренних структур данных гарантирует целостность сохраняемых данных.

Тестирование независимых путей гарантирует однократное выполнение всех операторов модуля. При тестировании путей выполнения обнаруживаются следующие категории ошибок: ошибочные вычисления, некорректные сравнения, неправильный поток управления [10].

Наиболее общими ошибками вычислений являются:

1) неправильный или непонятый приоритет арифметических операций;
2) смешанная форма операций;
3) некорректная инициализация;
4) несогласованность в представлении точности;
5) некорректное символическое представление выражений.

Источниками ошибок сравнения и неправильных потоков управления являются:

1) сравнение различных типов данных;
2) некорректные логические операции и приоритетность;
3) ожидание эквивалентности в условиях, когда ошибки точности делают эквивалентность невозможной;
4) некорректное сравнение переменных;
5) неправильное прекращение цикла;
6) отказ в выходе при отклонении итерации;
7) неправильное изменение переменных цикла.

Обычно при проектировании модуля предвидят некоторые ошибочные условия. Для защиты от ошибочных условий в модуль вводят пути обработки ошибок. Такие пути тоже должны тестироваться. Тестирование путей обработки ошибок можно ориентировать на следующие ситуации:

1) донесение об ошибке невразумительно;
2) текст донесения не соответствует обнаруженной ошибке;
3) вмешательство системных средств регистрации аварии произошло до обработки ошибки в модуле;
4) обработка исключительного условия некорректна;
5) описание ошибки не позволяет определить ее причину.

И наконец, перейдем к граничному тестированию. Модули часто отказывают на «границах». Это означает, что ошибки часто происходят:

1) при обработке n-го элемента n-элементного массива;
2) при выполнении m-й итерации цикла с m проходами;
3) при появлении минимального (максимального) значения.

Тестовые варианты, ориентированные на данные ситуации, имеют высокую вероятность обнаружения ошибок.

Тестирование элементов обычно рассматривается как дополнение к этапу кодирования. Оно начинается после разработки текста модуля. Так как модуль не является автономной системой, то для реализации тестирования требуются дополнительные средства, представленные на рис. 16.2.

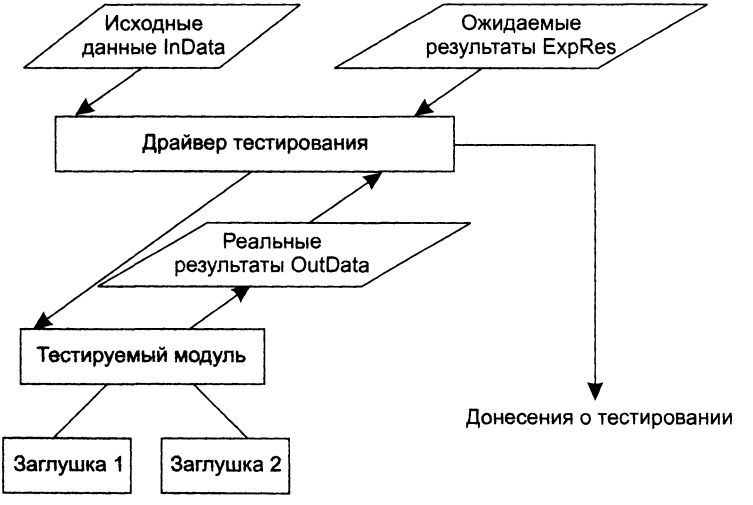


Рис. 16.2. Программная среда для тестирования модуля

Дополнительными средствами являются драйвер тестирования и заглушки. Драйвер — управляющая программа, которая принимает исходные данные (InData) и ожидаемые результаты (ExpRes) тестовых вариантов, запускает в работу тестируемый модуль, получает из модуля реальные результаты (OutData) и формирует

донесения о тестировании. Алгоритм работы тестового драйвера приведен на рис. 16.3.

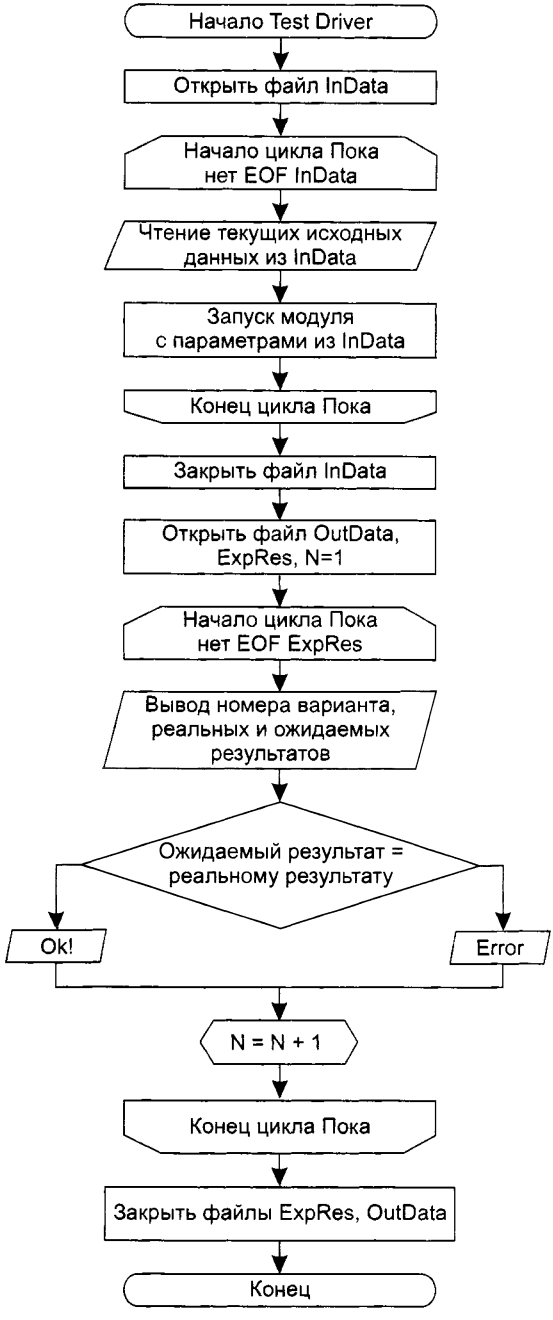


Рис. 16.3. Алгоритм работы драйвера тестирования

Заглушки замещают модули, которые вызываются тестируемым модулем. Заглушка или «фиктивная подпрограмма» реализует интерфейс подчиненного модуля, может выполнять минимальную обработку данных, имитирует прием и возврат данных.

Создание драйвера и заглушек подразумевает дополнительные затраты, так как они не поставляются с конечным программным продуктом.

Если эти средства просты, то дополнительные затраты невелики. Увы, многие модули не могут быть адекватно протестированы с помощью простых дополнительных средств. В этих случаях полное тестирование может быть отложено до шага тестирования интеграции (где драйверы или заглушки также используются).

Тестирование элемента просто осуществить, если модуль имеет высокую связность. При реализации модулем только одной функции количество тестовых вариантов уменьшается, а ошибки легко предсказываются и обнаруживаются.

Тестирование интеграции

Тестирование интеграции поддерживает сборку цельной программной системы.

Цель сборки и тестирования интеграции: взять модули, протестированные как элементы, и построить программную структуру, требуемую проектом [10].

Тесты проводятся для обнаружения ошибок интерфейса. Перечислим некоторые категории ошибок интерфейса:

- потеря данных при прохождении через интерфейс;
- отсутствие в модуле необходимой ссылки;
- неблагоприятное влияние одного модуля на другой;
- подфункции при объединении не образуют требуемую главную функцию;
- отдельные (допустимые) неточности при интеграции выходят за допустимый уровень;
- проблемы при работе с глобальными структурами данных.

Существует два варианта тестирования, поддерживающих процесс интеграции: нисходящее тестирование и восходящее тестирование. Рассмотрим каждый из них.

Нисходящее тестирование интеграции

В данном подходе модули объединяются движением сверху вниз по управляющей иерархии, начиная от главного управляющего модуля. Подчиненные модули добавляются в структуру или в результате поиска в глубину, или в результате поиска в ширину.

Рассмотрим пример (рис. 16.4). Интеграция поиском в глубину будет подключать все модули, находящиеся на главном управляющем пути структуры (по вертикали). Выбор главного управляющего пути отчасти произволен и зависит от характеристик, определяемых приложением. Например, при выборе левого пути прежде всего будут подключены модули M1, M2, M5. Следующим подключается модуль M8 или M6 (если это необходимо для правильного функционирования M2). Затем строится центральный или правый управляющий путь.

При интеграции поиском в ширину структура последовательно проходится по уровням-горизонталям. На каждом уровне подключаются модули, непосредственно подчиненные управляющему модулю — начальнику. В этом случае прежде всего подключаются модули M2, M3, M4. На следующем уровне — модули M5, M6 и т. д.

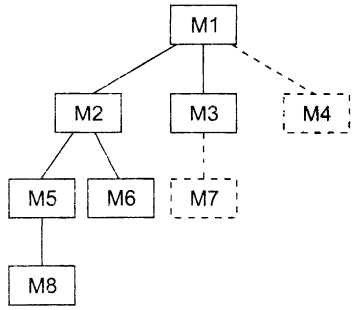


Рис. 16.4. Нисходящая интеграция системы

Опишем возможные шаги процесса нисходящей интеграции.

1. Главный управляющий модуль (находится на вершине иерархии) используется как тестовый драйвер. Все непосредственно подчиненные ему модули временно замещаются заглушками.
2. Одна из заглушек заменяется реальным модулем. Модуль выбирается поиском в ширину или в глубину.
3. После подключения каждого модуля (и установки на нем заглушек) проводится набор тестов, проверяющих полученную структуру.
4. Если в модуле-драйвере уже нет заглушек, производится смена модуля-драйвера (поиском в ширину или в глубину).
5. Выполняется возврат на шаг 2 (до тех пор, пока не будет построена целая структура).

Достоинство нисходящей интеграции: ошибки в главной, управляющей части системы выявляются в первую очередь.

Недостаток: трудности в ситуациях, когда для полного тестирования на верхних уровнях нужны результаты обработки с нижних уровней иерархии.

Существуют 3 возможности борьбы с этим недостатком:

1) откладывать некоторые тесты до замещения заглушек модулями;
2) разрабатывать заглушки, частично выполняющие функции модулей;
3) подключать модули движением снизу вверх.

Первая возможность вызывает сложности в оценке результатов тестирования.

Для реализации второй возможности выбирается одна из следующих категорий заглушек:

- заглушка A — отображает трассируемое сообщение;
- заглушка B — отображает проходящий параметр;
- заглушка C — возвращает величину из таблицы;

- заглушка D — выполняет табличный поиск по ключу (входному параметру) и возвращает связанный с ним выходной параметр.

Категории заглушек представлены на рис. 16.5.

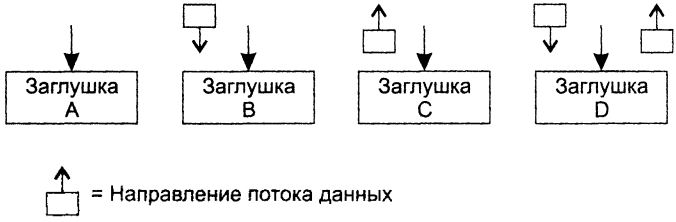


Рис. 16.5. Категории заглушек

Очевидно, что заглушка A наиболее проста, а заглушка D наиболее сложна в реализации.

Этот подход работоспособен, но может привести к существенным затратам, так как заглушки становятся все более сложными.

Третью возможность обсудим отдельно.

Восходящее тестирование интеграции

При восходящем тестировании интеграции сборка и тестирование системы начинается с модулей-атомов, располагаемых на нижних уровнях иерархии. Модули подключаются движением снизу вверх. Подчиненные модули всегда доступны и нет необходимости в заглушках.

Рассмотрим шаги методики восходящей интеграции.

1. Модули нижнего уровня объединяются в кластеры (группы, блоки), выполняющие определенную программную подфункцию.
2. Для координации вводов-выводов тестового варианта пишется драйвер, управляющий тестированием кластеров.
3. Тестируется кластер.
4. Драйверы удаляются, а кластеры объединяются в структуру движением вверх.

Пример восходящей интеграции системы приведен на рис. 16.6.

Модули объединяются в кластеры 1, 2, 3. Каждый кластер тестируется драйвером. Модули в кластерах 1 и 2 подчинены модулю Ma, поэтому драйверы D1 и D2 удаляются и кластеры подключают прямо к Ma. Аналогично драйвер D3 удаляется перед подключением кластера 3 к модулю Mb. В последнюю очередь к модулю Mc подключаются модули Ma и Mb

Рассмотрим различные типы драйверов:

- драйвер A — вызывает подчиненный модуль;
- драйвер B — посылает элемент данных (параметр) из внутренней таблицы;
- драйвер C — отображает параметр из подчиненного модуля;
- драйвер D — является комбинацией драйверов B и C.

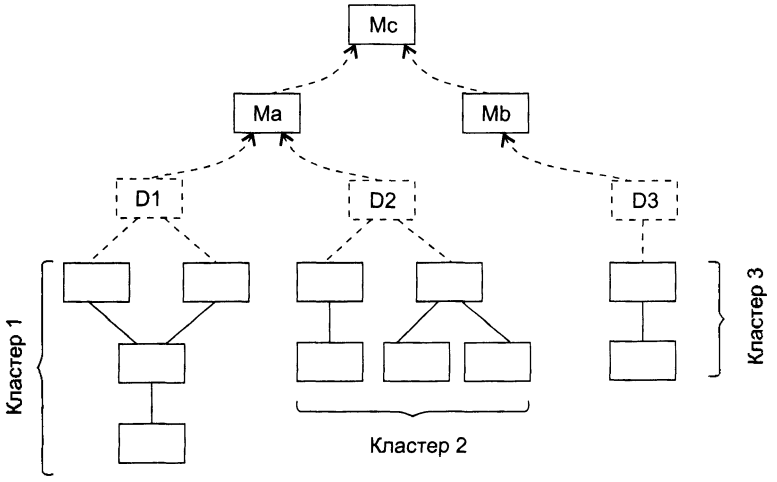


Рис. 16.6. Восходящая интеграция системы

Очевидно, что драйвер A наиболее прост, а драйвер D наиболее сложен в реализации. Различные типы драйверов представлены на рис. 16.7.

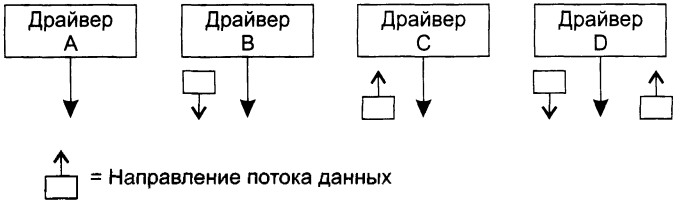


Рис. 16.7. Различные типы драйверов

По мере продвижения интеграции вверх необходимость в выделении драйверов уменьшается. Как правило, в двухуровневой структуре драйверы не нужны.

Сравнение нисходящего и восходящего тестирования интеграции

Нисходящее тестирование:

1) *основной недостаток* — необходимость заглушек и связанные с ними трудности тестирования;
2) *основное достоинство* — возможность раннего тестирования главных управляющих функций.

Восходящее тестирование:

1) *основной недостаток* — система не существует как объект до тех пор, пока не будет добавлен последний модуль;
2) *основное достоинство* — упрощается разработка тестовых вариантов, отсутствуют заглушки.

Возможен комбинированный подход. В нем для верхних уровней иерархии применяют нисходящую стратегию, а для нижних уровней — восходящую стратегию тестирования [10, 29].

При проведении тестирования интеграции очень важно выявить критические модули. Признаки критического модуля:

1) реализует несколько требований к программной системе;
2) имеет высокий уровень управления (находится достаточно высоко в программной структуре);
3) имеет высокую сложность или склонность к ошибкам (как индикатор может использоваться цикломатическая сложность — ее верхний разумный предел составляет 10);
4) имеет определенные требования к производительности обработки.

Критические модули должны тестироваться как можно раньше. Кроме того, к ним должно применяться регрессионное тестирование (повторение уже выполненных тестов в полном или частичном объеме).

Тестирование правильности

После окончания тестирования интеграции программная система (ПС) собрана в единый корпус, интерфейсные ошибки обнаружены и откорректированы. Теперь начинается последний шаг программного тестирования — *тестирование правильности*. Цель — подтвердить, что функции, описанные в спецификации требований к ПС, соответствуют ожиданиям заказчика [88, 94]. Достаточно часто этот вид тестирования называется валидацией, а в русских переводах международных стандартов он получил название аттестации.

Подтверждение правильности ПС выполняется с помощью тестов «черного ящика», демонстрирующих соответствие требованиям. При обнаружении отклонений от спецификации требований создается список недостатков. Как правило, отклонения и ошибки, выявленные при подтверждении правильности, требуют изменения сроков разработки продукта.

Важным элементом подтверждения правильности является проверка конфигурации ПС. Конфигурацией ПС называют совокупность всех элементов информации, вырабатываемых в процессе конструирования ПС. Повторим, что минимальная конфигурация ПС включает следующие элементы:

1) системная спецификация;
2) план программного проекта;
3) спецификация требований к ПС. Работающий или бумажный макет;
4) предварительное руководство пользователя;
5) спецификация проектирования;
6) листинги исходных текстов программ;
7) план и методика тестирования. Тестовые варианты и полученные результаты;
8) руководства по работе и инсталляции;
9) exe-код выполняемой программы;

10) описание базы данных;

11) руководство пользователя по настройке;

12) документы сопровождения. Отчеты о проблемах ПС. Запросы сопровождения. Отчеты об изменениях;

13) стандарты и методики разработки ПС.

Проверка конфигурации гарантирует, что все элементы конфигурации ПС правильно разработаны, учтены и достаточно детализированы для поддержки этапа сопровождения в жизненном цикле ПС.

Разработчик не может предугадать, как заказчик будет реально использовать ПС. Для обнаружения ошибок, которые способен найти только конечный пользователь, используют процесс, включающий альфа- и бета-тестирование.

Альфа-тестирование проводится заказчиком в организации разработчика. Разработчик фиксирует все выявленные заказчиком ошибки и проблемы использования.

Бета-тестирование проводится конечным пользователем в организации заказчика. Разработчик в этом процессе участия не принимает. Фактически бета-тестирование — это реальное применение ПС в среде, которая не управляется разработчиком. Заказчик сам записывает все обнаруженные проблемы и сообщает о них разработчику. Бета-тестирование проводится в течение фиксированного срока (около года). По результатам выявленных проблем разработчик изменяет ПС и тем самым подготавливает продукт полностью на базе заказчика.

Системное тестирование

Системное тестирование подразумевает выход за рамки области действия программного проекта и проводится не только программным разработчиком. Классическая проблема системного тестирования — указание причины. Она возникает, когда разработчик одного элемента компьютерной системы обвиняет разработчика другого элемента в причине возникновения дефекта. Для защиты от подобного обвинения разработчик программного элемента должен:

1) предусмотреть средства обработки ошибки, которые тестируют все вводы информации от других элементов системы;

2) провести тесты, моделирующие неудачные данные или другие потенциальные ошибки интерфейса ПС;

3) записать результаты тестов, чтобы использовать их как доказательство невиновности в случае «указания причины»;

4) принять участие в планировании и проектировании системных тестов, чтобы гарантировать адекватное тестирование ПС.

В конечном счете системные тесты должны проверять, что все системные элементы правильно объединены и выполняют назначенные функции. Рассмотрим основные типы системных тестов [29, 75].

Тестирование восстановления

Многие компьютерные системы должны восстанавливаться после отказов и возобновлять обработку в пределах заданного времени. В некоторых случаях система должна быть отказоустойчивой, то есть отказы обработки не должны быть причиной прекращения работы системы. В других случаях системный отказ должен быть устранен в пределах заданного кванта времени, иначе заказчику наносится серьезный экономический ущерб.

Тестирование восстановления использует самые разные пути для того, чтобы заставить ПС отказать, и проверяет полноту выполненного восстановления. При автоматическом восстановлении оценивается правильность повторной инициализации, механизмы копирования контрольных точек, восстановление данных, перезапуск. При ручном восстановлении оценивается, находится ли среднее время восстановления в допустимых пределах.

Тестирование безопасности

Компьютерные системы очень часто являются мишенью незаконного проникновения. Под проникновением понимается широкий диапазон действий: попытки хакеров проникнуть в систему из-за спортивного интереса, месть рассерженных служащих, взлом мошенниками для незаконной наживы.

Тестирование безопасности проверяет фактическую реакцию защитных механизмов, встроенных в систему, на проникновение.

В ходе тестирования безопасности испытатель играет роль взломщика. Ему разрешено все:

- попытки узнать пароль с помощью внешних средств;
- атака системы с помощью специальных утилит, анализирующих защиты;
- подавление, ошеломление системы (в надежде, что она откажется обслуживать других клиентов);
- целенаправленное введение ошибок в надежде проникнуть в систему в ходе восстановления;
- просмотр несекретных данных в надежде найти ключ для входа в систему.

Конечно, при неограниченном времени и ресурсах хорошее тестирование безопасности взломает любую систему. Задача проектировщика системы — сделать цену проникновения более высокой, чем цена получаемой в результате информации.

Стрессовое тестирование

На предыдущих шагах тестирования способы «белого» и «черного ящиков» обеспечивали полную оценку нормальных программных функций и качества функционирования. Стрессовые тесты проектируются для навязывания программам ненормальных ситуаций. В сущности, проектировщик стрессового теста спрашивает, как сильно можно расшатать систему, прежде чем она откажет?

Стрессовое тестирование производится при ненормальных запросах на ресурсы системы (по количеству, частоте, размеру-объему).

Примеры:

- генерируется 10 прерываний в секунду (при средней частоте 1, 2 прерывания в секунду);
- скорость ввода данных увеличивается прямо пропорционально их важности (чтобы определить реакцию входных функций);
- формируются варианты, требующие максимума памяти и других ресурсов;
- генерируются варианты, вызывающие переполнение виртуальной памяти;
- проектируются варианты, вызывающие чрезмерный поиск данных на диске.

По существу, испытатель пытается разрушить систему. Разновидность стрессового тестирования называется *тестированием чувствительности*. В некоторых ситуациях (обычно в математических алгоритмах) очень малый диапазон данных, содержащийся в границах правильных данных системы, может вызвать ошибочную обработку или резкое понижение производительности. Тестирование чувствительности обнаруживает комбинации данных, которые могут вызвать нестабильность или неправильность обработки.

Тестирование производительности

В системах реального времени и встроенных системах недопустимо ПО, которое реализует требуемые функции, но не соответствует требованиям производительности.

Тестирование производительности проверяет скорость работы ПО в компьютерной системе. Производительность тестируется на всех шагах процесса тестирования. Даже на уровне элемента при проведении тестов «белого ящика» может оцениваться производительность индивидуального модуля. Тем не менее, пока все системные элементы не объединятся полностью, не может быть установлена истинная производительность системы. Иногда тестирование производительности сочетают со стрессовым тестированием. При этом нередко требуется специальный аппаратный и программный инструментарий. Например, часто требуется точное измерение используемого ресурса (процессорного цикла и т. д.). Внешний инструментарий регулярно отслеживает интервалы выполнения, регистрирует события (например, прерывания) и машинные состояния. С помощью инструментария испытатель может обнаружить состояния, которые приводят к деградации и возможным отказам системы.

Искусство отладки

Отладка – это локализация и устранение ошибок. Отладка является следствием успешного тестирования. Это значит, что если тестовый вариант обнаруживает ошибку, то процесс отладки уничтожает ее.

Итак, процессу отладки предшествует выполнение тестового варианта. Его результаты оцениваются, регистрируется несоответствие между ожидаемым и реальным результатом. Несоответствие является симптомом скрытой причины. Про-

цесс отладки пытается сопоставить симптом с причиной, вследствие чего приводит к исправлению ошибки. Возможны два исхода процесса отладки:

1) причина найдена, исправлена, уничтожена;
2) причина не найдена.

Во втором случае отладчик может предполагать причину. Для проверки этой причины он просит разработать дополнительный тестовый вариант, который поможет проверить предположение. Таким образом, запускается итерационный процесс коррекции ошибки.

Возможны разные способы проявления ошибок:

1) программа завершается нормально, но выдает неверные результаты;
2) программа зависает;
3) программа завершается по прерыванию;
4) программа завершается, выдает ожидаемые результаты, но хранимые данные испорчены (это самый неприятный вариант).

Характер проявления ошибок также может меняться. Симптом ошибки может быть:

- постоянным;
- мерцающим;
- пороговым (проявляется при превышении некоторого порога в обработке — 200 самолетов на экране отслеживаются, а 201-й — нет);
- отложенным (проявляется только после исправления маскирующих ошибок).

В ходе отладки мы встречаем ошибки в широком диапазоне: от мелких неприятностей до катастроф. Следствием увеличения ошибок является усиление давления на отладчика — «найди ошибки быстрее!!!» Часто из-за этого давления разработчик устраняет одну ошибку и вносит две новые ошибки.

Английский термин *debugging* (отладка) дословно переводится как ловля блох, который отражает специфику процесса — погоню за объектами отладки, блохами. Рассмотрим, как может быть организован этот процесс ловли блох [10, 88].

Различают две группы методов отладки:

- аналитические;
- экспериментальные.

Аналитические методы базируются на анализе выходных данных для тестовых прогонов. Экспериментальные методы базируются на использовании вспомогательных средств отладки (отладочные печати, трассировки), позволяющих уточнить характер поведения программы при тех или иных исходных данных.

Общая стратегия отладки — обратное прохождение от замеченного симптома ошибки к исходной аномалии (месту в программе, где ошибка совершена).

В простейшем случае место проявления симптома и ошибочный фрагмент совпадают. Но чаще всего они далеко отстоят друг от друга.

Цель отладки — найти оператор программы, при исполнении которого правильные аргументы приводят к неправильным результатам. Если место проявления симптома ошибки не является искомой аномалией, то один из аргументов оператора

должен быть неверным. Поэтому надо перейти к исследованию предыдущего оператора, выработавшего этот неверный аргумент. В итоге пошаговое обратное прослеживание приводит к искомому ошибочному месту.

В разных методах прослеживание организуется по-разному. В аналитических методах — на основе логических заключений о поведении программы. Цель — шаг за шагом уменьшать область программы, подозреваемой в наличии ошибки. Здесь определяется корреляция между значениями выходных данных и особенностями поведения.

Основное *преимущество аналитических методов отладки* состоит в том, что исходная программа остается без изменений.

В экспериментальных методах для прослеживания выполняется:

1) выдача значений переменных в указанных точках;
2) трассировка переменных (выдача их значений при каждом изменении);
3) трассировка потоков управления (имен вызываемых процедур, меток, на которые передается управление, номеров операторов перехода).

Преимущество экспериментальных методов отладки состоит в том, что основная рутинная работа по анализу процесса вычислений перекладывается на компьютер. Многие трансляторы имеют встроенные средства отладки для получения информации о ходе выполнения программы.

Недостаток экспериментальных методов отладки — в программу вносятся изменения, при исключении которых могут появиться ошибки. Впрочем, некоторые системы программирования создают специальный отладочный экземпляр программы, а в основной экземпляр не вмешиваются.

Контрольные вопросы

1. Поясните суть методики тестирования программной системы.
2. Когда и зачем выполняется тестирование элементов? Какой этап разработки оно проверяет?
3. Когда и зачем выполняется тестирование интеграции? Какой этап разработки оно проверяет?
4. Когда и зачем выполняется тестирование правильности? Какой этап разработки оно проверяет?
5. Когда и зачем выполняется системное тестирование? Какой этап разработки оно проверяет?
6. Поясните суть тестирования элементов.
7. Перечислите наиболее общие ошибки вычислений.
8. Перечислите источники ошибок сравнения и неправильных потоков управления.
9. На какие ситуации ориентировано тестирование путей обработки ошибок?
10. Что такое драйвер тестирования?
11. Что такое заглушка?
12. Поясните порядок работы драйвера тестирования.

13. В чем цель тестирования интеграции?
14. Какие категории ошибок интерфейса вы знаете?
15. В чем суть нисходящего тестирования интеграции?
16. Поясните шаги процесса нисходящей интеграции.
17. Поясните достоинства и недостатки нисходящей интеграции.
18. Какие категории заглушек вы знаете?
19. В чем суть восходящего тестирования интеграции?
20. Поясните шаги процесса восходящей интеграции.
21. Поясните достоинства и недостатки восходящей интеграции.
22. Какие категории драйверов вы знаете?
23. Какова комбинированная стратегия интеграции?
24. Каковы признаки критического модуля?
25. Что такое регрессионное тестирование?
26. В чем суть тестирования правильности?
27. Какие элементы включает минимальная конфигурация программной системы?
28. Что такое альфа-тестирование?
29. Что такое бета-тестирование?
30. В чем суть системного тестирования?
31. Как защищаться от проблемы «указание причины»?
32. В чем суть тестирования восстановления?
33. В чем суть тестирования безопасности?
34. В чем суть стрессового тестирования?
35. В чем суть тестирования производительности?
36. Что такое отладка?
37. Какие способы проявления ошибок вы знаете?
38. Какие симптомы ошибки вы знаете?
39. В чем суть аналитических методов отладки?
40. Поясните достоинства и недостатки аналитических методов отладки.
41. В чем суть экспериментальных методов отладки?
42. Поясните достоинства и недостатки экспериментальных методов отладки.

Глава 17

Объектно-ориентированное тестирование

Необходимость и важность тестирования ПО трудно переоценить. Вместе с тем следует отметить, что тестирование является сложной и трудоемкой деятельностью. Об этом свидетельствует содержание 14, 15 и 16 глав, в которых описывались классические основы тестирования, разработанные в расчете (в основном) на процедурное ПО. В этой главе рассматриваются вопросы объектно-ориентированного тестирования [33, 34, 64, 74, 75]. Существует мнение, что объектно-ориентированное тестирование мало чем отличается от процедурно-ориентированного тестирования. Конечно, многие понятия, подходы и способы тестирования у них общие, но в целом это мнение ошибочно. Напротив, особенности объектно-ориентированных систем должны вносить и вносят существенные изменения как в последовательность этапов, так и в содержание этапов тестирования. Сгруппируем эти изменения по трем направлениям:

- расширение области применения тестирования;
- изменение методики тестирования;
- учет особенностей объектно-ориентированного ПО при проектировании тестовых вариантов.

Обсудим каждое из выделенных направлений отдельно.

Расширение области применения объектно-ориентированного тестирования

Разработка объектно-ориентированного ПО начинается с создания визуальных моделей, отражающих статические и динамические характеристики будущей системы. Вначале эти модели фиксируют исходные требования заказчика, затем формализуют реализацию этих требований путем выделения объектов, которые взаимодействуют друг с другом посредством передачи сообщений. Исследование моделей взаимодействия приводит к построению моделей классов и их отношений, составляющих основу логического представления системы. При переходе к физи-

ческому представлению строятся модели компоновки и размещения системы. На создание моделей приходится большая часть затрат объектно-ориентированного процесса разработки. Если к этому добавить, что цена устранения ошибки стремительно растет с каждой итерацией разработки, то совершенно логично требование тестировать объектно-ориентированные модели анализа и проектирования.

Критерии тестирования моделей: правильность, полнота, согласованность [34, 75].

О синтаксической правильности судят по правильности использования языка моделирования (например, UML). О семантической правильности судят по соответствию модели реальным проблемам. Для определения того, отражает ли модель реальный мир, она оценивается экспертами, имеющими знания и опыт в конкретной проблемной области. Эксперты анализируют содержание классов, наследование классов, выявляя пропуски и неоднозначности. Проверяется соответствие отношений классов реалиям физического мира.

О согласованности судят путем рассмотрения противоречий между элементами в модели. Несогласованная модель имеет представления в одной части, которые противоречат представлениям в других частях модели.

Для оценки согласованности нужно исследовать каждый класс и его взаимодействия с другими классами. Для упрощения такого исследования удобно использовать модель Класс — Обязанность — Сотрудничество *CRC* (Class — Responsibility — Collaboration). Основной элемент этой модели *CRC*-карта [24, 102]. Кстати, *CRC*-карты — любимый инструмент проектирования в XP-процессах.

По сути *CRC*-карта — это расчерченная карточка размером 6 на 10 сантиметров. Она помогает установить задачи класса и выявить его окружение (классы-собеседники). Для каждого класса создается отдельная карта.

В каждой *CRC*-карте указывается имя класса, его обязанности (операции) и его сотрудничества (другие классы, в которые он посылает сообщения и от которых он зависит при выполнении своих обязанностей). Сотрудничества подразумевают наличие ассоциаций и зависимостей между классами. Они фиксируются в других моделях — диаграмме коммуникации (последовательности) обобщенных объектов и диаграмме классов.

CRC-карта намеренно сделана простой, даже ее ограниченный размер имеет определенный смысл: если список обязанностей и сотрудников не помещается на карте, то, наверное, данный класс надо разделить на несколько классов.

Для оценки модели (диаграммы) классов на основе *CRC*-карт рекомендуются следующие шаги [74].

1. Выполняется перекрестный просмотр *CRC*-карты и диаграммы коммуникации (последовательности) обобщенных объектов. Цель — проверить наличие сотрудников, согласованность информации в обеих моделях.
2. Исследуются обязанности *CRC*-карты. Цель — определить, предусмотрена ли в карте сотрудника обязанность, которая делегируется ему из данной карты. Например, в табл. 17.1 показана *CRC*-карта Банкомат. Для этой карты выясняем, выполняется ли обязанность чтение Карты Клиента, которая требует использования сотрудника Карта Клиента. Это означает, что класс Карта Клиента должен иметь операцию, которая позволяет ему быть прочитанным.

Таблица 17.1. CRC-карта Банкомат

Имя Класса: Банкомат	
Обязанности	**Сотрудники**
Чтение Карты Клиента	Карта Клиента
Идентификация клиента	База данных клиентов
Проверка счета	База данных счетов
Выдача денег	Блок денег
Выдача квитанции	Блок квитанций
Захват карты	Блок карт

3. Организуется проход по каждому соединению *CRC*-карты. Проверяется корректность запросов, выполняемых через соединения. Такая проверка гарантирует, что каждый сотрудник, предоставляющий услугу, получает обоснованный запрос. Например, если произошла ошибка и класс База данных клиентов получает от класса Банкомат запрос на состояние Счета, он не сможет его выполнить — ведь База данных клиентов не знает состояния их счетов.
4. Определяется, требуются ли другие классы и правильно ли распределены обязанности по классам? Для этого используют проходы по соединениям, исследованные на шаге 3.
5. Определяется, нужно ли объединять часто запрашиваемые обязанности. Например, в любой ситуации используют пару обязанностей — чтение Карты Клиента и идентификация клиента. Их можно объединить в новую обязанность проверка клиента, которая подразумевает как чтение его карты, так и идентификацию клиента.
6. Шаги 1–5 применяются итеративно, к каждому классу и на каждом шаге эволюции объектно-ориентированной модели.

Изменение методики при объектно-ориентированном тестировании

В классической методике тестирования действия начинаются с тестирования элементов, а заканчиваются тестированием системы. Вначале тестируют модули, затем тестируют интеграцию модулей, проверяют правильность реализации требований, после чего тестируют взаимодействие всех блоков компьютерной системы.

Особенности тестирования объектно-ориентированных «модулей»

При рассмотрении объектно-ориентированного ПО меняется понятие модуля. Наименьшим тестируемым элементом теперь является класс (объект). Класс содержит несколько операций и атрибутов. Поэтому сильно изменяется содержание тестирования модулей.

В данном случае нельзя тестировать отдельную операцию изолированно, как это принято в стандартном подходе к тестированию модулей. Любую операцию приходится рассматривать как часть класса.

Например, представим иерархию классов, в которой операция op() определена для суперкласса и наследуется несколькими подклассами. Каждый подкласс использует операцию op(), но она применяется в контексте его приватных атрибутов и операций. Этот контекст меняется, поэтому операцию op() надо тестировать в контексте каждого подкласса. Таким образом, изолированное тестирование операции op(), являющееся традиционным подходом в тестировании модулей, не имеет смысла в объектно-ориентированной среде.

Выводы:

- тестированию модулей традиционного ПО соответствует тестирование классов объектно-ориентированного ПО;
- тестирование традиционных модулей ориентировано на поток управления внутри модуля и поток данных через интерфейс модуля;
- тестирование классов ориентировано на операции, инкапсулированные в классе, и состояния в пространстве поведения класса.

Тестирование объектно-ориентированной интеграции

Объектно-ориентированное ПО не имеет иерархической управляющей структуры, поэтому здесь неприменимы методики как восходящей, так и нисходящей интеграции. Мало того, классический прием интеграции (добавление по одной операции в класс) зачастую не осуществим.

Р. Байндер предлагает две методики интеграции объектно-ориентированных систем [32]:

- тестирование, основанное на потоках;
- тестирование, основанное на использовании.

В первой методике объектом интеграции является набор классов, обслуживающий единичный ввод данных в систему. Иными словами, средства обслуживания каждого потока интегрируются и тестируются отдельно. Для проверки отсутствия побочных эффектов применяют регрессионное тестирование.

По второй методике вначале интегрируются и тестируются независимые классы. Далее работают с первым слоем зависимых классов (которые используют независимые классы), со вторым слоем и т. д. В отличие от стандартной интеграции, везде, где возможно, избегают драйверов и заглушек.

Д. МакГрегор считает, что одним из шагов объектно-ориентированного тестирования интеграции должно быть кластерное тестирование [74]. Кластер сотрудничающих классов определяется исследованием *CRC*-модели или диаграммы коммуникации объектов. Тестовые варианты для кластера ориентированы на обнаружение ошибок сотрудничества.

Объектно-ориентированное тестирование правильности

При проверке правильности исчезают подробности отношений классов. Как и традиционное подтверждение правильности, подтверждение правильности объектно-ориентированного ПО ориентировано на видимые действия пользователя и распознаваемые пользователем выводы из системы.

Для упрощения разработки тестов используются элементы Use Case, являющиеся частью модели требований. Каждый элемент Use Case задает сценарий, который позволяет обнаруживать ошибки во взаимодействии пользователя с системой.

Для подтверждения правильности может проводиться обычное тестирование «черного ящика».

Полезную для формирования тестов правильности информацию содержат диаграммы взаимодействия, диаграммы деятельности, а также диаграммы конечных автоматов.

Проектирование объектно-ориентированных тестовых вариантов

Традиционные тестовые варианты ориентированы на проверку последовательности: ввод исходных данных — обработка — вывод результатов, или на проверку внутренней управляющей (информационной) структуры отдельных модулей. Объектно-ориентированные тестовые варианты проверяют состояния классов. Получение информации о состоянии затрудняют такие объектно-ориентированные характеристики, как инкапсуляция, полиморфизм и наследование.

Инкапсуляция

Информацию о состоянии класса можно получить только с помощью встроенных в него операций, которые возвращают значения атрибутов класса.

Полиморфизм

При вызове полиморфной операции трудно определить, какая реализация будет проверяться. Пусть нужно проверить вызов функции:

```
y = functionA(x).
```

В стандартном ПО достаточно рассмотреть одну реализацию поведения, которая обеспечивает вычисление функции. В объектно-ориентированном ПО придется рассматривать поведение реализации Базовый_класс::functionA(x), Производный_класс::functionA(x), Наследник_Производного_класса::functionA(x). Здесь двойным двоеточием от имени операции отделяется имя класса, в котором размещена операция (это обозначение UML). Таким образом, в объектно-ориентированном контексте каждый раз при вызове functionA(x) следует рассматривать набор различных поведений. Конечно, подход к тестированию базовых и производных классов в основном одинаков. Разница состоит только в учете используемых системных ресурсов.

Наследование

Наследование также может усложнить проектирование тестовых вариантов. Пусть Родительский_класс содержит операции унаследована() и переопределена(). Дочерний_класс переопределяет операцию переопределена() по-своему. Очевидно, что реализация Дочерний_класс::переопределена() должна повторно тестироваться — ведь ее содержание изменилось. Но надо ли повторно тестировать операцию Дочерний _класс::унаследована()?

Рассмотрим следующий случай. Положим, что операция Дочерний_класс::унаследована() вызывает операцию переопределена(). К чему это приводит? Поскольку реализация операции переопределена() изменена, операция Дочерний_класс::унаследована() может не соответствовать этой новой реализации. Поэтому нужны новые тесты, хотя содержание операции унаследована() не изменено.

Важно отметить, что для операции Дочерний_класс::унаследована() может проводиться только подмножество тестов. Если часть операции унаследована() не зависит от операции переопределена(), то есть нет ее вызова или вызова любого кода, который косвенно ее вызывает, то ее не надо повторно тестировать в дочернем классе.

Родительский_класс::переопределена() и Дочерний_класс::переопределена() — это две разные операции с различными спецификациями и реализациями. Каждая из них проверяется самостоятельным набором тестов. Эти тесты нацелены на вероятные ошибки: ошибки интеграции, ошибки условий, граничные ошибки и т. д. Однако сами операции, как правило, похожи. Наборы их тестов будут перекрываться. Чем лучше качество объектно-ориентированного проектирования, тем больше перекрытие. Таким образом, новые тесты надо формировать только для тех требований к операции Дочерний_класс::переопределена(), которые не покрываются тестами для операции Родительский_класс::переопределена().

Выводы:

- тесты для операции Родительский_класс::переопределена() частично применимы к операции Дочерний_класс::переопределена();
- входные данные тестов подходят к операциям обоих классов;
- ожидаемые результаты для операции Родительского_класса отличаются от ожидаемых результатов для операции Дочернего_Класса.

К операциям класса применимы классические способы тестирования «белого ящика», которые гарантируют проверку каждого оператора и их управляющих связей. При большом количестве операций от тестирования по принципу «белого ящика» приходится отказываться. Меньших затрат потребует тестирование на уровне классов.

Способы тестирования «черного ящика» также применимы к объектно-ориентированным системам. Полезную входную информацию для тестирования «черного ящика» и тестирования состояний обеспечивают элементы Use Case.

Тестирование, основанное на ошибках

Цель тестирования, основанного на ошибках, — проектирование тестов, ориентированных на обнаружение предполагаемых ошибок [71]. Разработчик выдвигает гипотезу о предполагаемых ошибках. Для проверки его предположений разрабатываются тестовые варианты.

В качестве примера рассмотрим булево выражение

```
if (X and not Y or Z).
```

Инженер по тестированию строит гипотезу о предполагаемых ошибках выражения:

- операция not сдвинулась влево от нужного места (она должна применяться к Z);
- вместо and должно быть or;
- вокруг not Y or Z должны быть круглые скобки.

Для каждой предполагаемой ошибки проектируются тестовые варианты, которые заставляют некорректное выражение отказать.

Обсудим первую предполагаемую ошибку. Значения (X=false, Y=false, Z=false) устанавливают указанное выражение в false. Если вместо not Y должно быть not Z,

то выражение принимает неправильное значение, которое приводит к неправильному ветвлению. Аналогичные рассуждения применяются к генерации тестов по двум другим ошибкам.

Эффективность этой методики зависит от того, насколько правильно инженер выделяет предполагаемую ошибку. Если действительные ошибки объектно-ориентированной системы не воспринимаются как предполагаемые, то этот подход не лучше стохастического тестирования. Если же визуальные модели анализа и проектирования обеспечивают понимание ошибочных действий, то тестирование, основанное на ошибках, поможет найти подавляющее большинство ошибок при малых затратах.

При тестировании интеграции предполагаемые ошибки ищутся в пересылаемых сообщениях. В этом случае рассматривают три типа ошибок: неожиданный результат, вызов не той операции, некорректный вызов. Для определения предполагаемых ошибок (при вызове операций) нужно исследовать процесс выполнения операции.

Тестирование интеграции применяется как к атрибутам, так и к операциям. Поведение объекта определяется значениями, которые получают его атрибуты. Поэтому проверка значений атрибутов обеспечивает проверку правильности поведения.

При тестировании интеграции ищутся ошибки в объектах-клиентах, запрашивающих услуги, а не в объектах-серверах, предоставляющих эти услуги. Тестирование интеграции определяет ошибки в вызывающем коде, а не в вызываемом коде. Вызов операции используют как средство, обеспечивающее тестирование вызывающего кода.

Тестирование, основанное на сценариях

Тестирование, основанное на ошибках, оставляет в стороне два важных типа ошибок:

- некорректные спецификации;
- взаимодействия между подсистемами.

Ошибка из-за неправильной спецификации означает, что продукт не выполняет то, что хочет заказчик. Он делает что-то неправильно или в нем пропущена важная функциональная возможность. В любом случае страдает качество — соответствие требованиям заказчика.

Ошибки, связанные с взаимодействием подсистем, происходят, когда поведение одной подсистемы создает предпосылки для отказа другой подсистемы.

Тестирование, основанное на сценариях, ориентировано на действия пользователя, а не на действия программной системы [71]. Это означает фиксацию задач, которые выполняет пользователь, а затем применение их в качестве тестовых вариантов. Задачи пользователя фиксируются с помощью элементов Use Case.

Сценарии элементов Use Case обнаруживают ошибки взаимодействия, каждая из которых может быть следствием многих причин. Поэтому соответствующие тестовые варианты более сложны и лучше отражают реальные проблемы, чем тесты, основанные на ошибках. С помощью одного теста, основанного на сценарии, проверяется множество подсистем.

Рассмотрим, например, проектирование тестов для текстового редактора, основанных на сценариях.

Рабочие сценарии опишем в виде спецификаций элементов Use Case.

Элемент Use Case: Исправлять черновик.

Предпосылки: Обычно печатают черновик, читают его и обнаруживают ошибки, которые не видны на экране. Этот элемент Use Case описывает события, которые при этом происходят.

1. Печатать весь документ.
2. Прочитать документ, изменить определенные страницы.
3. После внесения изменения страница перепечатывается.
4. Иногда перепечатываются несколько страниц.

Этот сценарий определяет как требования тестов, так и требования пользователя. Требования пользователя очевидны, ему нужны:

- метод для печати отдельной страницы;
- метод для печати диапазона страниц.

В ходе тестирования проверяется редакция текста как до печати, так и после печати. Разработчик теста может надеяться обнаружить, что функция печати вызывает ошибки в функции редактирования. Это будет означать, что в действительности две программные функции зависят друг от друга.

Элемент Use Case: Печатать новую копию.

Предпосылки: Кто-то просит пользователя напечатать копию документа.

1. Открыть документ.
2. Напечатать документ.
3. Закрыть документ.

И в этом случае подход к тестированию почти очевиден. За исключением того, что не определено, откуда появился документ. Он был создан в ранней задаче. Означает ли это, что только эта задача влияет на сценарий?

Во многих современных редакторах запоминаются данные о последней печати документа. По умолчанию эту печать можно повторить. После сценария Исправлять черновик достаточно выбрать в меню Печать, а затем нажать кнопку Печать в диалоговом окне — в результате повторяется печать последней исправленной страницы. Таким образом, откорректированный сценарий примет вид:

Элемент Use Case: Печатать новую копию.

1. Открыть документ.
2. Выбрать в меню пункт Печать.
3. Проверить, что печаталось, и если печатался диапазон страниц, то выбрать опцию Печатать целый документ.
4. Нажать кнопку Печать.
5. Закрыть документ.

Этот сценарий указывает возможную ошибку спецификации: редактор не делает того, что пользователь ожидает от него. Заказчики часто забывают о проверке, предусмотренной шагом 3. Они раздражаются, когда рысью бегут к принтеру и находят одну страницу вместо ожидаемых 100 страниц. Раздраженные заказчики считают, что в спецификации допущена ошибка.

Разработчик может опустить эту зависимость в тестовом варианте, но, вероятно, проблема обнаружится в ходе тестирования. И тогда разработчик будет выкрикивать: «Я предполагал, я предполагал это учесть!!!»

Тестирование поверхностной и глубинной структуры

Поверхностная структура — это видимая извне структура объектно-ориентированной системы. Она отражает взгляд пользователя, который видит не функции, а объекты для обработки. Тестирование поверхностной структуры основывается на задачах пользователя. Главное — выяснить задачи пользователя. Для разработчика это нетривиальная проблема, поскольку требует отказа от своей точки зрения.

Глубинная структура отражает внутренние технические подробности объектно-ориентированной системы (на уровне проектных моделей и программного текста). Тесты глубинной структуры исследуют зависимости, поведение и механизмы взаимодействия, которые создаются в ходе проектирования подсистем и объектов.

В качестве базиса для тестирования глубинной структуры используются модели анализа и проектирования. Например, разработчик исследует диаграмму взаимодействия (невидимую извне) и спрашивает: «Проверяет ли тест взаимодействие, отмеченное на диаграмме?»

Диаграммы классов обеспечивают понимание структуры наследования, которая используется в тестах, основанных на ошибках. Рассмотрим операцию обработать (ссылка_на_РодительскийКласс). Что произойдет, если в вызове этой операции указать ссылку на дочерний класс? Есть ли различия в поведении, которые должны отражаться в операции обработать()? Эти вопросы инициируют создание конкретных тестов.

Способы тестирования содержания класса

Описываемые ниже способы ориентированы на отдельный класс и операции, которые инкапсулированы классом.

Стохастическое тестирование класса

При стохастическом тестировании исходные данные для тестовых вариантов генерируются случайным образом. Обсудим методику, предложенную С. Кирани и В. Тсай [66].

Рассмотрим класс Счет, который имеет следующие операции: открыть(), установить(), положить(), снять(), остаток(), итог(), ограничитьКредит(), закрыть().

Каждая из этих операций применяется при определенных ограничениях:

- счет должен быть открыт перед применением других операций;
- счет должен быть закрыт после завершения всех операций.

Даже с этими ограничениями существует множество допустимых перестановок операций. Минимальная работа экземпляра Счета включает следующую последовательность операций:

открыть() ▸ установить() ▸ положить() ▸ снять() ▸ закрыть().

Здесь стрелка обозначает отношение следования. Иначе говоря, здесь записано, что экземпляр Счета сначала выполняет операцию открытия, затем установки и т. д. Эта последовательность является минимальным тестовым вариантом для Счета. Впрочем, в эту последовательность можно встроить группировку, обеспечивающую создание других вариантов поведения:

открыть() ▸ установить() ▸ положить() ▸
▸ [остаток()•снять()•итог()•ограничитьКредит()•положить()]^n ▸ снять() ▸ закрыть().

Здесь приняты дополнительные обозначения: точка означает композицию операций по принципу логического И/ИЛИ, пара квадратных скобок — группировку, а показатель степени — количество повторений группировки.

Набор различных последовательностей может генерироваться случайным образом:

Тестовый вариант N:

открыть() ▸ установить() ▸ положить() ▸ остаток() ▸ снять() ▸
▸ итог() ▸ снять() ▸ закрыть().

Тестовый вариант M:

открыть() ▸ установить() ▸ положить() ▸ итог() ▸ ограничитьКредит() ▸
▸ снять() ▸ остаток() ▸ снять() ▸ закрыть().

Эти и другие тесты случайных последовательностей проводятся для проверки различных вариантов жизни объектов.

Тестирование разбиений на уровне классов

Тестирование разбиений уменьшает количество тестовых вариантов, требуемых для проверки классов (тем же способом, что и разбиение по эквивалентности для стандартного ПО). Области ввода и вывода разбивают на категории, а тестовые варианты разрабатываются для проверки каждой категории.

Обычно используют одну из трех категорий разбиения [66]. Категории образуются операциями класса.

Первый способ — *разбиение на категории по состояниям*. Основывается на способности операций изменять состояние класса. Обратимся к классу Счет. Операции снять(), положить() изменяют его состояние и образуют первую категорию. Операции остаток(), итог(), ограничитьКредит() не меняют состояние Счета и образуют вторую категорию. Проектируемые тесты отдельно проверяют операции, которые изменяют состояние, а также те операции, которые не изменяют состояние. Таким образом, для нашего примера:

Тестовый вариант 1:

открыть() ▸ установить() ▸ положить() ▸ положить() ▸ снять() ▸ снять() ▸ закрыть().

Тестовый вариант 2:

открыть() ▸ установить() ▸ положить() ▸ остаток() ▸ итог() ▸ ограничитьКредит() ▸
▸ снять() ▸ закрыть().

ТВ1 изменяет состояние объекта, в то время как ТВ2 проверяет операции, которые не меняют состояние. Правда, в ТВ2 пришлось включить операции минимальной тестовой последовательности, поэтому для нейтрализации влияния операций снять() и положить() их аргументы должны иметь одинаковые значения.

Второй способ — *разбиение на категории по атрибутам*. Основывается на атрибутах, которые используются операциями. В классе Счет для определения разбиений можно использовать атрибуты остаток и ограничение кредита. Например, на основе атрибута ограничение кредита операции подразделяются на три категории:

1) операции, которые используют ограничение кредита;
2) операции, которые изменяют ограничение кредита;
3) операции, которые не используют и не изменяют ограничение кредита.

Для каждой категории создается тестовая последовательность.

Третий способ – *разбиение на категории по функциональности*. Основывается на общности функций, которые выполняют операции. Например, операции в классе Счет могут быть разбиты на категории:

- операции инициализации (открыть(), установить());
- вычислительные операции (положить(), снять());
- запросы (остаток(), итог(), ограничитьКредит());
- операции завершения (закрыть()).

Способы тестирования взаимодействия классов

Для тестирования сотрудничества классов могут использоваться различные способы [66]:

- стохастическое тестирование;
- тестирование разбиений;
- тестирование на основе сценариев;
- тестирование на основе состояний.

В качестве примера рассмотрим программную модель банковской системы, в состав которой входят классы Банк, Банкомат, ИнтерфейсБанкомата, Счет, Работа с Наличными, ПодтверждениеПравильности, имеющие следующие операции:

Банк:

проверитьСчет();	запросДепозита ();	разрешитьКарту();
проверитьPIN();	инфоСчета();	снятьРазрешен();
проверитьПолис();	открытьСчет();	закрытьСчет().
запросСнятия();	начальнДепозит();	

Банкомат:

картаВставлена();	положить();	состояниеСчета();
пароль();	снять();	завершить().

ИнтерфейсБанкомата:

проверитьСостояние();	выдатьНаличные();	читатьИнфоКарты();
состояниеПоложить();	печатьСостСчета();	получитьКолвоНалич().

Счет:

ограничКредит();	остаток();	положить();
типСчета();	снять();	закрыть().

ПодтверждениеПравильности:

подтвPIN();	подтвСчет().

Диаграмма коммуникации объектов банковской системы представлена на рис. 17.1. На этой диаграмме отображены связи между обобщенными объектами, стрелки передачи сообщений подписаны именами вызываемых операций.

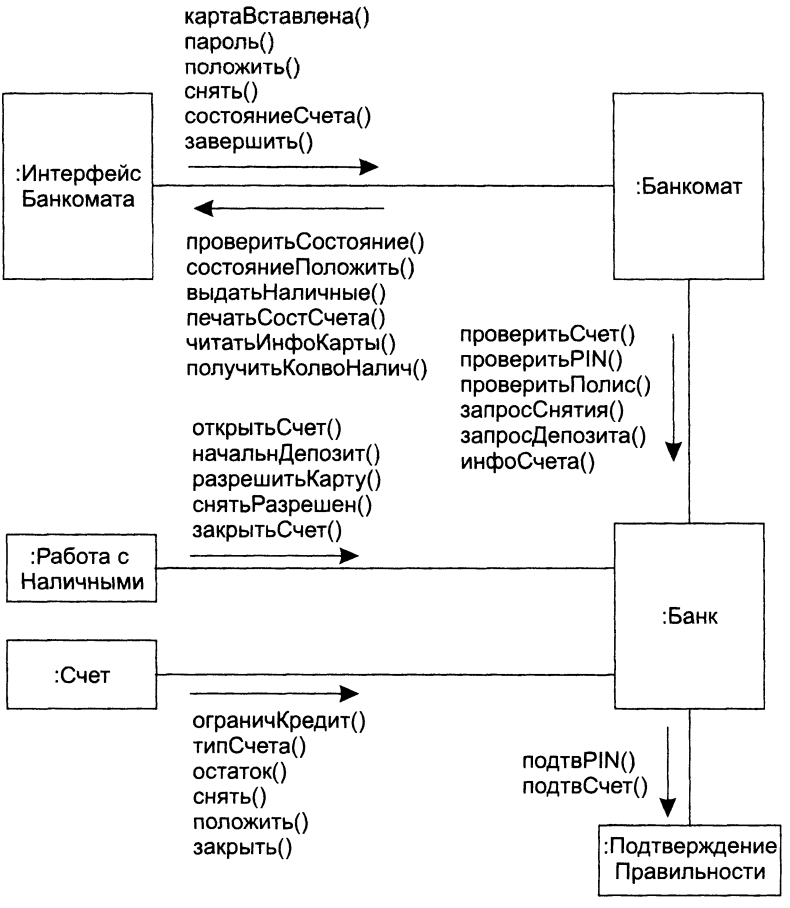


Рис. 17.1. Диаграмма коммуникации объектов банковской системы

Стохастическое тестирование

Стохастические тестовые варианты генерируются следующей последовательностью шагов:

1. Для создания тестов используют списки операций каждого класса-клиента. Операции будут посылать сообщения в классы-серверы.
2. Для каждого созданного сообщения определяется класс-сотрудник и соответствующая операция в классе-сервере.
3. Для каждой операции в классе-сервере, которая вызывается сообщением из класса-клиента, определяются сообщения, которые она, в свою очередь, посылает.
4. Для каждого из сообщений определяется следующий уровень вызываемых операций; они вставляются в тестовую последовательность.

В качестве примера приведем последовательность операций для класса Банк, вызываемых классом Банкомат:

проверитьСчет() ▸ проверитьPIN() ▸ [[проверитьПолис() ▸
▸ запросСнятия()]•запросДепозита()•инфоСчета()]^n.

ПРИМЕЧАНИЕ

Здесь приняты следующие обозначения: стрелка означает отношение следования, точка — композицию операций по принципу логического И/ИЛИ, пара квадратных скобок — группировку операций классов, показатель степени — количество повторений группировки из операций классов.

Случайный тестовый вариант для класса Банк может иметь вид:

Тестовый Вариант N: проверитьСчет() ▸ проверитьPIN() ▸ запросДепозита().

Для выявления сотрудников, включенных в этот тест, рассматриваются сообщения, связанные с каждой операцией, записанной в *ТВ N*. Для выполнения заданий проверитьСчет() и проверитьPIN() банк должен сотрудничать с классом ПодтверждениеПравильности. Для выполнения задания запросДепозита() банк должен сотрудничать с классом Счет. Отсюда новый *ТВ*, который проверяет отмеченные сотрудничества:

Тестовый Вариант M: проверитьСчет()$_{Банк}$ ▸ {подтвСчет()$_{ПодтвПрав}$} ▸
▸ проверитьPIN()$_{Банк}$ ▸ {подтвPIN()$_{ПодтвПрав}$} ▸ запросДепозита()$_{Банк}$ ▸ {положить()$_{Счет}$}.

В этой последовательности операции классов-сотрудников банка помещены в фигурные скобки, индексы отображают принадлежность операций к конкретным классам.

Тестирование разбиений

В основу этого метода положен тот же подход, который применялся к отдельному классу. Отличие в том, что тестовая последовательность расширяется для включения тех операций, которые вызываются с помощью сообщений для сотрудничающих классов.

Другой подход к тестированию разбиений основан на взаимодействиях с конкретным классом. Как показано на рис. 17.1, Банк получает сообщения от Банкомата и класса Работа с Наличными. Поэтому операции внутри Банка тестируются разбиением их на те, которые обслуживают класс Банкомат, и на те, которые обслуживают класс Работа с Наличными. Для дальнейшего уточнения может быть использовано разбиение на категории по состояниям.

Тестирование на основе состояний

В качестве источника исходной информации используют диаграммы конечных автоматов, фиксирующие динамику поведения класса. Данный способ позволяет получить набор тестов, проверяющих поведение класса, и тех классов, которые сотрудничают с ним [66].

В качестве примера на рис. 17.2 показана диаграмма конечного автомата класса Счет.

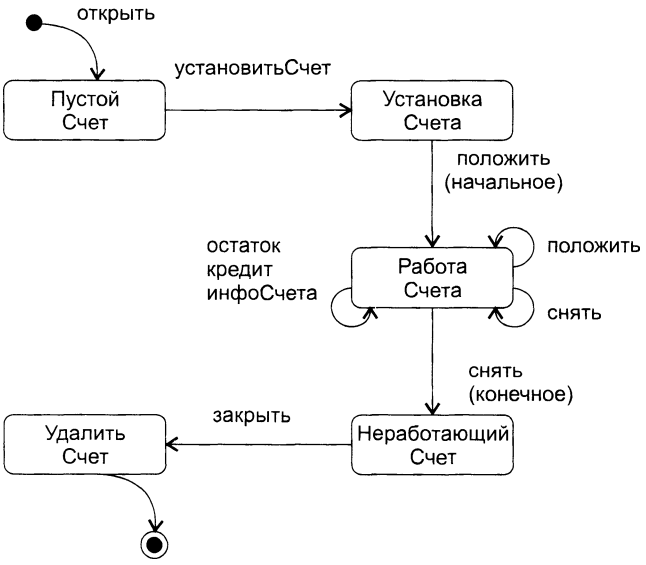


Рис. 17.2. Диаграмма конечного автомата класса Счет

Видим, что объект Счета начинает свою жизнь в состоянии Пустой Счет, а заканчивает жизнь — в состоянии Удалить Счет. Наибольшее количество событий (и действий) связано с состоянием Работа Счета. Для упрощения рисунка здесь принято, что имена событий совпадают с именами действий (поэтому действия не показаны).

Проектируемые тесты должны обеспечить покрытие всех состояний. Это значит, что тестовые варианты должны инициировать переходы через все состояния объекта:

Тестовый вариант 1:

открыть ▸ установитьСчет ▸ положить(начальное) ▸ снять(конечное) ▸ закрыть.

Отметим, что эта последовательность аналогична минимальной тестовой последовательности. Добавим к минимальной последовательности дополнительные тестовые варианты:

Тестовый вариант 2:

открыть ▸ установитьСчет ▸ положить(начальное) ▸ положить ▸ остаток ▸
▸ кредит ▸ снять(конечное) ▸ закрыть.

Тестовый вариант 3:

открыть ▸ установить ▸ положить(начальное) ▸ положить ▸ снять ▸ инфоСчета ▸
▸ снять(конечное) ▸ закрыть.

Для гарантии проверки всех вариантов поведения количество тестовых вариантов может быть увеличено. Когда поведение класса определяется в сотрудничестве с несколькими классами, для отслеживания «потока поведения» используют

набор диаграмм конечных автоматов, характеризующих смену состояний других классов.

Возможна другая методика исследования состояний — *«преимущественно в ширину»*. В этой методике:

- каждый тестовый вариант проверяет один новый переход;
- новый переход можно проверять, если полностью проверены все предшествующие переходы, то есть переходы между предыдущими состояниями.

Рассмотрим объект класса Карта Клиента (рис. 17.3). Начальное состояние карты Не определена, то есть не установлен номер карты. После чтения карты (в ходе диалога с банкоматом) объект переходит в состояние Определена. Это означает, что определены банковские идентификаторы номер Карты и дата Истечения Срока. Карта клиента переходит в состояние Предъявляется на рассмотрение, когда проводится ее авторизация, и в состояние Разрешена, когда авторизация подтверждается. Переход карты клиента из одного состояния в другое проверяется отдельным тестовым вариантом.

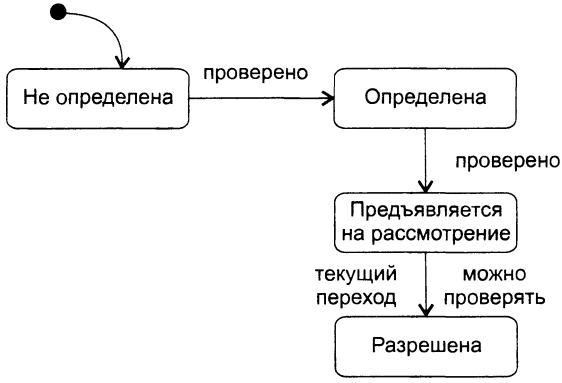


Рис. 17.3. Тестирование «преимущественно в ширину»

Подход «преимущественно в ширину» требует: нельзя проверять Разрешена перед проверкой Не определена, Определена и Предъявляется на рассмотрение. В противном случае нарушается условие этого подхода: перед тестированием текущего перехода должны быть протестированы все переходы, ведущие к нему.

Предваряющее тестирование и рефакторинг при экстремальной разработке

Предваряющее тестирование (test-first design) и рефакторинг — основной способ разработки при экстремальном программировании.

Роль предваряющего тестирования часто подчеркивают, употребляя термин «разработка через тестирование» (test-driven development).

Обычно рефакторингом называют внесение в код небольших изменений, сохраняющих функциональность и улучшающих структуру программы. Более широко

рефакторинг определяют как технику разработки ПО через множество изменений кода, направленных на добавление функциональности и улучшение структуры.

Предваряющее тестирование и рефакторинг задают такой порядок создания и последующего улучшения ПО, при котором сначала пишутся тесты, а затем программируется код, который будет подвергаться этим тестам. Программист выбирает задачу, затем пишет тестовые варианты, которые приводят к отказу программы, так как программа еще не выполняет данную задачу. Далее он модифицирует программу так, чтобы тесты проходили и задача выполнялась. Программист продолжает писать новые тестовые варианты и модифицировать программу (для их выполнения) до тех пор, пока программа не будет исполнять все свои обязанности. После этого программист небольшими шагами улучшает ее структуру (проводит рефакторинг), после каждого из шагов запускает все тесты, чтобы убедиться, что программа по-прежнему работает.

Для демонстрации такого подхода рассмотрим пример конкретной разработки. Будем опираться на технику, описанную Робертом Мартином [72].

Создадим программу для регистрации посещений кафе-кондитерской. Каждый раз, когда лакомка посещает кафе, вводится количество купленных булочек, их стоимость и текущий вес любителя (любительницы) сладостей. Система отслеживает эти значения и выдает отчеты. Программу будем писать на языке Java.

Для экстремального тестирования удобно использовать среду Junit, авторами которой являются Кент Бек и Эрик Гамма (Kent Beck и Erich Gamma). Прежде всего создадим среду для хранения тестов модулей. Это очень важно для предваряющего тестирования: вначале пишется тестовый вариант, а только потом — код программы. Необходимый код имеет следующий вид.

Листинг 17.1. ТестЛакомки.java

```
import junit.framework.*;
public class ТестЛакомки extends TestCase
{
    public ТестЛакомки (String name)
    {
        super(name);
    }
}
```

Видно, что при использовании среды Junit класс-контейнер тестовых вариантов должен быть наследником от класса TestCase. Кстати, условимся весь новый код выделять полужирным шрифтом.

Создадим первый тестовый вариант. Одна из целей создаваемой программы — запись количества посещений кафе. Поэтому должен существовать объект класса ПосещениеКафе, содержащий нужные данные. Следовательно, надо написать тест, создающий этот объект и опрашивающий его атрибуты. Тесты будем записывать как тестовые функции (их имена должны начинаться с префикса тест). Введем тестовую функцию тестСоздатьПосещениеКафе() (листинг 17.2).

Листинг 17.2. ТестЛакомки.java

```
import junit.framework.*;
public class ТестЛакомки extends TestCase
```

продолжение

Листинг 17.2 (продолжение)

```
{
    public ТестЛакомки (String name)
    {
        super(name);
    }
    public void тестСоздатьПосещениеКафе()
    {
        ПосещениеКафе v = new ПосещениеКафе();
    }
}
```

Для компиляции этого фрагмента подключим класс ПосещениеКафе.

Листинг 17.3. ТестЛакомки.java и ПосещениеКафе.java

```
ТестЛакомки.java
import junit.framework.*;
import ПосещениеКафе;
public class ТестЛакомки extends TestCase
{
    public ТестЛакомки (String name)
    {
        super(name);
    }
    public void тестСоздатьПосещениеКафе()
    {
        ПосещениеКафе v = new ПосещениеКафе();
    }
}
ПосещениеКафе.java
public class ПосещениеКафе
{
}
```

Этот код компилируется, тест проходит, и мы готовы добавить необходимую функциональность.

Листинг 17.4. ТестЛакомки.java и ПосещениеКафе.java

```
ТестЛакомки.java
import junit.framework.*;
import ПосещениеКафе;
import java.util.Date;
public class ТестЛакомки extends TestCase
{
    public ТестЛакомки(String name)
    {
        super(name);
    }
    public void тестСоздатьПосещениеКафе()
    {
        Date дата = new Date();
        double булочки = 7.0; // 7 булочек
        double стоимость = 12.5 * 7;
        // цена 1 булочки = 12.5 руб.
        double вес = 60.0; // взвешивание лакомки
```

```
        double дельта = 0.0001; // точность
        ПосещениеКафе v =
            new ПосещениеКафе(дата, булочки, стоимость, вес);
        assertEquals(дата, v.получитьДату());
        assertEquals(12.5 * 7, v.получитьСтоимость(), дельта);
        assertEquals(7.0, v.получитьБулочки(), дельта);
        assertEquals(60.0, v.получитьВес(), дельта);
        assertEquals(12.5, v.получитьЦену(), дельта);
    }
}
```

ПосещениеКафе.java

```
import java.util.Date;
public class ПосещениеКафе
{
    private Date егоДата;
    private double егоБулочки;
    private double егоСтоимость;
    private double егоВес;
    public ПосещениеКафе(Date дата, double булочки,
        double стоимость, double вес)
    {
        егоДата = дата;
        егоБулочки = булочки;
        егоСтоимость = стоимость;
        егоВес = вес;
    }
    public Date получитьДату() {return егоДата;}
    public double получитьБулочки() {return егоБулочки;}
    public double получитьСтоимость() {return егоСтоимость;}
    public double получитьЦену(){return егоСтоимость/егоБулочки;}
    public double получитьВес() {return егоВес;}
}
```

На этом шаге мы добавили тесты в класс ТестЛакомки, а также добавили методы в класс ПосещениеКафе. Унаследованные методы assertEquals() позволяют проводить сравнение ожидаемых и фактических результатов тестирования.

Очевидно, вы удивитесь этому подходу. Неужели нельзя вначале написать весь код класса ПосещениеКафе, а потом создать тесты? Ответ достаточно прост. Написание тестов перед написанием программного кода дает важное преимущество: мы знаем, что весь ранее созданный код компилируется и выполняется. Следовательно, любая ошибка вызывается текущими изменениями, а не более ранним кодом. И значимость этого преимущества усиливается по мере продвижения вперед.

Далее, определимся с хранением объектов класса ПосещениеКафе. Очевидно, что атрибут егоВес характеризует лакомку. Таким образом, объект класса ПосещениеКафе записывает часть состояния лакомки на момент посещения кафе. Следовательно, нужно создать объект класса Лакомка и содержать объекты класса ПосещениеКафе в нем.

Листинг 17.5. ТестЛакомки.java и Лакомка.java

ТестЛакомки.java

```
import junit.framework.*;
import ПосещениеКафе;
import java.util.Date
```

продолжение ⇙

Листинг 17.5 (продолжение)

```
public class ТестЛакомки extends TestCase
{
    public ТестЛакомки(String name)
    {
        super(name);
    }
    . . .
    public void тестСоздатьЛакомку()
    {
        Лакомка g = new Лакомка();
        assertEquals(0, g.получитьЧислоПосещений());
    }
}
Лакомка.java
public class Лакомка
{
    public int получитьЧислоПосещений()
    {
        return 0;
    }
}
```

Листинг 17.5 показывает начальный шаг. Мы написали новую тестовую функцию тестСоздатьЛакомку(). Эта функция создает объект класса Лакомка и затем убеждается, что хранимое количество посещений равно 0. Конечно, реализация метода получитьЧислоПосещений() неверна, но она обеспечивает прохождение теста. Это позволит нам в будущем выполнить рефакторинг (для улучшения решения).

Введем в класс Лакомку объект-контейнер, хранящий данные о разных посещениях (как элементы списка в массиве изменяемого размера). Для его создания используем класс-контейнер ArrayList из библиотеки Java 2. В будущем нам потребуются три метода контейнера: add() (добавить элемент в контейнер), get() (получить элемент из контейнера), size() (вернуть количество элементов в контейнере).

Листинг 17.6. Лакомка.java

```
import java.util.ArrayList;
public class Лакомка
{
    private ArrayList егоПосещения = new ArrayList();
// создание объекта егоПосещения – контейнера посещений
    public int получитьЧислоПосещений ()
    {
        return егоПосещения.size();
// возврат количества элементов в контейнере
// оно равно количеству посещений кафе
    }
}
```

Отметим, что после каждого изменения мы прогоняем все тесты, а не только функцию тестСоздатьЛакомку(). Это дает гарантию, что изменения не испортили уже работающий код.

На следующем шаге следует определить, как к Лакомке добавляется посещение кафе. Так будет выглядеть простейший тестовый вариант.

Листинг 17.7. ТестЛакомки.java

```
public void тестДобавитьПосещение()
{
    double булочки = 7.0; // 7 булочек
    double стоимость = 12.5 * 7; // цена 1 булочки = 12.5 руб.
    double вес = 60.0; // взвешивание лакомки
    double дельта = 0.0001; // точность
    Лакомка g = new Лакомка();
    g.добавитьПосещениеКафе(булочки, стоимость, вес);
    assertEquals(1, g.получитьЧислоПосещений());
}
```

В этом тесте объект класса ПосещениеКафе не создается. Очевидно, что создавать объект и добавлять его в список должен метод добавитьПосещениеКафе() класса Лакомка.

Листинг 17.8. Лакомка.java

```
public void добавитьПосещениеКафе(double булочки, double стоимость,
                                                    double вес)
{
    ПосещениеКафе v =
        new ПосещениеКафе(new Date(),булочки, стоимость, вес);
    егоПосещения.add(v);
// добавление эл-та v в контейнер посещений
}
```

Опять прогоняются все тесты. Анализ программного кода в функциях тестДобавитьПосещение() и тестСоздатьПосещениеКафе() показывает, что он частично дублируется. Обе функции создают одинаковые локальные переменные и инициализируют их одинаковыми значениями. Чтобы избавиться от дублирования, проведем рефакторинг тестируемой программы и сделаем локальные переменные атрибутами класса.

Листинг 17.9. ТестЛакомки.java

```
import junit.framework.*;
import ПосещениеКафе;
import java.util.Date;
public class ТестЛакомки extends TestCase
{
    private double булочки = 7.0; // 7 булочек
    private double стоимость = 12.5 * 7;
// цена 1 булочки = 12.5 р.
    private double вес = 60.0; // взвешивание лакомки
    private double дельта = 0.0001; // точность
    public ТестЛакомки(String name)
    {
        super(name);
    }
    public void тестСоздатьПосещениеКафе()
    {
        Date дата = new Date();
        ПосещениеКафе v = new ПосещениеКафе(дата, булочки,
                                                стоимость, вес);
```

продолжение

Листинг 17.9 (продолжение)

```
        assertEquals(date, v.получитьДату());
        assertEquals(12.5 * 7, v.получитьСтоимость(), дельта);
        assertEquals(7.0, v.получитьБулочки(), дельта);
        assertEquals(60.0, v.получитьВес(), дельта);
        assertEquals(12.5, v.получитьЦену(), дельта);
    }
    public void тестСоздатьЛакомку()
    {
        Лакомка g = new Лакомка ();
        assertEquals(0, g.получитьЧислоПосещений());
    }
    public void тестДобавитьПосещение()
    {
        Лакомка g = new Лакомка();
        g.добавитьПосещениеКафе(булочки, стоимость, вес);
        assertEquals(1, g.получитьЧислоПосещений());
    }
}
```

Еще раз подчеркнем: наличие тестов позволяет определить, что этот рефакторинг ничего не разрушил в программе. Мы будем убеждаться в этом преимуществе постоянно после очередного применения рефакторинга для реструктуризации программы. Каждый раз после внесения в код изменений запускаются тесты и проверяется работоспособность программы.

Очередная задача — после добавления к Лакомке объектов класса ПосещениеКафе у Лакомки можно запрашивать генерацию отчетов. Сначала напишем тесты, начнем с простейшего теста.

Листинг 17.10. ТестЛакомки.java

```
public void тестОтчетаОдногоПосещения()
{
    Лакомка g = new Лакомка();
    g.добавитьПосещениеКафе(булочки, стоимость, вес);
    Отчет r = g.создатьОтчет();
    assertEquals(0, r.получитьИзменениеВеса(), дельта);
    assertEquals(булочки, r.получитьПотреблениеБулочек(),
                                                        дельта);
    assertEquals(0, r.получитьВесНаБулочку(), дельта);
    assertEquals(стоимость, r.получитьСтоимостьБулочек(),
                                                        дельта);
}
```

При создании этого тестового варианта мы обдумали детали генерации отчета. Во-первых, Лакомка должна обладать методом создатьОтчет(). Во-вторых, этот метод должен возвращать объект класса с именем Отчет. В-третьих, Отчет должен иметь несколько методов-селекторов.

Значения, возвращаемые методами-селекторами, следует проанализировать. Для вычисления изменения веса (или приращения веса на одну булочку) одного посещения кафе недостаточно. Чтобы вычислить эти значения, необходимы, как минимум, два посещения. С другой стороны, одного визита достаточно, чтобы сосчитать потребление и стоимость булочек.

Разумеется, тестовый вариант не компилируется. Поэтому необходимо добавить соответствующие методы и классы. Сначала добавим код, обеспечивающий компиляцию, но не обеспечивающий выполнение тестов.

Листинг 17.11. Лакомка.java, ТестЛакомки.java и Отчет.java

```
Лакомка.java
public Отчет создатьОтчет()
{
    return new Отчет();
}
ТестЛакомки.java
public void тестОтчетаОдногоПосещения()
{
    Лакомка g = new Лакомка();
    g.добавитьПосещениеКафе(булочки, стоимость, вес);
    Отчет r = g.создатьОтчет();
    assertEquals(0, r.получитьИзменениеВеса(), дельта);
    assertEquals(булочки, r.получитьПотреблениеБулочек(),
                                                          дельта);
    assertEquals(0, r.получитьВесНаБулочку(), дельта);
    assertEquals(стоимость, r.получитьСтоимостьБулочек(),
                                                          дельта);
}
Отчет.java
public class Отчет
{
    public double получитьИзменениеВеса()
                                        {return егоИзменениеВеса;}
    public double получитьВесНаБулочку()
                                        {return егоВесНаБулочку;}
    public double получитьСтоимостьБулочек()
                                    {return егоСтоимостьБулочек;}
    public double получитьПотреблениеБулочек()
                                    {return егоПотреблениеБулочек;}
    private double егоИзменениеВеса;
    private double егоВесНаБулочку;
    private double егоСтоимостьБулочек;
    private double егоПотреблениеБулочек;
}
```

Код в листинге 17.11 компилируется и запускается, но его недостаточно для того, чтобы прошли тесты. Нужен рефакторинг кода. Для начала сделаем минимально возможные изменения.

Листинг 17.12. Лакомка.java и Отчет.java

```
Лакомка.java
public Отчет создатьОтчет()
{
    Отчет r = new Отчет();
    ПосещениеКафе v = (ПосещениеКафе) егоПосещения.get(0);
// занести в v первый элемент из контейнера посещений
    r.устВесНаБулочку(0);
    r.устИзменениеВеса(0);
    r.устСтоимостьБулочек(v.получитьСтоимость());
```

продолжение

Листинг 17.12 (продолжение)

```
    r.устПотреблениеБулочек(v.получитьБулочки());
    return r;
}
Отчет.java
public void устВесНаБулочку(double wpb)
                                          {егоВесНаБулочку = wpb;}
public void устИзменениеВеса(double kg)
                                          {егоИзменениеВеса = kg;}
public void устСтоимостьБулочек(double ct)
                                        {егоСтоимостьБулочек = ct;}
public void устПотреблениеБулочек (double b)
                                       {егоПотреблениеБулочек = b;}
```

Предполагаем, что Лакомке разрешено только одно посещение. В этой версии метода создатьОтчет() устанавливаются и возвращаются значения атрибутов Отчета.

Такой способ разработки метода создатьОтчет может показаться странным, ведь его реализация не завершена. Однако преимущество по-прежнему в том, что между каждой компиляцией и тестированием вносятся только контролируемые добавления. Если что-то отказывает, можно просто вернуться к предыдущей версии и начать сначала, необходимость в сложной отладке отсутствует.

Для завершения кода продумаем тесты для Лакомки без посещений и с несколькими посещениями кафе. Начнем с теста и кода для варианта без посещений.

Листинг 17.13. ТестЛакомки.java и Лакомка.java

```
ТестЛакомки.java
public void тестОтчетаБезПосещений()
{
    Лакомка g = new Лакомка();
    Отчет r = g.создатьОтчет();
    assertEquals(0, r.получитьИзменениеВеса(), дельта);
    assertEquals(0, r.получитьПотреблениеБулочек(), дельта);
    assertEquals(0, r.получитьВесНаБулочку()), дельта;
    assertEquals(0, r.получитьСтоимостьБулочек(), дельта);
}
Лакомка.java
public Отчет создатьОтчет()
{
    Отчет r = new Отчет();
    if (егоПосещения.size() == 0)
{
        r.устВесНаБулочку(0);
        r.устИзменениеВеса(0);
        r.устСтоимостьБулочек(0);
        r.устПотреблениеБулочек(0);
    }
    else
    {
        ПосещениеКафе v = (ПосещениеКафе) егоПосещения.get(0);
// занести в v первый элемент из контейнера посещений
        r.устВесНаБулочку(0);
        r.устИзменениеВеса(0);
        r.устСтоимостьБулочек(v.получитьСтоимость());
        r.устПотреблениеБулочек(v.получитьБулочки());
```

```
    }
    return r;
}
```

Теперь начнем создавать тестовый вариант для нескольких посещений.

Листинг 17.14. ТестЛакомки.java

```
public void тестОтчетаНесколькихПосещений()
{
    Лакомка g = new Лакомка();
    g.добавитьПосещениеКафе(7, 87.5, 60.7);
    g.добавитьПосещениеКафе(14, 175, 62.1);
    g.добавитьПосещениеКафе(28, 350, 64.9);
    Отчет r = g.создатьОтчет();
    assertEquals(4.2, r.получитьИзменениеВеса(), дельта);
    assertEquals(49, r.получитьПотреблениеБулочек(), дельта);
    assertEquals(0.086, r.получитьВесНаБулочку(), дельта);
    assertEquals(612.5, r.получитьСтоимостьБулочек(), дельта);
}
```

Мы установили число посещений для Лакомки равное трем. Предполагается, что цена булочки составляет 12,5 руб., а изменение веса — 0,1 кг на одну булочку. Таким образом, за 175 руб. лакомка покупает и съедает 14 булочек, полнея на 1,4 кг.

Но здесь какая-то ошибка. Скорость изменения веса должна определяться коэффициентом 0,1 кг на одну булочку. А если разделить 4,2 (изменение веса) на 49 (количество булочек), то получаем коэффициент 0,086. В чем причина несоответствия?

После размышлений становится понятно, что вес лакомки регистрируется на выходе из кафе. Поэтому приращение веса и потребление булочек во время первого посещения не учитывается. Изменим исходные данные теста.

Листинг 17.15. ТестЛакомки.java

```
public void тестОтчетаНесколькихПосещений()
{
    Лакомка g = new Лакомка();
    g.добавитьПосещениеКафе(7, 87.5, 60.7);
    g.добавитьПосещениеКафе(14, 175, 62.1);
    g.добавитьПосещениеКафе(28, 350, 64.9);
    Отчет r = g.создатьОтчет();
    assertEquals(4.2, r.получитьИзменениеВеса(), дельта);
    assertEquals(42, r.получитьПотреблениеБулочек(), дельта);
    assertEquals(0.1, r.получитьВесНаБулочку(), дельта);
    assertEquals(612.5, r.получитьСтоимостьБулочек(), дельта);
}
```

Этот тест корректен. Никогда не известно, с чем встретишься при написании тестов. Можно быть уверенным лишь в том, что, определяя понятия дважды (при написании тестов и кода), вы найдете больше ошибок, чем при простом написании кода.

Теперь добавим код, обеспечивающий прохождение теста из листинга 17.15.

Листинг 17.16. Лакомка.java

```
public Отчет создатьОтчет()
{
    Отчет r = new Отчет();
    if (егоПосещения.size() == 0)
    {
        r.устВесНаБулочку(0);
        r.устИзменениеВеса(0);
        r.устСтоимостьБулочек(0);
        r.устПотреблениеБулочек(0);
    }
    else if (егоПосещения.size() == 1)
    {
        ПосещениеКафе v = (ПосещениеКафе) егоПосещения.get(0);
// занести в v первый элемент из контейнера посещений
        r.устВесНаБулочку(0);
        r.устИзменениеВеса(0);
        r.устСтоимостьБулочек(v.получитьСтоимость());
        r.устПотреблениеБулочек(v.получитьБулочки());
    }
    else
    {
        double первыйЗамер = 0;
        double последнийЗамер = 0;
        double общаяСтоимость = 0;
        double потреблениеБулочек = 0;
        for (int i = 0; i < егоПосещения.size(); i++)
// проход по всем элементам контейнера посещений
        {
            ПосещениеКафе v = (ПосещениеКафе)
                                        егоПосещения.get(i);
// занести в v i-й элемент из контейнера посещений
            if (i == 0)
            {
                первыйЗамер = v.получитьВес();
// занести в первыйЗамер вес при 1-м посещении
                потреблениеБулочек -= v.получитьБулочки();
            }
            if (i== егоПосещения.size()- 1) последнийЗамер =
                                                    v.получитьВес();
// занести в последнийЗамер вес при послед. посещении
            общаяСтоимость += v.получитьСтоимость();
            потреблениеБулочек += v.получитьБулочки();
        }
        double изменение = последнийЗамер – первыйЗамер;
        r.устВесНаБулочку(изменение/потреблениеБулочек);
        r.устИзменениеВеса(изменение);
        r.устСтоимостьБулочек(общаяСтоимость);
        r.устПотреблениеБулочек(потреблениеБулочек);
    }
    return r;
}
```

Данный код из-за множества специальных случаев выглядит неуклюже. Для устранения специальных случаев нужно провести рефакторинг. Поскольку третий специальный случай наиболее универсален, надо убрать первые два случая.

Когда мы это сделаем, запуск тестового варианта тестОтчетаОдногоПосещения завершится неудачей. Причина в том, что обработка одного посещения включала булочки, купленные только во время первого посещения. А как мы уже обнаружили, если число посещений равно единице, то потребление булочек должно быть равно нулю. Поэтому исправим тестовый вариант и код.

Листинг 17.17. Лакомка.java

```
public Отчет создатьОтчет()
{
    Отчет r = new Отчет();
    double первыйЗамер = 0;
    double последнийЗамер = 0;
    double общаяСтоимость = 0;
    double потреблениеБулочек = 0;
    for (int i= 0; i< егоПосещения.size(); i++)
// проход по всем элементам контейнера посещений
    {
        ПосещениеКафе v = (ПосещениеКафе) егоПосещения.get(i);
// занести в v i-й элемент из контейнера посещений
        if (i == 0)
        {
            первыйЗамер = v.получитьВес();
//занести в первыйЗамер вес при 1-м посещении
            потреблениеБулочек -= v.получитьБулочки();
        }
        if (i== егоПосещения.size()- 1) последнийЗамер =
                                                v.получитьВес();
// занести в последнийЗамер вес при послед. посещении
        общаяСтоимость += v.получитьСтоимость();
        потреблениеБулочек += v.получитьБулочки();
    }
    double изменение = последнийЗамер – первыйЗамер;
    r.устВесНаБулочку(изменение/потреблениеБулочек);
    r.устИзменениеВеса(изменение);
    r.устСтоимостьБулочек(общаяСтоимость);
    r.устПотреблениеБулочек(потреблениеБулочек);
    return r;
}
```

Теперь попытаемся сделать функцию короче и понятнее. Переместим фрагменты кода так, чтобы их можно было вынести в отдельные функции.

Листинг 17.18. Лакомка.java

```
public Отчет создатьОтчет()
{
    Отчет r = new Отчет();
    double изменение = 0;
    double общаяСтоимость = 0;
    double потреблениеБулочек = 0;
    double первыеБулочки = 0;
```

продолжение ⇨

Листинг 17.18 (продолжение)

```
    double wpb = 0;
    if (егоПосещения.size() > 0)
    {
        ПосещениеКафе первоеПосещение =
                            (ПосещениеКафе) егоПосещения.get(0);
        ПосещениеКафе последнееПосещение = (ПосещениеКафе)
                    егоПосещения.get(егоПосещения.size() - 1);
        double первыйЗамер = первоеПосещение.получитьВес();
        double последнийЗамер =
                            последнееПосещение.получитьВес();
        изменение = последнийЗамер - первыйЗамер;
        первыеБулочки = первоеПосещение.получитьБулочки();
        for (int i = 0 ; i < егоПосещения.size(); i++)
        {
            ПосещениеКафе v = (ПосещениеКафе)
                                                егоПосещения.get(i);
            общаяСтоимость += v.получитьСтоимость();
            потреблениеБулочек += v.получитьБулочки();
        }
        потреблениеБулочек -= первыеБулочки;
        if (потреблениеБулочек > 0)
                            wpb = изменение / потреблениеБулочек;
    }
    r.устВесНаБулочку(wpb);
    r.устИзменениеВеса(изменение);
    r.устСтоимостьБулочек(общаяСтоимость);
    r.устПотреблениеБулочек(потреблениеБулочек);
    return r;
}
```

Листинг 17.18 иллюстрирует промежуточный шаг в перемещении фрагментов кода. На пути к нему мы выполнили несколько более мелких шагов. Каждый из этих шагов тестировался. И вот теперь тесты стали завершаться успешно. Облегченно вздохнув, мы увидели, как можно улучшить код. Начнем с разбиения единственного цикла на два цикла.

Листинг 17.19. Лакомка.java

```
if (егоПосещения.size() > 0)
{
    ПосещениеКафе первоеПосещение =
                            (ПосещениеКафе) егоПосещения.get(0);
    ПосещениеКафе последнееПосещение = (ПосещениеКафе)
                    егоПосещения.get(егоПосещения.size() - 1);
    double первыйЗамер = первоеПосещение.получитьВес();
    double последнийЗамер = последнееПосещение.получитьВес();
    изменение = последнийЗамер - первыйЗамер;
    первыеБулочки = первоеПосещение.получитьБулочки();
    for (int i = 0; i < егоПосещения.size(); i++)
    {
        ПосещениеКафе v = (ПосещениеКафе) егоПосещения.get(i);
        потреблениеБулочек += v.получитьБулочки();
    }
    for (int i = 0; i < егоПосещения.size(); i++)
```

```
    {
        ПосещениеКафе v = (ПосещениеКафе) егоПосещения.get(i);
        общаяСтоимость += v.получитьСтоимость();
    }
    потреблениеБулочек -= первыеБулочки;
    if (потреблениеБулочек > 0)
                            wpb = изменение / потреблениеБулочек;
}
```

Выполним тестирование. На следующем шаге поместим каждый цикл в отдельный приватный метод.

Листинг 17.20. Лакомка.java

```
public Отчет создатьОтчет()
{
    Отчет r = new Отчет();
    double изменение = 0;
    double общаяСтоимость = 0;
    double потреблениеБулочек = 0;
    double первыеБулочки = 0;
    double wpb = 0;
    if (егоПосещения.size() > 0)
    {
        ПосещениеКафе первоеПосещение =
                            (ПосещениеКафе) егоПосещения.get(0);
        ПосещениеКафе последнееПосещение = (ПосещениеКафе)
                    егоПосещения.get(егоПосещения.size() - 1);
        double первыйЗамер = первоеПосещение.получитьВес();
        double последнийЗамер =
                                последнееПосещение.получитьВес();
        изменение = последнийЗамер - первыйЗамер;
        первыеБулочки = первоеПосещение.получитьБулочки();
        потреблениеБулочек = вычПотреблениеБулочек();
        общаяСтоимость = вычОбщуюСтоимость();
        потреблениеБулочек -= первыеБулочки;
        if (потреблениеБулочек > 0)
                            wpb = изменение / потреблениеБулочек;
    }
    r.устВесНаБулочку(wpb);
    r.устИзменениеВеса(изменение);
    r.устСтоимостьБулочек(общаяСтоимость);
    r.устПотреблениеБулочек(потреблениеБулочек);
    return r;
}
private double вычОбщуюСтоимость()
{
    double общаяСтоимость = 0;
    for (int i = 0; i < егоПосещения.size(); i++);
    {
        ПосещениеКафе v = (ПосещениеКафе) егоПосещения.get(i);
        общаяСтоимость += v.получитьСтоимость();
    }
    return общаяСтоимость;
}
private double вычПотреблениеБулочек()
```

продолжение ☞

Листинг 17.20 (продолжение)

```
{
    double потреблениеБулочек = 0;
    for (int i = 0; i < егоПосещения.size(); i++)
    {
        ПосещениеКафе v = (ПосещениеКафе) егоПосещения.get(i);
        потреблениеБулочек += v.получитьБулочки();
    }
    return потреблениеБулочек;
}
```

После соответствующего тестирования перенесем обработку вариантов потребления булочек в метод вычПотреблениеБулочек().

Листинг 17.21. Лакомка.java

```
public Отчет создатьОтчет()
{
    ...
    if (егоПосещения.size() > 0)
    {
        ПосещениеКафе первоеПосещение =
                            (ПосещениеКафе) егоПосещения.get(0);
        ПосещениеКафе последнееПосещение = (ПосещениеКафе)
                    егоПосещения.get(егоПосещения.size() - 1);
        double первыйЗамер = первоеПосещение.получитьВес();
        double последнийЗамер =
                                последнееПосещение.получитьВес();
        изменение = последнийЗамер - первыйЗамер;
        потреблениеБулочек = вычПотреблениеБулочек();
        общаяСтоимость = вычОбщуюСтоимость();
        if (потреблениеБулочек > 0)
                        wpb = изменение / потреблениеБулочек;
    }
    ...
    return r;
}
private double вычОбщуюСтоимость()
{
    double общаяСтоимость = 0;
    for (int i= 0; i < егоПосещения.size(); i++);
    {
        ПосещениеКафе v = (ПосещениеКафе) егоПосещения.get(i);
        общаяСтоимость += v.получитьСтоимость();
    }
    return общаяСтоимость;
}
private double вычПотреблениеБулочек()
{
    double потреблениеБулочек = 0;
    if (егоПосещения.size() > 0)
    {
        for (int i = 1; i < егоПосещения.size(); i++)
        {
            ПосещениеКафе v = (ПосещениеКафе)
                                                    егоПосещения.get(i);
```

```
            потреблениеБулочек += v.получитьБулочки();
        }
    }
    return потреблениеБулочек;
}
```

Заметим, что функция вычПотреблениеБулочек() теперь суммирует потребление булочек, начиная со второго посещения. И опять выполняем тестирование. На следующем шаге выделим функцию для расчета изменения веса.

Листинг 17.22. Лакомка.java

```
public Отчет создатьОтчет()
{
    Отчет r = new Отчет();
    double изменение = 0;
    double общаяСтоимость = 0;
    double потреблениеБулочек = 0;
    double первыеБулочки = 0;
    double wpb = 0;
    if (егоПосещения.size() > 0)
    {
        изменение = вычИзменение();
        потреблениеБулочек = вычПотреблениеБулочек();
        общаяСтоимость = вычОбщуюСтоимость();
        if (потреблениеБулочек > 0)
                        wpb = изменение / потреблениеБулочек;
    }
    r.устВесНаБулочку(wpb);
    r.устИзменениеВеса(изменение);
    r.устСтоимостьБулочек(общаяСтоимость);
    r.устПотреблениеБулочек(потреблениеБулочек);
    return r;
}
private double вычИзменение()
{
    double изменение = 0;
    if (егоПосещения.size() > 0)
    {
        ПосещениеКафе первоеПосещение =
                        (ПосещениеКафе) егоПосещения.get(0);
        ПосещениеКафе последнееПосещение = (ПосещениеКафе)
                    егоПосещения.get(егоПосещения.size() - 1);
        double первыйЗамер = первоеПосещение.получитьВес();
        double последнийЗамер =
                        последнееПосещение.получитьВес();
        изменение = последнийЗамер - первыйЗамер;
    }
    return изменение;
}
```

После очередного запуска тестов переместим условия в главном методе создатьОтчет() и подчистим лишние места.

Листинг 17.23. Лакомка.java

```
public Отчет создатьОтчет()
{
```

продолжение ⇩

Листинг 17.23 (продолжение)

```
    double изменение = вычИзменение();
    double потреблениеБулочек = вычПотреблениеБулочек();
    double общаяСтоимость = вычОбщуюСтоимость();
    double wpb = 0;
    if (потреблениеБулочек > 0)
                        wpb = изменение / потреблениеБулочек;
    Отчет r = new Отчет();
    r.устВесНаБулочку(wpb);
    r.устИзменениеВеса(изменение);
    r.устСтоимостьБулочек(общаяСтоимость);
    r.устПотреблениеБулочек(потреблениеБулочек);
    return r;
}
private double вычИзменение()
{
    double изменение = 0;
    if (егоПосещения.size() > 1)
    {
        ПосещениеКафе первоеПосещение =
                        (ПосещениеКафе) егоПосещения.get(0);
        ПосещениеКафе последнееПосещение = (ПосещениеКафе)
                    егоПосещения.get(егоПосещения.size() - 1);
        double первыйЗамер = первоеПосещение.получитьВес();
        double последнийЗамер =
                            последнееПосещение.получитьВес();
        изменение = последнийЗамер - первыйЗамер;
    }
    return изменение;
}
private double вычОбщуюСтоимость()
{
    double общаяСтоимость = 0;
    for (int i= 0; i < егоПосещения.size(); i++);
    {
        ПосещениеКафе v = (ПосещениеКафе) егоПосещения.get(i);
        общаяСтоимость += v.получитьСтоимость();
    }
    return общаяСтоимость;
}
private double вычПотреблениеБулочек()
{
    double потреблениеБулочек = 0;
    if (егоПосещения.size() > 1)
    {
        for (int i = 1; i < егоПосещения.size(); i++)
        {
            ПосещениеКафе v = (ПосещениеКафе)
                                        егоПосещения.get(i);
            потреблениеБулочек += v.получитьБулочки();
        }
    }
    return потреблениеБулочек;
}
```

После окончательного прогона тестов констатируем, что цель достигнута — код стал компактным и понятным, обязанности разнесены по отдельным функциям.

Таким образом, в рассмотренном подходе программа считается завершенной не тогда, когда она заработала, а когда она стала максимально простой и ясной.

Контрольные вопросы и упражнения

1. Что такое *CRC*-карта? Как ее применить для тестирования визуальных моделей?
2. Поясните особенности тестирования объектно-ориентированных модулей.
3. В чем состоит суть методики тестирования интеграции объектно-ориентированных систем, основанной на потоках?
4. Поясните содержание методики тестирования интеграции объектно-ориентированных систем, основанной на использовании.
5. В чем заключаются особенности объектно-ориентированного тестирования правильности?
6. К чему приводит учет инкапсуляции, полиморфизма и наследования при проектировании тестовых вариантов?
7. Поясните содержание тестирования, основанного на ошибках.
8. Поясните содержание тестирования, основанного на сценариях.
9. Чем отличается тестирование поверхностной структуры от тестирования глубинной структуры системы?
10. В чем состоит стохастическое тестирование класса?
11. Охарактеризуйте тестирование разбиений на уровне классов. Как в этом случае получить категории разбиения?
12. Поясните на примере разбиение на категории по состояниям.
13. Приведите пример разбиения на категории по атрибутам.
14. Перечислите известные вам методы тестирования взаимодействия классов. Поясните их содержание.
15. Приведите пример стохастического тестирования взаимодействия классов.
16. Приведите пример тестирования взаимодействия классов путем разбиений.
17. Приведите пример тестирования взаимодействия классов на основе состояний. В чем заключается особенность методики «преимущественно в ширину»?
18. Протестируйте отдельные классы из приложения «Простой интерфейс пользователя для встроенной системы», описанного в главе 13.
19. Протестируйте взаимодействия классов из приложения «Простой интерфейс пользователя для встроенной системы», описанного в главе 13.
20. Протестируйте отдельные классы из приложения «Система управления торговым автоматом», описанного в главе 13.
21. Протестируйте взаимодействия классов из приложения «Система управления торговым автоматом», описанного в главе 13.
22. Поясните суть предваряющего тестирования.
23. Какую роль в процессе экстремальной разработки играет рефакторинг?

Глава 18

Обеспечение качества программных систем

Неотъемлемым свойством современной программной системы является высокое качество. Здесь рассматривается понятие качества ПО и все аспекты его обеспечения: цели, факторы, количественный подход к определению уровня качества, описывается вся панорама деятельности по обеспечению качества. Отдельно обсуждаются технические проверки и аудиты, инспектирование, верификация и валидация, а также оформление плана по обеспечению качества ПО.

Определение качества программного обеспечения

В авторитетном словаре программной инженерии IEEE Std 610.12-90 «IEEE Standard Glossary of Software Engineering Terminology» записано:

1. Качество ПО — это степень соответствия системы, компонента или процесса определенным требованиям.
2. Качество ПО — это степень соответствия системы, компонента или процесса нуждам или ожиданиям заказчика, пользователя.

Фактически эти альтернативные определения придуманы двумя отцами-основателями, архитекторами современного представления о качестве — Филиппом Кросби и Джозефом Джураном.

В 1979 году Ф. Кросби писал: «качество означает соответствие требованиям».

Дж. Джуран в 1988 году ввел такое определение:

«1) качество определяется теми характеристиками продукта, которые удовлетворяют потребности заказчика и таким образом обеспечивают удовлетворение продуктом; 2) качество означает отсутствие недостатков».

Определение Кросби отсылает к степени соответствия написанного ПО тем спецификациям требований, которые были подготовлены заказчиком. Это означает, что погрешности спецификации не учитываются и не снижают программное качество, оцениваемое в пределах погрешностей подхода.

Определение Джурана нацелено на полное удовлетворение заказчика и объявляет истинной целью достижения качества «осуществление реальных нужд заказчика». В этом случае разработчик обязан приложить существенные усилия к исследованию и корректировке спецификации требований заказчика. Основное допущение этого определения состоит в том, что заказчик освобождается от любой ответственности за точность и завершенность программной спецификации. Мало того, заказчику позволяют выражать его реальные потребности (которые могут существенно отличаться от проектной спецификации) на весьма поздней стадии проекта, даже на заключительной стадии. Как результат затруднения могут возникнуть на протяжении всего процесса разработки, особенно при попытках доказать, насколько хорошо программа реализует потребности заказчика.

Дополнительные черты качества включены в определение Роджера Прессмана (2000 г.):

«Качество ПО — это соответствие точно определенным функциональным требованиям и требованиям производительности, точно документированным стандартам разработки, а также подразумеваемым характеристикам, ожидаемым от всего профессионально разработанного ПО».

Здесь внимание акцентируется на трех важных моментах:

- Функциональные и нефункциональные требования к ПО являются фундаментом для измерения качества. Несоответствие требованиям свидетельствует о недостатке качества.
- Стандарты определяют порядок разработки, многочисленные проверки, управление изменениями, контроль версий и набор метрик для количественного измерения важных параметров ПО, показывающих уровень качества. Задаваемый стандартами процесс разработки позволяет избежать проектного хаоса — главного поставщика плохого качества. Игнорирование стандартов всегда отражается на качестве продукта.
- Существует множество подразумеваемых требований, которые часто не упоминаются (например, пожелание легкой сопровождаемости, удобство использования) и являются следствиями применения хороших практик программной инженерии, отражающих достигнутый мировой уровень технологий. Эти требования ориентированы на максимальное удовлетворение всех заинтересованных лиц. Если ПО их не обеспечивает, качество считают сомнительным.

Определение и цели обеспечения качества ПО

Наиболее часто используют определение «обеспечения качества ПО» (SQA — software quality assurance) из словаря IEEE (1990 г.).

Обеспечение качества ПО — это:

1) планируемый и систематичный паттерн всех действий, которые обеспечивают полную уверенность в том, что элемент или продукт соответствует определенным техническим требованиям;
2) набор видов деятельности, проектируемый для оценки процесса, по которому продукты разрабатываются или производятся. Отличается от понятия «контроль качества».

Охарактеризуем данное определение:

- Систематическое планирование и реализация. SQA основано на планировании и применении множества действий, которые интегрируются во все этапы процесса разработки ПО. Это придает заказчику уверенность в том, что программный продукт будет соответствовать всем техническим требованиям.
- Определение IEEE относится лишь к процессу разработки ПО.
- Определение IEEE связывает сопровождение только со спецификациями на технические требования.

Акцентируя внимание на планируемой и систематической реализации, определение IEEE ограничивает область применения SQA, исключая вопросы сопровождения, а также ограничения по времени и бюджету. Очевидно, что для улучшения результатов и достижения более полного удовлетворения заказчика следует учесть следующие расширения:

- Распространить область действия SQA на весь жизненный цикл программной системы. Такое решение позволяет интегрировать вопросы качества в функции сопровождения ПО.
- Не ограничивать SQA техническими вопросами функциональных требований, а распространить его на вопросы планирования и учета бюджета. Известно, что вопросы планирования и бюджетных ограничений существенно влияют на реализацию функциональных технических требований. Очень часто ограничения по времени препятствуют внесению «опасных» изменений в план-график проекта, невзирая на функциональные требования. Нечто подобное может происходить и в условиях серьезных бюджетных ограничений, налагающих запрет на дополнительные ресурсы.

В итоге принимаем следующее расширенное определение:

Обеспечение качества ПО — систематический, планируемый набор действий, необходимых для формирования приемлемого уровня уверенности в том, что процесс разработки и сопровождения программной системы соответствует установленным функциональным техническим требованиям, а также организаторским требованиями соблюдения план-графика и бюджетных ограничений.

Чем отличается контроль качества от обеспечения качества? Эти термины представляют отдельные и различные понятия:

- Контроль качества определен как «набор видов деятельности, проектируемых для оценки качества разрабатываемого или производимого продукта» (IEEE, 1990); другими словами, главная цель этих видов деятельности — отбраковать продукты, не прошедшие оценку. Контроль качества применяется к разработке продукта до момента ее завершения, то есть до отправки продукта заказчику. Типичной деятельностью контроля качества является тестирование.
- Главная цель обеспечения качества — минимизировать стоимость гарантированного качества за счет множества действий, выполняемых на различных этапах процесса разработки (производства). Эти действия предотвращают ошибки, выявляют и корректируют их на ранних стадиях процесса разработки. В результате действия по обеспечению качества существенно уменьшают процент

продуктов, непригодных для отправки, и в то же время сокращают стоимость гарантированного качества в большинстве случаев.

Следовательно, действия контроля качества являются лишь частью в общем списке действий по обеспечению качества.

Действия SQA покрывают функциональные, организационные и экономические аспекты разработки и сопровождения ПО. Сгруппируем цели обеспечения качества по этапам жизненного цикла ПО:

- На этапе разработки ПО (ориентированы на процесс):
  1. Обеспечение приемлемого уровня уверенности в том, что ПО будет соответствовать функциональным техническим требованиям.
  2. Обеспечение приемлемого уровня уверенности в том, что ПО будет создаваться в соответствии с план-графиком и бюджетными требованиями.
  3. Инициализация и управление действиями по совершенствованию и повышению эффективности разработки ПО, а также действий SQA. Это означает улучшение перспектив достижения функциональных и организационных (менеджерских) требований (при снижении стоимости разработки ПО и действий SQA).
- На этапе сопровождения ПО (ориентированы на продукт):
  4. Обеспечение приемлемого уровня уверенности в том, что действия по сопровождению ПО будут соответствовать функциональным техническим требованиям.
  5. Обеспечение приемлемого уровня уверенности в том, что действия по сопровождению ПО будут соответствовать план-графику и бюджетным требованиям.
  6. Инициализация и управление действиями по совершенствованию и повышению эффективности сопровождения ПО, а также действий SQA. Это означает улучшение перспектив достижения функциональных и организационных (менеджерских) требований при сокращении стоимости.

Факторы качества ПО

Качество ПО определяется множеством факторов, которые сильно зависят от предметной области и требований заказчика. Классической моделью факторов качества считают модель Дж. МакКолла, предложенную более 30 лет назад [40, 88]. Дж. МакКолл сгруппировал 11 факторов в три категории:

- Операционные факторы (правильность, надежность, эффективность, целостность, практичность).
- Факторы изменяемости (сопровождаемость, гибкость, тестируемость).
- Факторы перемещаемости (мобильность, многократность использования, способность к взаимодействию).

Во многих случаях нельзя было прямо измерить приведенные факторы. Поэтому были придуманы метрики, используемые для выражения каждого фактора по формуле:

$$F_j = \sum_{i=1}^{n} c_i \times m_i,$$

где F_j — j-й фактор качества ПО, c_i — весовой коэффициент, m_i — метрика, оказывающая влияние на фактор качества ПО.

Оказалось, что многие из придуманных метрик тоже должны измеряться только косвенным путем. Для их вычислений создавались проверочные листы со схемой применения:

1) задаются несколько вопросов;
2) ответы на вопросы записываются;
3) записанные ответы обрабатываются (взвешиваются, суммируются и т. д.).

В настоящее время для определения факторов качества применяют серию из четырех международных стандартов: ISO/IEC 9126-1:2001 «Quality model», ISO/IEC TR 9126-2:2003 «External metrics», ISO/IEC TR 9126-3:2003 «Internal metrics», ISO/IEC TR 9126-4:2004 «Quality in use metrics».

В этих стандартах определены три группы метрик качества: внутренние метрики, внешние метрики и метрики «при использовании».

Внутренние метрики фиксируют внутреннее качество, проявляющееся в ходе разработки ПО.

Внешние метрики определяют внешнее качество, заданное требованиями заказчика и демонстрируемое характеристиками конечной системы.

Метрики «при использовании» измеряют качество в ходе нормальной эксплуатации программной системы конечными пользователями.

Внутренние и внешние метрики ориентированы на шесть характеристик качества:

- *Функциональность* (functionality) — определяет способность продукта обеспечивать требуемые функции и операции;
- *Надежность* (reliability) — показывает возможности продукта для обеспечения достаточного уровня работоспособности в реальных условиях;
- *Практичность* (usability) — означает понятность, легкость использования и изучения продукта;
- *Эффективность* (efficiency) — демонстрирует эффективность продукта при разумном задействовании ресурсов;
- *Сопровождаемость* (maintainability) — измеряет возможности продукта быть изменчивым и обновляемым, а также удобным в сопровождении;
- *Переносимость* (portability) — означает способность продукта быть системой, переносимой из одной аппаратно-программной среды в другую.

В свою очередь, каждая характеристика детализируется своим набором атрибутов (табл. 18.1–18.6).

Таблица 18.1. Детализация характеристики «функциональность»

Атрибут	Описание
Пригодность (suitability)	Пригодность к обеспечению требуемой функциональности
Точность (accuracy)	Обеспечение заданной точности результатов
Способность к взаимодействию (interoperability)	Способность к взаимодействию с внешними системами
Защищенность (security)	Защищенность внутренней информации от неавторизованного использования
Согласованность (compliance)	Отсутствие противоречий с ограничениями стандарта

Таблица 18.2. Детализация характеристики «надежность»

Атрибут	Описание
Зрелость (maturity)	Способность продукта избегать ошибок при генерации исключений и ошибках данных
Устойчивость к отказам (fault tolerance)	Способность продукта поддерживать определенный уровень производительности при генерации исключений и ошибках данных
Восстанавливаемость (recoverability)	Способность продукта восстанавливать определенный уровень производительности при генерации исключений и ошибках данных
Согласованность (compliance)	Отсутствие противоречий с ограничениями стандарта

Таблица 18.3. Детализация характеристики «практичность»

Атрибут	Описание
Понятность (understandability)	Обеспечение понимания пользователем порядка решения конкретной задачи с помощью продукта
Обучаемость (learnability)	Возможность обучения пользователя работе с помощью продукта
Удобство работы (operability)	Обеспечение пользователя возможностью решать все требуемые задачи
Привлекательность (attractiveness)	Притягательность продукта с точки зрения пользователя
Согласованность (compliance)	Отсутствие противоречий с ограничениями стандарта

Таблица 18.4. Детализация характеристики «эффективность»

Атрибут	Описание
Временная эффективность (time behavior)	Обеспечение быстрого взаимодействия и достаточной скорости решения задачи
Ресурсоемкость (resource behavior)	Задействование разумного объема ресурсов при решении задачи
Согласованность (compliance)	Отсутствие противоречий с ограничениями стандарта

Таблица 18.5. Детализация характеристики «сопровождаемость»

Атрибут	Описание
Анализируемость (analysability)	Обеспечение возможности самоанализа при поиске причины ошибочного поведения
Изменяемость (changeability)	Способность изменять структуру программы
Стабильность (stability)	Сохранение стабильности даже при изменении структуры продукта
Тестируемость (testability)	Поддержка валидации продукта с помощью тестирования
Согласованность (compliance)	Отсутствие противоречий с ограничениями стандарта

Таблица 18.6. Детализация характеристики «переносимость»

Атрибут	Описание
Адаптируемость (adaptability)	Возможность адаптироваться к другой среде без привлечения дополнительной функциональности
Простота установки (installability)	Способность инсталлироваться в конкретной среде
Способность к сосуществованию (co-existence)	Способность продукта разделять среду с другой системой
Взаимозаменяемость (replaceability)	Замещаемость компонентов при корректировке системы
Согласованность (compliance)	Отсутствие противоречий с ограничениями стандарта

Приведем примеры метрик для оценки атрибутов качества:

- *Полнота реализации функций* — процент реализованных функций по отношению к количеству, заданному требованиями. Позволяет измерить функциональную пригодность.
- *Корректность реализации функций* — правильность их реализации по отношению к требованиям. Позволяет измерить функциональную пригодность, зрелость.
- *Отношение числа проведенных тестов к общему их числу.* Позволяет измерить зрелость.
- *Наглядность и полнота документации.* Позволяет измерить понятность.

Метрики «при использовании» генерируются для четырех характеристик качества, перечисленных в табл. 18.7.

Таблица 18.7. Характеристики качества для метрик «при использовании»

Характеристика	Описание
Эффективность (effectiveness)	Возможность для пользователя достичь цели с достаточной точностью и полнотой
Продуктивность (productivity)	Возможность для пользователя достичь цели при использовании достаточного объема ресурсов, обеспечивающих нужную производительность
Безопасность (safety)	Обеспечение приемлемого уровня рисков (относящихся к людским ресурсам, данным, среде и т. д.)
Удовлетворенность запросов (satisfaction)	Возможность полного решения задачи с помощью продукта

Схемы вычисления метрик качества для перечисленных характеристик подробно описаны во второй – четвертой частях стандарта. Стандарт определяет возможность вычисления 80 базовых и более 250 производных метрик, предлагает обоснованный выбор мер и шкал для формируемых значений.

Порядок применения стандартов ISO/IEC 9126-1-4 к обеспечению качества ПО задает другая линейка стандартов (ISO/IEC 14598-1, -2, -3, -4, -5, -6), связывая измерения с конкретными этапами жизненного цикла программной системы.

В 2005 году начат процесс создания линейки международных стандартов второго поколения по качеству ПО – стандартов серии 25 000. Данная серия вбирает в себя весь положительный опыт линейки 9126. В частности, в ней уточнен и расширен до восьми набор характеристик качества для внутренних и внешних метрик, существенно обновлен набор характеристик (и введено 13 атрибутов) для метрик «при использовании». Процесс создания серии 25 000 еще не завершен.

Стандартизованные характеристики качества оказывают существенную помощь заинтересованным лицам при формировании требований, они могут служить шаблоном требований. Ведь они определяют:

- Что должно делать ПО (например, поддерживать полный цикл покупки товара в интернет-магазине).
- Степень надежности ПО (работать круглосуточно, 365 дней в году).
- Степень удобства использования (обучать продавца всем функциям и возможностям за один рабочий день).
- Степень эффективности (поддерживать параллельную работу 100 покупателей).
- Удобство сопровождения (сохранять работоспособность при добавлении новых функций заказа, оплаты, доставки).
- Степень переносимости (работать в среде операционных систем Linux, Windows XP, Windows Vista и Windows 7, формировать отчеты в форматах MS Word, Excel, HTML, XML).

При необходимости требования могут детализироваться с использованием полного набора атрибутов качества.

Деятельность по обеспечению качества ПО

Обеспечение качества ПО (SQA) – защитная деятельность, применяемая на всем протяжении жизненного цикла системы и включающая следующие виды действий:

- Применение технических методов и средств анализа, проектирования, кодирования и сопровождения.
- Проведение технических проверок и аудитов на каждом шаге разработки.
- Верификация и валидация продукта.
- Внедрение в жизнь стандартов.
- Контроль изменений.
- Проведение измерений показателей качества и их оценка.
- Сохранение записей и формирование отчетности.

Технические методы и средства помогают аналитику получить высококачественную спецификацию, а проектировщику (программисту) — провести высококачественное проектирование (программирование).

После того как спецификация и проектное решение (программный код) созданы, осуществляется оценка их качества. Для этого используют самые разные действия: технические проверки и аудиты, верификацию и валидацию. Подробно эти действия мы обсудим отдельно.

Техническая проверка и аудит организуются как встреча персонала с единственной целью — обнаружить проблемы качества. Верификация и валидация — это не единичный акт, а процесс, проверяющий соответствие ПО своей спецификации и требованиям заказчиков. Верификация и валидация охватывает полный жизненный цикл ПО, начинается на этапе анализа требований и завершается проверкой готовой программной системы. Основными средствами верификации и валидации являются инспектирование и тестирование. Многие разработчики используют тестирование ПО как спасательную сеть для обеспечения качества. Иными словами, они полагают, что полное тестирование обнаружит большинство ошибок, вследствие чего ослабляется необходимость в других действиях по обеспечению качества. К сожалению, тестирование эффективно не для всех классов ошибок.

Диапазон стандартов и процедурных норм, применяемых к процессу разработки ПО разными компаниями, существенно различен. Во многих случаях стандарты задаются заказчиком или контрактом. В других случаях они выбираются самими разработчиками. Хорошо организованный процесс SQA создает в компании атмосферу культивирования качества, стимулирующий команду к качественной работе и поиску путей повышения качества. При этом стандарты и процедурные нормы считаются «камертоном качества».

Если стандарты приняты, то действия по обеспечению качества должны гарантировать их соблюдение. Оценка соответствия стандартам может выполняться разработчиками как часть технической проверки, или в ситуациях, где требуется независимая верификация, специальная группа качества может проводить свою собственную ревизию.

Наиболее вероятным источником угрозы для качества ПО являются изменения. Каждое изменение в ПО рассматривается как потенциальный источник ошибок или как предпосылка для распространения ошибок. Процесс контроля изменений (он входит в процесс управления конфигурацией) прямо способствует качеству ПО:

1) формализацией запросов на изменения;
2) оценкой сущности изменения;
3) контролем влияния изменения.

Контроль изменений осуществляется как на стадии разработки, так и на стадии сопровождения.

Проведение измерений показателей качества — действие, которое интегрируется в любую инженерную дисциплину. Важной целью обеспечения качества считается отслеживание качества и оценка влияния методологических и процедурных изменений на улучшение качества ПО. Для достижения этой цели должны собираться результаты измерений, проводимых по метрикам качества ПО.

Сохранение записей и формирование отчетности предусматривает процедуры для сбора и распространения информации. Результаты проверок, аудитов, инспектирований, контроля изменений, тестирования и другой деятельности группы контроля качества должны стать частью архива записей по проекту и распространяться среди разработчиков по мере необходимости. Например, результаты каждой технической проверки по проектированию записываются и могут помещаться в брошюру, которая содержит всю техническую информацию о системе и информацию по качеству.

Технические проверки и аудиты

Проверки ПО — фильтр для процесса разработки. Они позволяют очистить от дефектов такие виды действий, как анализ, проектирование, кодирование. Существует много типов проверок, которые могут проводиться как часть процесса SQA (неформальная встреча у кофейного автомата, формальное представление программного проекта аудитории из заказчиков, руководства, технического персонала и т. д.). У каждого типа свое место. Мы будем рассматривать техническую проверку, которая проводится программными инженерами для программных инженеров.

Назначение технической проверки:

1. Обнаружение ошибок и противоречий в функциях, логике или реализации для любой формы представления ПО, включая документацию.
2. Проверка соответствия ПО требованиям.
3. Обеспечение того, чтобы ПО представлялось согласно определенным стандартам.
4. Получение универсальной формы представления ПО.
5. Формирование более управляемого проекта.

Проверка основана на документации, однако она не ограничивается только спецификацией, диаграммами или программным кодом. Проверяются и такие документы, как план-график разработки, планы тестирования, механизм управления конфигурацией, стандарты и документация для пользователя.

В группу проверки включаются такие специалисты, которые могут принести максимальную пользу. Например, при проверке артефактов проектирования следует привлечь проектировщиков из схожих проектов.

Группу образуют 3–4 человека, один из них назначается старшим. Кроме этого, по частным вопросам могут привлекаться дополнительные сотрудники. Проверяемые документы раздаются до начала проверки, для предварительного ознакомления. Сама проверка длится не более двух часов. По окончании проверки все замечания заносятся в отчет. Отчет подписывается всеми проверяющими.

Аудит отличается от технической проверки тем, что не ограничивается по времени, а в состав группы входят лица из внешних организаций. Здесь, как правило, разработку рассматривают с точки зрения контрактных обязательств, привязывая выполненную работу к план-графику проекта.

Очевидная польза технических проверок — раннее обнаружение программных дефектов. Обнаруженные дефекты исправляются перед следующим шагом разработки. Статистика показывает, что 50–60% всех ошибок разработки ПО приходится на действия по проектированию. Технические проверки обеспечивают

обнаружение до 75% недостатков проектирования. Таким образом, обнаруживая и устраняя большой процент ошибок, процесс проверки существенно сокращает стоимость последующих шагов разработки и сопровождения.

Анализ большого числа программных проектов привел к следующим цифрам. Примем цену устранения ошибки, обнаруженной в ходе проектирования, за единицу. Тогда для ошибки, обнаруженной перед тестированием, цена составит 6,5 единицы; в течение тестирования — 15 единиц, а после завершения проекта — 60–100 единиц.

Инспектирование

В отличие от тестирования инспектирование является статическим способом поиска ошибок. Инспектирование заключается в просмотре и проверке артефактов проекта на наличие ошибок. Такими артефактами могут быть требования, результаты проектирования, программный код и т. д. Главная цель инспектирования — обнаружение дефектов, а не исследование общих проблем проекта. Дефектами являются либо ошибки во фрагменте системы, либо несоответствие фрагмента принятым стандартам.

Идея инспектирования принадлежит сотруднику фирмы IBM М. Фагану [15, 40]. В настоящее время данный метод верификации ПО получил широкое применение. Подразумевается, что инспектирование осуществляется коллегами автора, которые анализируют исходный программный код и помогают автору найти в нем дефекты.

Процесс инспектирования формализован, выполняется группой разработчиков, состоящей из четырех человек. Члены группы должны выполнять следующие роли: автора, рецензента, корректора и регистратора (табл. 18.8). Автор отвечает за свою работу и за исправление в ней дефектов. Рецензент комментирует программный код, корректор проверяет код, регистратор записывает обнаруженные дефекты.

Таблица 18.8. Роли группы инспектирования

Роль	Описание
Рецензент	Комментирует (рецензирует) программный код на собрании группы инспектирования, отвечает за правильное проведение инспектирования
Автор	Несет ответственность за свою программу или документ, после завершения инспектирования самостоятельно исправляет все обнаруженные дефекты
Корректор	Ищет в программе (или документе) ошибки, противоречия и упущения
Регистратор	Отвечает за учет описания и классификацию обнаруженных дефектов, записывает результаты собрания группы инспектирования

Рассмотрим содержание этапов процесса инспектирования (рис. 18.1).

1. *Планирование.* Рецензент составляет план инспектирования исходя из диалога с автором и другими членами группы инспектирования.

2. *Предварительный просмотр.* На собрании автор знакомит группу с назначением и основными особенностями программы (объекта инспектирования).
3. *Индивидуальная подготовка.* Участникам необходимо тщательно подготовиться к инспектированию. Ведь инспектирующие должны понимать программу на том же уровне детализации, что и автор. Каждый член инспекционной группы самостоятельно вникает в программу и ее спецификацию, фиксируя обнаруженные недостатки. Этот этап требует существенных усилий.
4. *Собрание инспекционной группы.* Как только каждый из участников группы готов, проводится собрание, в ходе которого участники выполняют свои роли. На собрании активно используются проверочные списки, содержащие вопросы, на которые следует обратить особое внимание (табл. 18.9). Собрание длится не более двух часов и выявляет дефекты, аномалии и несоответствия стандартам. Способы исправления обнаруженных дефектов не рассматриваются.
5. *Исправление ошибок.* После инспектирования автор изменяет программу, исправляя обнаруженные ошибки.
6. *Доработка.* Рецензент принимает решение о необходимости повторного инспектирования. Если оно не требуется, все обнаруженные дефекты оформляются документально, а документ подписывается. Группа собирается в последний раз, подводит итоги, обновляет проверочные списки.

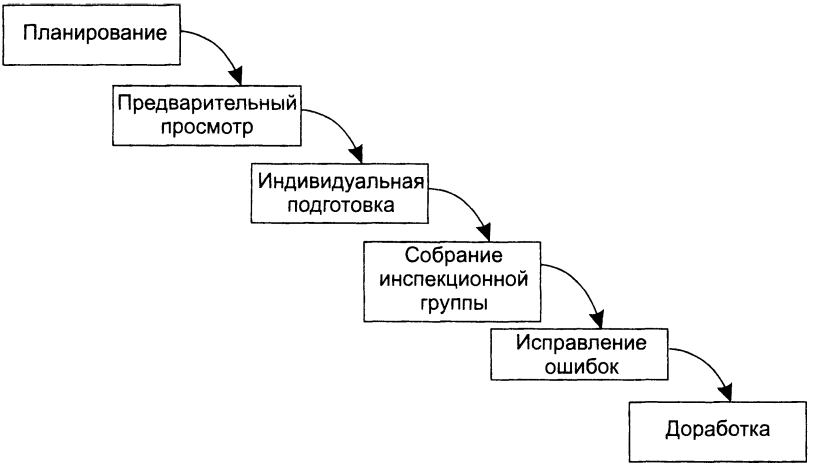


Рис. 18.1. Этапы процесса инспектирования

Таблица 18.9. Пример проверочного списка для инспектирования программного кода

Вопросы для выявления ошибок
Все ли переменные получили значения до начала их использования?
Все ли константы именованы?
Корректно ли заданы границы массивов?
Выполняются ли условия для каждого условного оператора?

продолжение

Таблица 18.9 (продолжение)

Вопросы для выявления ошибок
Все ли циклы завершаются?
Правильно ли расставлены скобки в составных операторах?
Все ли варианты выбора реализуются в case-операторах?
Используются ли в программе входные переменные?
Все ли выходные переменные получают значения перед выводом на печать?
Могут ли какие-нибудь входные данные привести к нарушению данных системы?
Все ли вызовы процедур и функций содержат правильное количество параметров?
Согласованы ли типы формальных и фактических параметров?
В правильном ли порядке расположены параметры?
Если модули обращаются к разделяемым ресурсам, обеспечены ли взаимное исключение и условная синхронизация?
Если связанная структура данных изменяется, правильно ли переопределяются ее связи?
Если используется динамическая память, правильно ли она распределяется?
Происходит ли сбор мусора и решена ли проблема повисших указателей?
Все ли возможные ошибки рассмотрены в условиях, определяющих исключительные ситуации?

В программной индустрии ведут учет времени, потраченного на инспектирование, и объема проверенной работы. Статистика показывает, что на инспектирование уходит порядка 10–15% бюджета разработки. И все же отмечается, что инспектирование, несмотря на большие затраты времени экспертов и дороговизну проведения, вполне окупает себя.

Верификация и валидация

Верификация и *валидация* (V&V — verification and validation) являются составной частью процесса контроля качества. Определения этих терминов из стандарта IEEE Std 1012-2004 «IEEE Standard for Software Verification and Validation» приведены в табл. 18.10 и 18.11.

Таблица 18.10. Определение понятия «верификация»

Вариант	**Описание**
А	Процесс оценки системы или компонента для определения — удовлетворяют ли продукты данного этапа разработки условиям, наложенным в начале этапа
Б	Процесс подтверждения того, что ПО и ассоциированные с ним продукты соответствуют требованиям (например, по правильности, законченности, совместимости, точности) ко всем видам деятельности жизненного цикла (приобретению, поставке, разработке, использованию и сопровождению); соответствие стандартам, практикам и соглашениям для всех процессов жизненного цикла; полное и успешное завершение каждой деятельности жизненного цикла и соответствие всем критериям для инициализации последующих видов деятельности жизненного цикла (например, корректной сборки ПО)

Таблица 18.11. Определение понятия «валидация»

Вариант	Описание
А	Оценка системы или компонента во время или в конце процесса разработки для определения — удовлетворяет ли она (он) указанным требованиям
Б	Процесс подтверждения того, что ПО и ассоциированные с ним продукты удовлетворяют системным требованиям к программному продукту, определенным для завершения каждого вида деятельности жизненного цикла, правильно решают проблемы (например, учитывают закономерности предметной области, бизнес-правила, используют правильные системные допущения), а также соответствуют назначению и нуждам пользователей

Верификация отвечает на вопрос: правильно ли строится, создается наша система? Валидация же проверяет: правильно ли работает система, отвечает ли построенная система пожеланиям и нуждам заказчика?

Приведем простой пример, воспользовавшись известной задачей Гленфорда Майерса [10]. Заказчик предложил разработать систему, которая «решает квадратные уравнения вида $ax^2 + bx + c = 0$ для коэффициентов, представляемых целыми числами». Сформулируем это пожелание в виде детальных требований:

1. Пользователь вводит целые числа a, b и c.
2. Система отыскивает решение уравнения $ax^2 + bx + c = 0$ с точностью 0,001.

Затем мы пишем систему, реализующую эти требования.

Верификация заключается в том, что мы анализируем процесс построения системы и проверяем, что в этом процессе все сделано правильно. Положим, что процесс разработки включает в себя следующие шаги:

1. Получение требований заказчика.
2. Подготовка детальных требований.
3. Одобрение заказчиком детальных требований.
4. Программирование системы.
5. Тестирование системы.

Верификация проверяет следующие вопросы:

1. Выражают ли требования реальные запросы и нужды заказчика? Здесь верификация может включать в себя следующие пункты: является ли уравнение $ax^2 + bx + c = 0$ корректной желаемой формой? Верно ли задана точность? Может ли быть $a = 0$?
2. Правильно ли реализован процесс одобрения заказчиком? Здесь возможны такие вопросы: Предоставили ли мы заказчику достаточно времени? Объяснили ли мы заказчику все используемые термины?
3. Реализует ли программный код системы все требования? Здесь следует выполнить инспектирование кода, где он анализируется с точки зрения соответствия каждому требованию.

4. В какой степени предлагаемые тесты покрывают область применимости системы? Для реальной системы следует применить инспектирование плана тестирования, тестовых вариантов и тестовых процедур. Эти действия дополняют собственно процесс тестирования.

Валидация нашего продукта заключается в получении одобрения заказчиком полных требований и в тестировании конечного кода.

Примеры тестовых вариантов:

- $a = 0, b = 0, c = 0$. Уравнение сводится к виду $0 = 0$ и не может быть разрешено относительно x.
- $a = 0, b = 0, c = 10$. Уравнение сводится к виду $10 = 0$, которое не имеет решений.
- $a = 0, b = 5, c = 17$. Соответствующее уравнение не является квадратным. Справится ли с ним система?
- $a = 6, b = 1, c = 2$. Это один из «нормальных» вариантов, для которого нужно вычислить ожидаемый результат.
- $a = 3, b = 7, c = 0$. Еще один «нормальный тест», когда один из корней равен нулю и т. д.

В качестве выводов отметим, что верификация проверяет соответствие ПО функциональным и нефункциональным требованиям. Валидация является более общим процессом, в ходе которого надо убедиться, что программная система соответствует ожиданиям заказчика. Валидация проводится после верификации, для того чтобы определить, насколько система соответствует не только спецификации, но и ожиданиям заказчика.

Конечно, в требованиях тоже встречаются ошибки и упущения, из-за которых конечный продукт может не совсем соответствовать ожиданиям заказчика. Поэтому требования тоже рекомендуется подвергать валидации.

В ходе верификации и валидации используют два основных действия:

- Инспектирование ПО. Инспектирование осуществляется на всех этапах разработки программной системы. Параллельно с инспектированием может выполняться автоматический анализ программного кода и других артефактов (например, различных документов). Инспектирование и автоматический анализ являются статическими методами верификации и валидации, поскольку им не требуется работающая система.
- Тестирование ПО. Требует запуска фрагментов исполняемого кода. Тестирование считается динамическим методом верификации и валидации, так как применяется к работающей системе.

Выводы: инспектирование можно выполнять на всех этапах разработки системы, а тестирование — лишь в тех случаях, когда создан исполняемый код.

Инспектирование и тестирование не следует считать конкурирующими инструментами верификации и валидации. У каждого из них есть свои достоинства и недостатки, поэтому целесообразно их совместное применение в верификации и валидации.

К методам инспектирования относят инспектирование программ и автоматический анализ исходного кода, проверяющий его синтаксическую и семантическую корректность (хотя современные средства проверки семантики весьма слабы). Но статичность ограничивает область действия инспектирования проверкой спецификаций. Увы, инспектировать функционирование продукта просто невозможно. Нельзя применить инспектирование и к проверке многих нефункциональных характеристик (например, производительности, надежности). Поэтому для этих целей используют тестирование.

В силу описанных причин, доминирующим, базовым методом верификации и валидации остается тестирование. На разных этапах процесса разработки ПО применяют различные виды тестирования, которые подробно обсуждались в главах 14–17.

Увы, создание идеального ПО может быть лишь мечтой, этакой «синей птицей». Верификация и валидация выявляют лишь степень приближения к идеалу заказчика, степень соответствия программной системы запланированным целям. Эта степень определяется:

- назначением системы (для критических систем степень соответствия максимальна, например для системы управления ядерным реактором);
- ожиданиями пользователей (хотят заплатить мало, но терпеть недостатки, или платят много и недостатков не признают);
- условиями на рынке программных продуктов (есть конкуренция для необходимого семейства продуктов, или она отсутствует).

План обеспечения качества ПО

План обеспечения качества ПО (SQA) отражает широкую панораму действий, ориентированных на формирование качества. Разрабатываемый группой SQA, план служит шаблоном действий, которые должны быть внедрены в каждый программный проект.

В стандарте IEEE Std 730-2002 предлагается следующая структура плана SQA:

1. Цель.
2. Упоминаемые документы.
3. Руководство.
   - 3.1. Организация.
   - 3.2. Задачи.
   - 3.3. Обязанности.
   - 3.4. Оцениваемые ресурсы обеспечения качества.
4. Документация.
   - 4.1. Цель.
   - 4.2. Минимальные требования к документации.
   - 4.3. Прочее.

5. Стандарты, практики, соглашения и метрики.
   5.1. Цель.
   5.2. Содержание.
6. Проверки.
   6.1. Цель.
   6.2. Минимальные требования:
      - Проверка спецификаций ПО;
      - Проверка архитектурного проектирования;
      - Проверка детального проектирования;
      - Проверка плана верификации и валидации;
      - Аудит функциональности;
      - Аудит физических компонентов;
      - Внутренние аудиты процесса;
      - Проверка организации;
      - Проверка плана управления конфигурацией;
      - Проверка пост-реализации.
   6.3. Прочие проверки и аудиты.
7. Тестирование.
   Может ссылаться на документацию по тестированию ПО.
8. Отчеты о проблемах и корректирующие действия.
9. Инструменты, технологии и методологии.
   Может ссылаться на план управления программным проектом.
10. Контроль носителей.
11. Контроль поставщиков.
12. Сбор, сопровождение и хранение протоколов.
13. Обучение.
14. Управление рисками.
    Может ссылаться на план управления программным проектом.
15. Словарь.
16. Процедура изменения плана и версии.

При работе над планом SQA важно придерживаться предельно лаконичного стиля изложения. Если документ окажется слишком длинным, то вряд ли его дочитают до конца, что сведет на нет саму идею плана обеспечения качества.

Контрольные вопросы и упражнения

1. Дайте определение понятий качества по Кросби и Джурану. Сравните возможности их применения в программной инженерии. За счет чего можно повысить их прагматичность?

2. Что называют обеспечением качества ПО? Почему «официальное» определение понадобилось расширить?
3. Сравните понятия «обеспечение качества» и контроль качества». Связаны ли они друг с другом?
4. Охарактеризуйте количественный подход к измерению качества. Что такое факторы качества? Как формируются метрики для их измерения?
5. Когда можно применять внутренние метрики? Приведите примеры.
6. Когда следует применять внешние метрики? Приведите примеры их применения.
7. Для чего применяют метрики «при использовании»? Ответ проиллюстрируйте примерами.
8. На какие характеристики качества ориентированы внешние и внутренние метрики? Детализируйте содержание характеристик.
9. Поясните атрибуты для «функциональности».
10. Охарактеризуйте атрибуты для «надежности».
11. Опишите атрибуты для «практичности».
12. Прокомментируйте атрибуты для «эффективности».
13. Какие акценты расставляют атрибуты для «сопровождаемости».
14. Поясните атрибуты для «переносимости».
15. Поясните характеристики качества, измеряемые метриками «при использовании».
16. Дайте развернутую картину содержания деятельности по обеспечению качества.
17. Прокомментируйте порядок проведения и задачи технической проверки. В чем состоят отличительные черты аудита?
18. Чем отличается инспектирование от технической проверки?
19. Укажите сходства и различия тестирования и инспектирования.
20. Каков порядок инспектирования?
21. Какие категории ошибок нельзя обнаружить при инспектировании?
22. Составьте проверочный список для инспектирования кода на двух известных вам языках программирования. Прокомментируйте различия этих списков.
23. Перечислите достоинства и недостатки инспектирования.
24. Чем, на ваш взгляд, обусловлена дороговизна инспектирования?
25. Чем схожи и в чем отличаются верификация и валидация?
26. Какие методы и при каких условиях применяют в верификации и валидации?
27. Составьте план обеспечения качества для одного из проектов, описанных в главе 13, для своего собственного проекта.

Глава 19

Автоматизация разработки визуальной модели программной системы

В современных условиях создание сложных программных приложений невозможно без использования систем автоматизированной разработки ПО (CASE-систем). CASE-системы существенно сокращают сроки и затраты разработки, оказывая помощь инженеру в проведении рутинных операций, облегчая его работу на самых разных этапах жизненного цикла разработки. Наиболее развитой из CASE-систем в настоящее время принято считать IBM Rational Software Architect (RSA). В данной главе рассматривается порядок применения Rational Software Architect при формировании требований, анализе, проектировании и генерации программного кода.

Общая характеристика системы IBM Rational Software Architect

IBM Rational Software Architect представляет собой интегрированное средство визуального моделирования объектно-ориентированных программных продуктов. Визуальное моделирование — процесс графического описания разрабатываемого программного обеспечения.

Ключевые функции RSA перечислены в табл. 19.1.

Таблица 19.1. Ключевые функции Rational Software Architect

Функции	Преимущества
Поддержка UML 2 для формирования требований, анализа и проектирования с использованием диаграмм Use Case, классов, диаграмм последовательности, диаграмм коммуникации, диаграмм деятельности, конечных автоматов, компонентов и развертывания	UML 2.0 позволяет создавать все артефакты визуального моделирования в стандартной нотации, принятой различными заинтересованными лицами

Функции	Преимущества
Формирование для UML-проектов отчетов в форматах HTML, PDF и XML	Создание отчетов и документации, которые могут просматривать разработчики и другие заинтересованные лица
Использование механизма трансформаций для быстрого перехода от модели к коду и от кода к модели для Java/J2EE, WSDL, XSD, SOA, C/C++ и CORBA IDL	Автоматизация задач по генерации кода на основе моделей проекта. Трансформации можно настраивать в соответствии с шаблонами генерации кода, принятыми в организации
Редактирование UML-диаграмм и программного кода Java/J2EE, объектов баз данных	Графическая нотация UML упрощает разработку и осмысление кода для новых и существующих приложений

Экран среды Rational Software Architect показан на рис. 19.1.

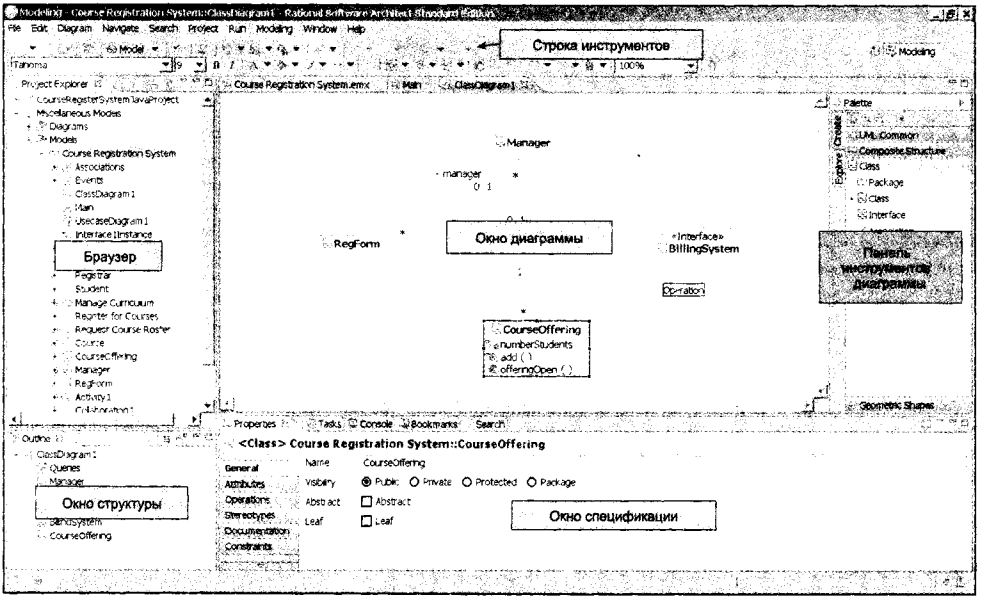


Рис. 19.1. Экран среды Rational Software Architect

В его составе выделим шесть элементов: строку инструментов, браузер, окно структуры, окно диаграммы, палитру инструментов диаграммы и окно спецификации.

Элементы строки инструментов (рис. 19.2) позволяют выполнять стандартные и специальные действия.

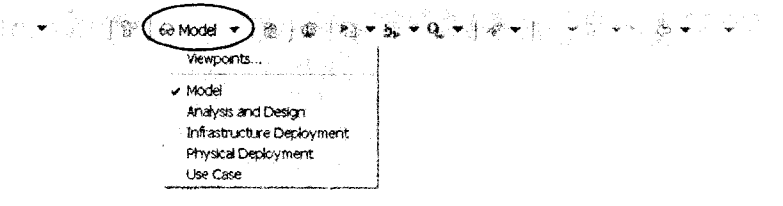


Рис. 19.2. Элементы строки инструментов Rational Software Architect

Обратим внимание на раскрывающийся список (Model Viewpoints), определяющий точку зрения на проект, которая меняет вид экрана RSA.

Браузер Rational Software Architect (Project Explorer) является инструментом иерархической навигации, позволяющим просматривать названия и пиктограммы, отображающие диаграммы и элементы визуальной модели (рис. 19.3). Знак плюс (+) рядом с папкой означает, что внутри папки находятся дополнительные элементы. Для «разворачивания» папки надо нажать на знак +. Если папка «развернута», то слева от нее появляется знак минус (–). Для «сворачивания» структуры папки нажимается знак минус. Браузер позволяет добавлять, перемещать, упорядочивать и сортировать элементы модели.

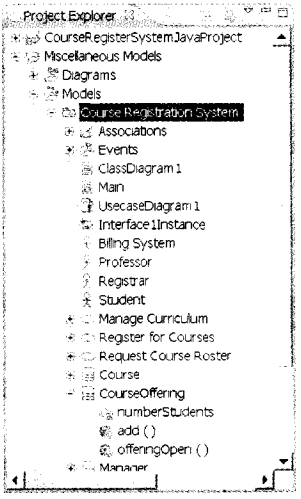


Рис. 19.3. Браузер Rational Software Architect

Окно структуры отображает структуру элемента модели, например диаграммы, открытой в данный момент в области окна диаграммы. Информация в окне (рис. 19.4) представляется в виде последовательности элементов.

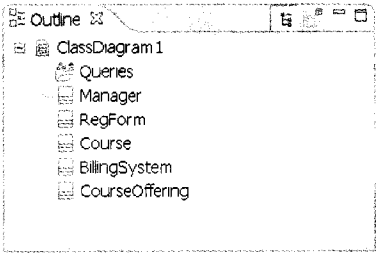


Рис. 19.4. Окно структуры в Rational Software Architect

Палитра инструментов диаграммы содержит средства, используемые при создании диаграммы. Содержание палитры меняется в зависимости от вида активной диаграммы.

В окне диаграммы можно создавать, отображать и изменять диаграмму на языке UML. В качестве примера на рис. 19.5 показано окно для диаграммы классов и палитра инструментов, используемых в этом типе диаграммы.

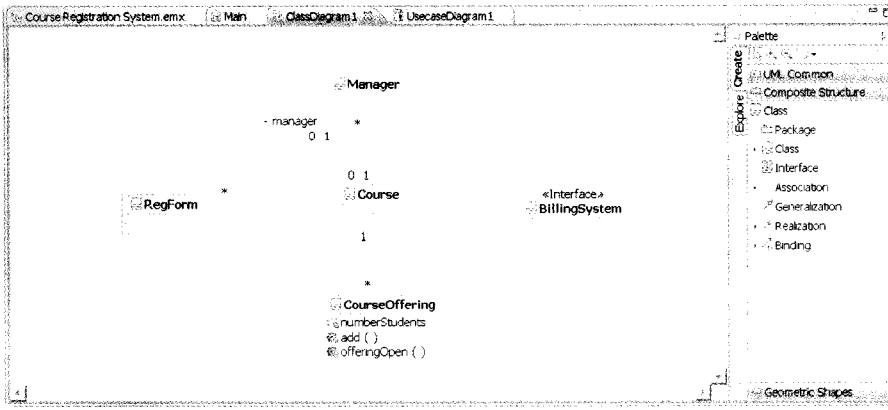


Рис. 19.5. Окно и палитра инструментов для диаграммы классов Rational Software Architect

Все модели, диаграммы, вершины и отношения диаграмм, равно как и остальные ресурсы проекта, имеют определенные свойства. Окно спецификации (рис. 19.6.) позволяет задавать и переопределять значения свойств. Все свойства в окне разбиты на категории. Состав этих категорий зависит от вида рассматриваемого объекта (модель, диаграмма, класс и т. д.), но в окне обязательно присутствуют следующие категории:

- General: Содержит основные характеристики объекта, например имя, уровень видимости и т. д.
- Documentation: Используется для описания (документирования) объекта.
- Advanced: Отображает страницу свойств в стиле Eclipse (базисной среды, использованной для создания RSA).

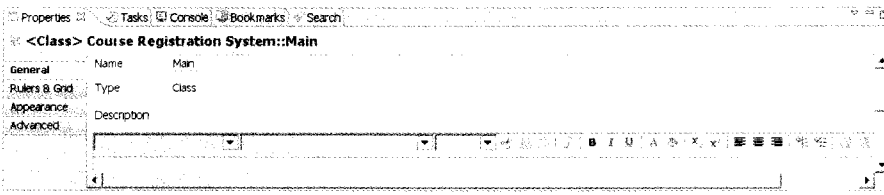


Рис. 19.6. Окно спецификации в Rational Software Architect

В качестве примера работы с Rational Software Architect рассмотрим построение модели университетской системы для регистрации учебных курсов. Эта система используется:

- профессором — для определения содержания читаемого курса;

- студентом — для выбора изучаемого курса;
- регистратором — для формирования учебного плана и расписания;
- учетной системой — для определения денежных затрат.

RSA предоставляет разнообразные варианты построения проектов и моделей. Воспользуемся одним из простейших вариантов. Для создания модели в главном меню системы RSA выберем пункт New — Other (рис. 19.7).

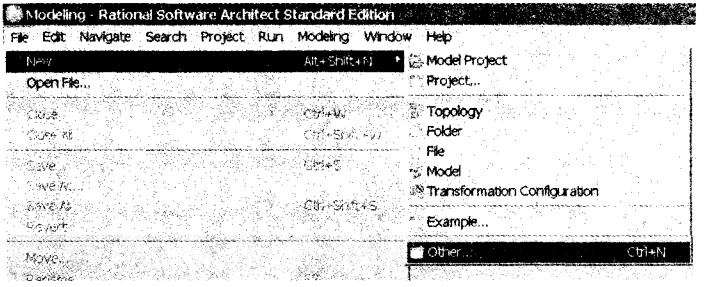


Рис. 19.7. Создание модели системы регистрации учебных курсов

На экране появится окно Select a wizard (Выберите программу-мастер), в котором выбираем пункт UML Model (рис. 19.8).

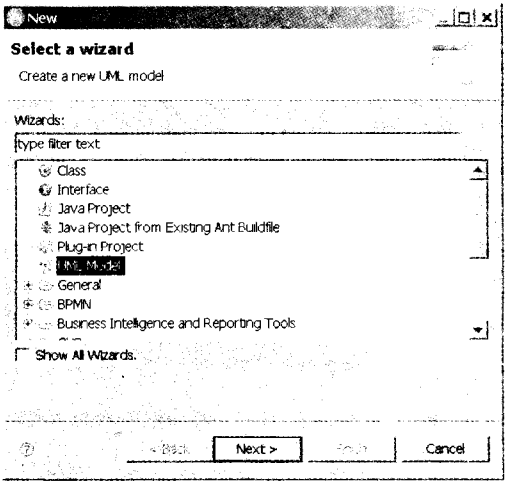


Рис. 19.8. Выбор вида модели

После щелчка по Next будет выведено окно Create Model (создание модели), где предлагается создать модель на базе стандартного шаблона или существующей модели. Выбираем вариант стандартного шаблона и щелкаем по Next (рис.19.9).

Это приводит к выводу следующего диалогового окна Create Model, позволяющего выбрать категорию, к которой относится создаваемая модель (раздел Categories), и один из стандартных шаблонов, предлагаемых для выбранной категории

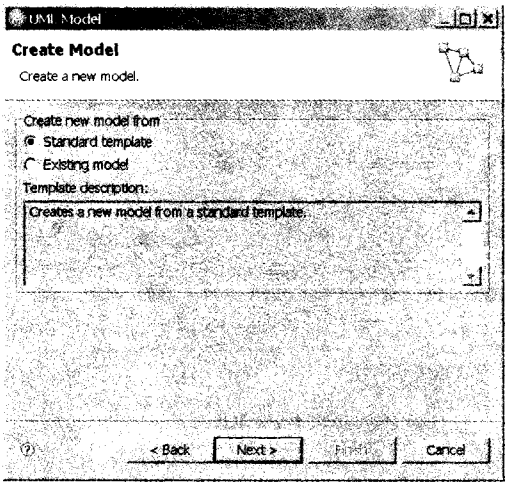


Рис. 19.9. Выбор стандартного шаблона в качестве основы для создания модели

(раздел Templates). Для нашего примера выберем категорию Analysis and Design (анализ и проектирование) и шаблон Blank Analysis Package (пустой пакет анализа). В окошке File name укажем название файла с нашей моделью Course Registration System (рис. 19.10).

Щелчком по Next переходим к следующему окну диалога. Пропускаем его (щелкаем по Next) и в следующем окне Model Capabilities (рис. 19.11), чтобы иметь

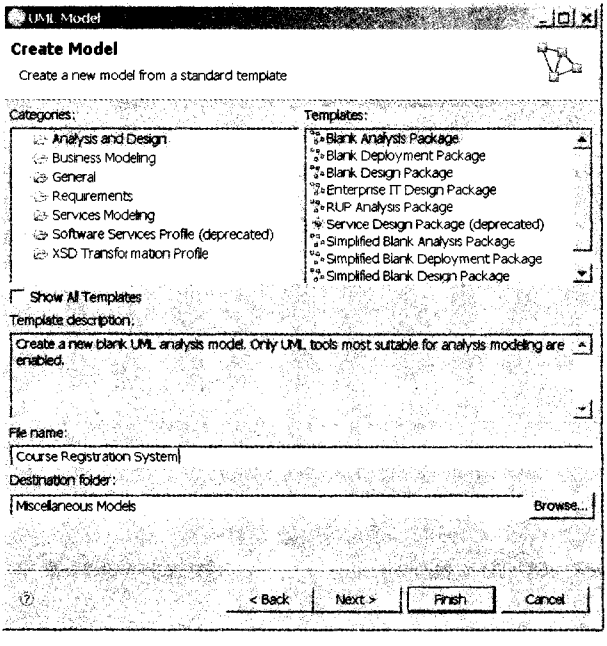


Рис. 19.10. Выбор шаблона для создания модели и определение ее названия

возможность максимального использования средств RSA, делаем щелчок по кнопке Enable All. Завершаем щелчком по кнопке Finish.

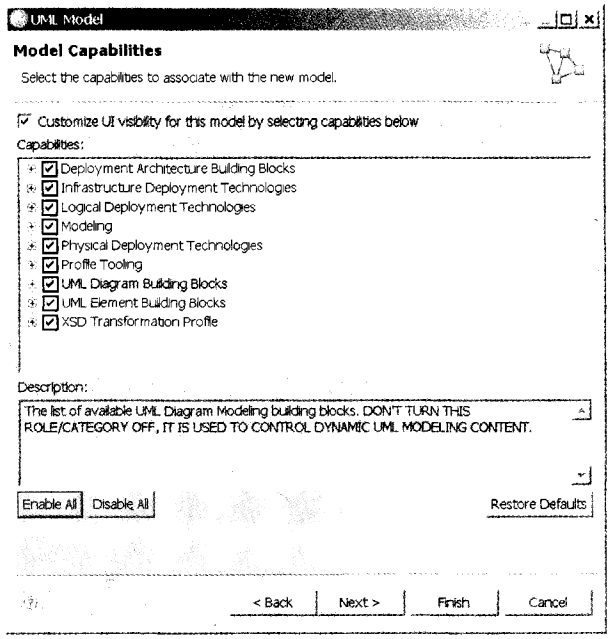


Рис. 19.11. Выбор возможностей RSA, доступных в новой модели

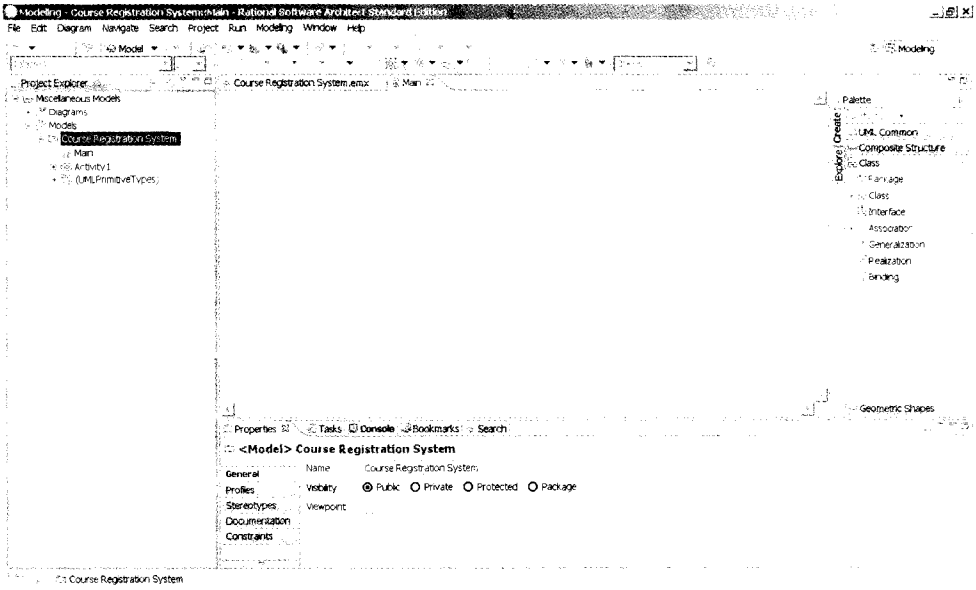


Рис. 19.12. Экран для модели системы регистрации учебных курсов

В результате перечисленных действий экран RSA приобретет вид, показанный на рис. 19.12. Центральную часть экрана занимает главное окно модели. В левой части экрана в браузере проектов (Project explorer) отображается дерево элементов модели.

Теперь приступим к построению модели.

Создание диаграммы Use Case

Моделирование системы регистрации курсов начнем с создания диаграммы Use Case. Этот тип диаграммы представляется актерами, элементами Use Case и отношениями между ними. Для создания диаграммы Use Case:

1) в окне браузера раскроем пункт Diagrams (диаграммы), для чего щелкнем по значку + слева от названия этого пункта;
2) в раскрывшемся списке выбираем нашу модель (Course Registration System) и делаем щелчок правой кнопкой мышки;
3) в появляющемся меню и подменю выбираем пункт Add Diagram > Use Case Diagram.

Описанную последовательность иллюстрирует рис. 19.13.

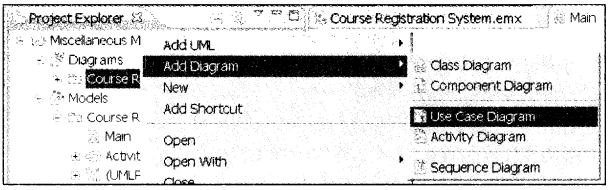


Рис. 19.13. Открытие окна диаграммы Use Case

На экране появится окно, в котором и будет создаваться диаграмма Use Case (рис. 19.14). Оставим ее название по умолчанию — UsecaseDiagram1. Справа от окна диаграммы располагается палитра инструментов (Palette), применяемых для создания диаграмм Use Case.

Первый шаг построения диаграммы состоит в определении актеров, фиксирующих роли внешних объектов, взаимодействующих с системой. В рассматриваемой предметной области будут фигурировать четыре актера — Student (студент), Professor (профессор), Registrar (регистратор) и Billing System (учетная система).

Для начала введем в диаграмму актера Student. Для этого выполним следующие действия:

1. В палитре инструментов Palette щелкнем по значку Actor.
2. Щелкнем в том месте диаграммы, куда мы собираемся добавить актера.
3. Пока пиктограмма актера остается выделенной, введем имя актера — Student.
4. Теперь опишем содержание актера Student. Такое описание можно сделать на странице Properties в окне, расположенном под диаграммой (изображение актера должно оставаться выделенным). В левой части страницы выбираем закладку Documentation, а в расположенное справа поле заносим текст описания:

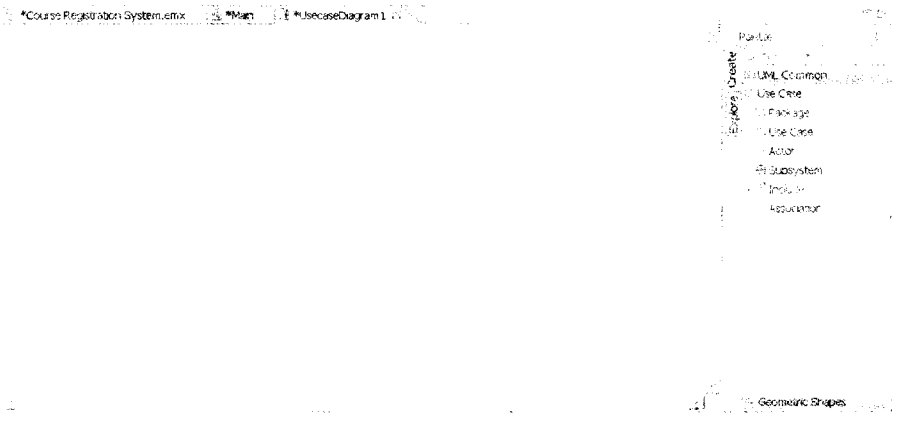


Рис. 19.14. Окно диаграммы Use Case

«Студент — это человек, обучающийся в университете». В результате диаграмма Use Case примет вид, представленный на рис. 19.15.

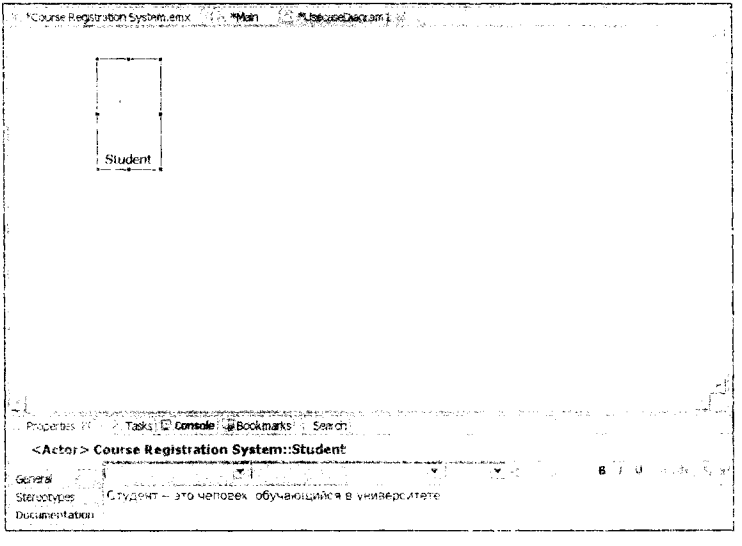


Рис. 19.15. Введение актера Student в диаграмму Use Case

Действуя аналогично, поместим в диаграмму трех других актеров (Professor, Registrar и Billing System — профессор, регистратор, учетная система). Для документирования этих актеров могут использоваться следующие тексты:

- «Профессор — это человек, который читает лекции в университете»
- «Регистратор — это человек, управляющий системой регистрации курсов».
- «Учетная система — это внешняя система, отвечающая за выписку счетов».

На этом этапе диаграмма имеет вид, показанный на рис. 19.16.

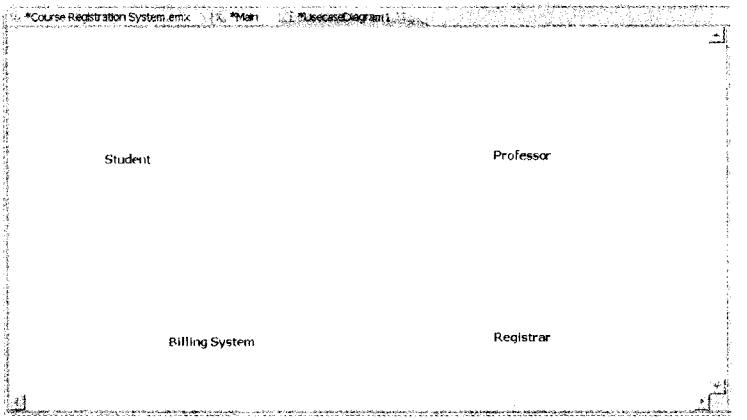


Рис. 19.16. Четыре актера в диаграмме Use Case

Далее для каждого актера нужно определить соответствующие элементы Use Case. Элемент Use Case представляет определенную часть функциональности, обеспечиваемой системой. Элементы Use Case можно идентифицировать путем рассмотрения каждого актера и его взаимодействия с системой. В нашей модели актер Student хочет регистрироваться на курсы (элемент Use Case Register for Courses). Актер Billing System получает информацию о регистрации. Актер Professor хочет запросить список курса (элемент Request Course Roster). Наконец, актер Registrar должен управлять учебным планом (элемент Manage Curriculum).

Процедура введения в диаграмму элементов Use Case и их описаний аналогична рассмотренной для актеров. Так, для элемента Register for Courses нужно:

1. На палитре инструментов щелкнуть по значку элемента Use Case.
2. Для добавления элемента щелкнуть в нужном месте диаграммы Use Case.
3. Пока элемент Use Case остается выделенным, ввести имя Register for Courses.
4. На странице Properties под окном диаграммы выбрать закладку Documentation и в расположенное справа поле внести текст описания: «Запускается студентом. Позволяет создавать, удалять, изменять и/или просматривать расписание студента в указанном семестре. Взаимодействует с учетной системой».

Диаграмма после этих операций показана на рис. 19.17.

Аналогичные действия выполняются для ввода других элементов Use Case (Request Course Roster, Manage Curriculum), при этом первоначальные описания элементов могут иметь следующий вид:

- «Запускается профессором. Позволяет выбирать курсы, которые он будет читать в указанном семестре, и получать расписание занятий» (для элемента Request Course Roster).
- «Запускается регистратором. Позволяет составлять каталог курсов на семестр, управлять информацией об учебных курсах, а также о студентах и преподавателях, работающих с системой» (для элемента Manage Curriculum).

Вид диаграммы после описанных действий представлен на рис. 19.18.

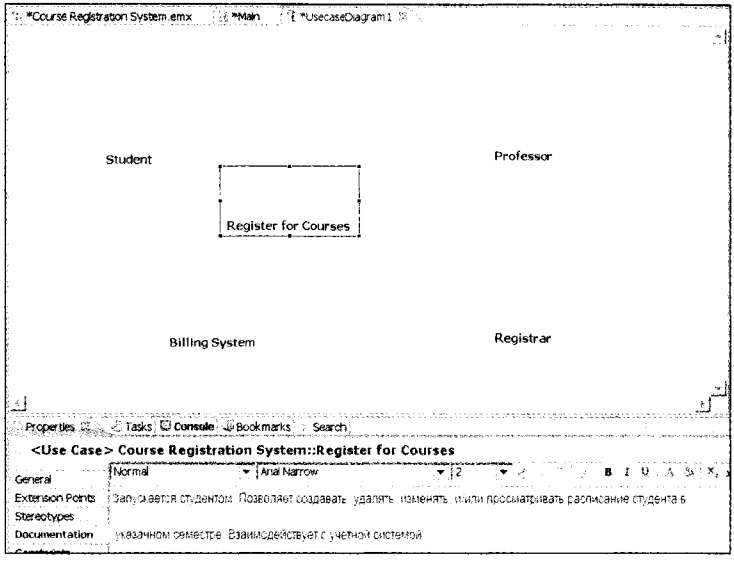


Рис. 19.17. Введение элемента Use Case Register for Courses в диаграмму Use Case

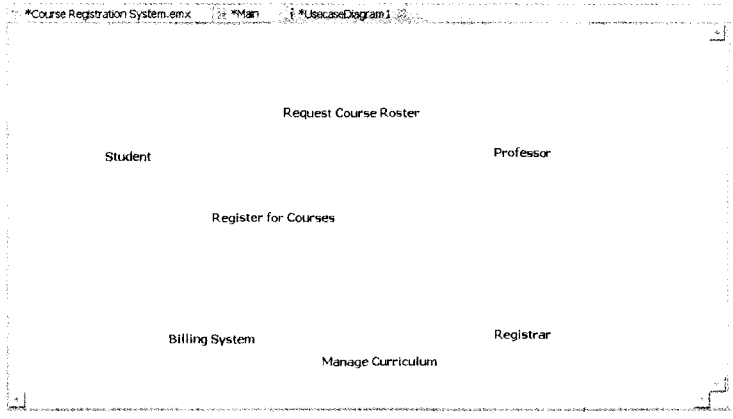


Рис. 19.18. Четыре актера и три элемента Use Case

Далее между актерами и элементами Use Case рисуются отношения. По умолчанию палитра инструментов предлагает ненаправленные ассоциации (значок Association). Направление взаимодействия показывают однонаправленные стрелки. С их помощью задается инициатор взаимодействия. Для включения режима направленных ассоциаций нужно щелкнуть по треугольнику слева от пиктограммы Association, после чего ниже появится значок Directed Association (рис. 19.19). Щелчок по этому значку включает режим направленных ассоциаций. Режим остается активным до возврата (оператором) режима ненаправленных ассоциаций.

В системе регистрации курсов актер Student инициирует элемент Use Case Register for Courses, который, в свою очередь, взаимодействует с актером Billing System. Актер

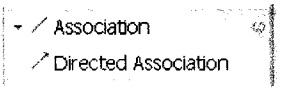


Рис. 19.19. Варианты ассоциаций

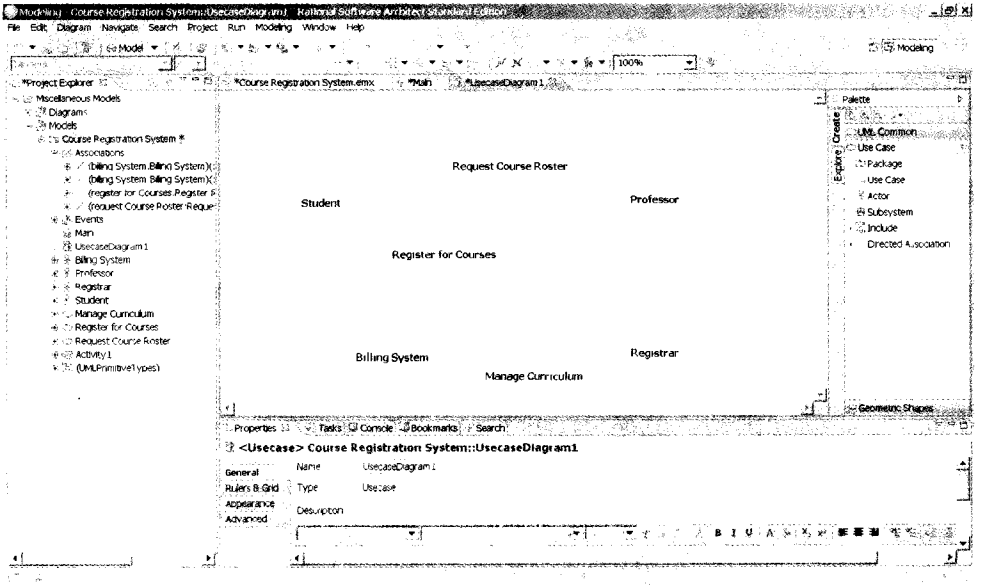


Рис. 19.20. Отношения между актерами и элементами Use Case

Professor инициирует элемент Use Case Request Course Roster. Актер Registrar инициирует элемент Use Case Manage Curriculum (рис. 19.20).

Для внесения отношений в диаграмму Use Case нужно выполнить следующие операции:

1. На палитре инструментов щелкнуть по значку однонаправленной ассоциации (Directed Association).
2. Установить курсор на изображение актера Student, нажать левую кнопку мыши и, удерживая ее, перетащить курсор на элемент Use Case Register for Courses, после чего отпустить кнопку.
3. На панели инструментов щелкнуть по значку однонаправленной ассоциации (Directed Association).
4. Установить курсор на изображение элемента Use Case Register for Courses, нажать левую кнопку мыши и, удерживая ее, перетащить курсор на актера Billing System, после чего отпустить кнопку.
5. Повторить аналогичные действия для ввода других ассоциаций (от актера Professor к элементу Use Case Request Course Roster и от актера Registrar к элементу Use Case Manage Curriculum).

ПРИМЕЧАНИЕ

В ходе установления ассоциации после отпускания кнопки мыши на изображении линии появляется окошко, в которое можно занести название ассоциации.

Обратите внимание, что все введенные актеры и элементы Use Case, а также установленные между ними ассоциации появляются и в браузере проекта (в левой части окна RSA).

Создание диаграммы последовательности

Функциональность элемента Use Case отображается графически в диаграмме последовательности (Sequence diagram). Эта диаграмма показывает упорядоченный по времени поток сообщений между участниками взаимодействия, например при добавлении студента к списку слушателей определенного учебного курса. Иными словами, словесное описание действий в элементе Use Case заменяется набором графических фигур — прямоугольников и стрелок диаграммы последовательности. Для создания диаграммы последовательности нужно определить обобщенные объекты (роли), называемые участниками взаимодействия. Такие обобщенные объекты размещаются в верхней части диаграммы и изображаются прямоугольниками, в которых указаны имя объекта и класса, разделенные двоеточием. Каждый объект имеет собственную временную ось (lifeline) в виде пунктирной линии под его прямоугольником. Сообщения между объектами представляются стрелками, начинающимися на пунктирной линии под отправителем сообщения и завершающимися на пунктирной линии под получателем этого сообщения. Еще одна особенность диаграммы — это спецификация выполнения, изображаемая в виде прямоугольника вытянутого вдоль временной оси. Верхняя грань этого прямоугольника определяет момент начала действия, порождаемого сообщением, а нижняя грань — момент окончания этого действия.

Рассмотрим процесс создания диаграммы последовательности Add a Course для элемента Use Case Register for Courses. Начальные действия при создании диаграммы последовательности напоминают те, что производились при создании диаграммы Use Case, но в данном случае выбираем пункт Sequence Diagram (рис. 19.21).

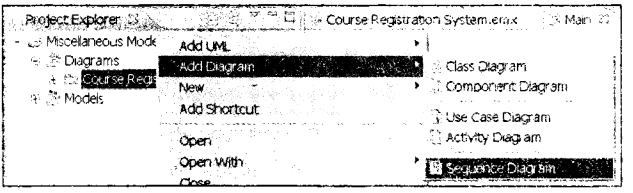


Рис. 19.21. Создание диаграммы последовательности

Появляющееся в браузере имя Interaction1 заменяем на Add a Course.

На экране возникает окно диаграммы последовательности, справа от которого располагается палитра инструментов. Заметим, что палитра теперь содержит иные инструменты, а именно те, которые используются при создании диаграмм последовательности.

Теперь мы будем добавлять в диаграмму такие обобщенные объекты и сообщения, которые реализуют необходимую функциональность. Сценарий, который мы собираемся формализовать с помощью диаграммы последовательности, уже существует — он является фрагментом текста, который содержит спецификация элемента Use Case Register for Courses.

Сценарий инициируется актером Student. Перетащим этого актера из браузера в диаграмму (рис. 19.22) и присвоим студенту имя John:

1. Нажмем левую кнопку мыши на значке актера Student в браузере и перетащим его в диаграмму последовательности, после чего отпустим кнопку.
2. Щелкаем по значку актера в диаграмме последовательности и вводим имя — John.

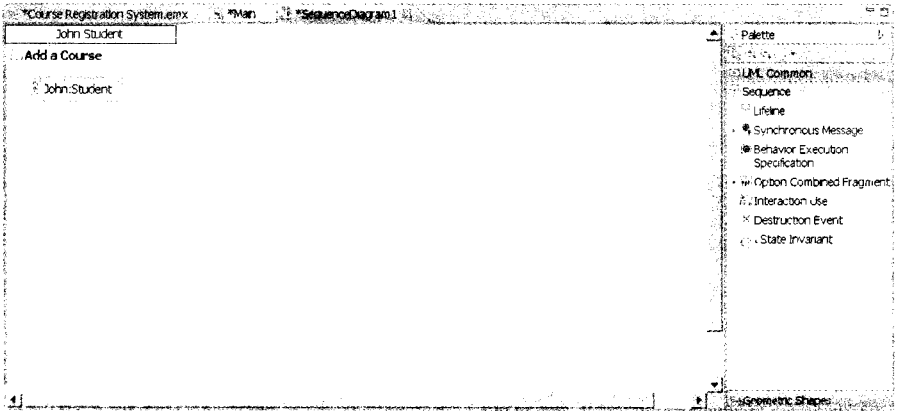


Рис. 19.22. Диаграмма последовательности с актером, инициирующим взаимодействие

В этом сценарии студент должен заполнить информацией (fill in info) регистрационную форму (registration form), после чего форма предъявляется на рассмотрение (submitted). Очевидно, что необходим обобщенный объект, который принимает информацию от студента. Создадим его, присвоим имя registration form и добавим в диаграмму два сообщения «fill in information» и «submit». С этой целью:

1) в палитре инструментов щелкнем по значку Lifeline;
2) щелкнем в окне диаграммы. На экране появляется меню выбора типа обобщенного объекта (рис. 19.23). Так как соответствующий класс мы еще не определяли, то выберем неопределенный тип (Unspecified Type);

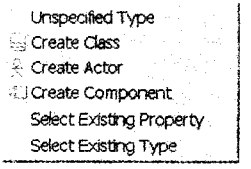


Рис. 19.23. Меню выбора типа обобщенного объекта

3) в появляющийся прямоугольник, пока он остается выделенным, введем имя registration form;

4) непосредственно под значком Lifeline на палитре инструментов располагается пиктограмма выбора типа сообщения. По умолчанию предлагается синхронное сообщение. Положим, нас этот тип не устраивает, поэтому щелкаем по треугольнику слева. В результате откроется меню возможных вариантов (рис. 19.24), в котором выбираем нужный тип (Asynchronous Message);

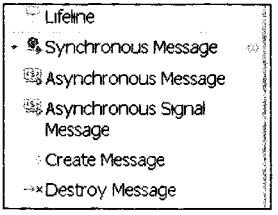


Рис. 19.24. Меню выбора типа сообщения

5) нажмем левую кнопку мыши на пунктирной линии под актером John и, удерживая кнопку нажатой, перетащим стрелку на пунктирную линию под прямоугольником registration form;

6) пока стрелка остается выделенной, введем имя сообщения — fill in information;

7) повторяем шаги 4–6 для создания сообщения submit.

После перечисленных действий диаграмма последовательности приобретет вид, представленный на рис. 19.25.

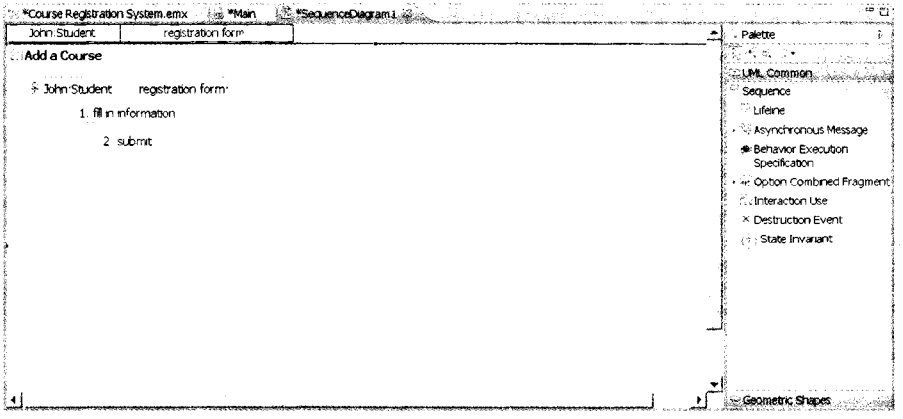


Рис. 19.25. Диаграмма последовательности после добавления registration form

Очевидно, что registration form является промежуточным звеном в цепи передачи информации. Будем считать, что John хочет записаться на курс math 101. Следующее звено — manager, которому registration form должна послать соответствующее сообще-

ние. Чтобы отразить это на диаграмме (рис. 19.26), нужно добавить обобщенный объект manager и передаваемое ему сообщение:

1. На палитре инструментов щелкнем по значку Lifeline.
2. Для добавления обобщенного объекта щелкнем в нужном месте диаграммы.
3. Пока обобщенный объект остается выделенным, введем имя manager.
4. На палитре инструментов щелкнем по значку сообщения (Asynchronous Message).
5. Щелкнем по пунктирной линии под registration form и перетащим стрелку на пунктирную линию под прямоугольником manager.
6. Пока стрелка остается выделенной, введем имя сообщения — add John to math 101.

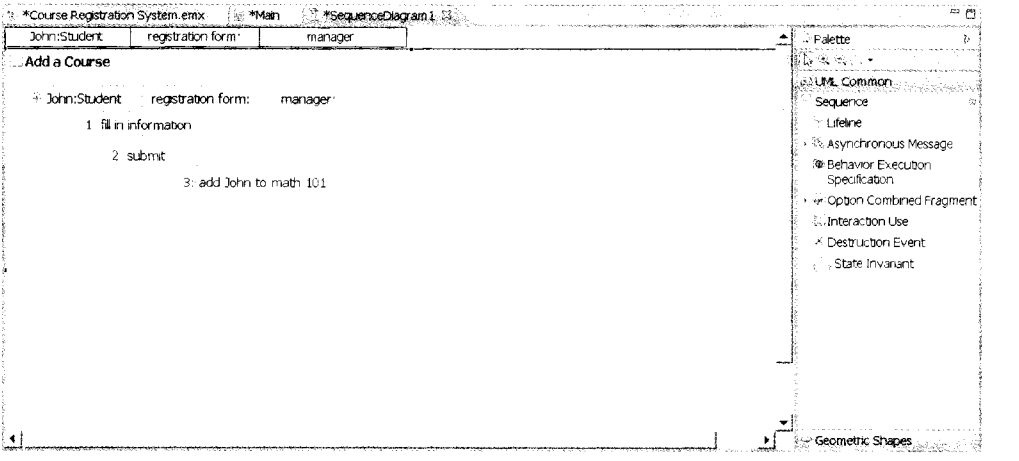


Рис. 19.26. Диаграмма последовательности после добавления manager

Дальнейшее построение диаграммы выполняется аналогичными действиями, поэтому опишем лишь последующие шаги сценария и отобразим диаграмму последовательности после каждого шага.

Обобщенный объект manager взаимодействует как с регистрационной формой, так и с набором учебных курсов. Для обслуживания нашего студента должен существовать обобщенный объект math 101, которому manager обязан передать, что John должен быть добавлен к курсу. Это реализуется с помощью сообщения add John (рис. 19.27).

Обобщенный объект math 101 не принимает самостоятельных решений о возможности добавления студентов. Этим занимается обобщенный объект класса Предложение курса, который назовем section 1. Участник взаимодействия math 101 обращается к section 1 с предложением добавить студента John, используя для этого два сообщения: accepting students? и add John (рис. 19.28).

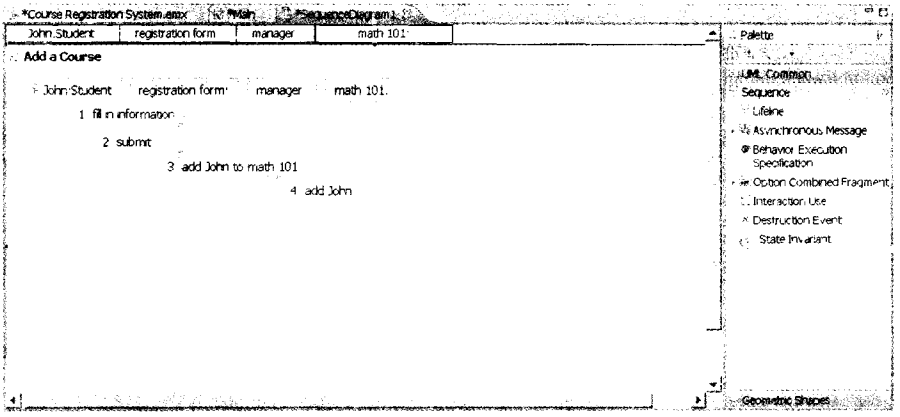


Рис. 19.27. Диаграмма последовательности после добавления math 101

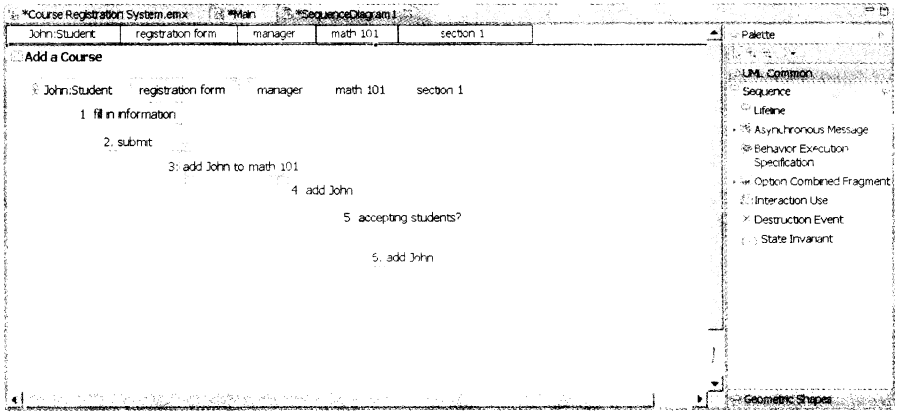


Рис. 19.28. Диаграмма последовательности после добавления section 1

Вопросами оплаты учебы занимается учетная система, а уведомляет ее о необходимости выписки счета manager. После того как последний удостоверился, что студенту John предоставляется возможность изучать курс math 101 (на диаграмме рис. 19.29 это не показано), он уведомляет учетную систему, представленную на диаграмме обобщенным объектом bill (billing system). В результате диаграмма последовательности Add a Course обретет вид, приведенный на рис. 19.29.

На этом предварительный этап создания диаграммы последовательности будем считать завершенным. Дальнейшую коррекцию диаграммы произведем после определения классов, их атрибутов и операций.

Создание диаграммы классов

Объекты из диаграмм последовательности группируются в классы. Основываясь на нашей диаграмме последовательности, мы можем идентифицировать следующие объекты и классы:

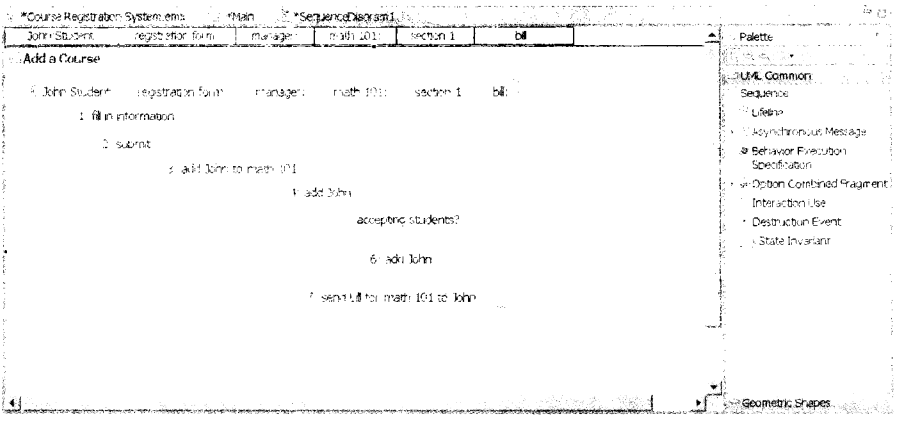


Рис. 19.29. Диаграмма последовательности после добавления bill

- registration form является обобщенным объектом класса RegForm;
- manager является обобщенным объектом класса Manager;
- math 101 является обобщенным объектом класса Course;
- section 1 является обобщенным объектом класса CourseOffering;
- bill является интерфейсом к внешней учетной системе, поэтому мы будем использовать имя BillingSystem как имя его класса.

Начнем с создания этих классов.

1. В браузере сделаем щелчок правой кнопкой мыши по названию нашей модели, а в открывшемся на пункте Add UML подменю выберем позицию Class (рис. 19.30).

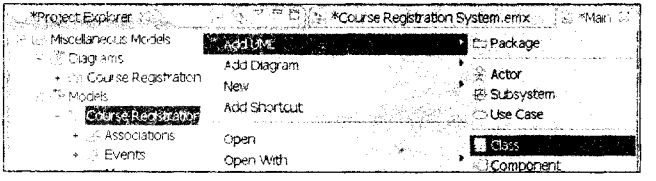


Рис. 19.30. Создание нового класса

2. Пока значок класса, появившийся в браузере, остается выделенным, введем имя класса RegForm.
3. Повторим предыдущие шаги для добавления других классов: Manager, Course, CourseOffering и BillingSystem.

В браузере появляются значки и названия созданных классов (рис. 19.31).

После создания классов они описываются (документируются). Например, у класса может быть такое описание: «CourseOffering — это предлагаемый университетом курс». Порядок добавления описания:

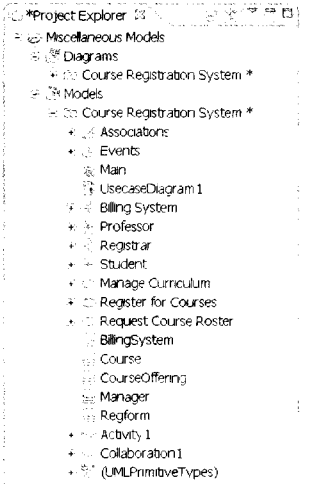


Рис. 19.31. Представление созданных классов в браузере

1. В браузере выделяется класс CourseOffering.
2. В окне свойств класса (оно располагается в нижней части экрана) выбирается страница Properties, а в ней — закладка Documentation.
3. В поле для текста описания вносится требуемый текст (рис. 19.32).

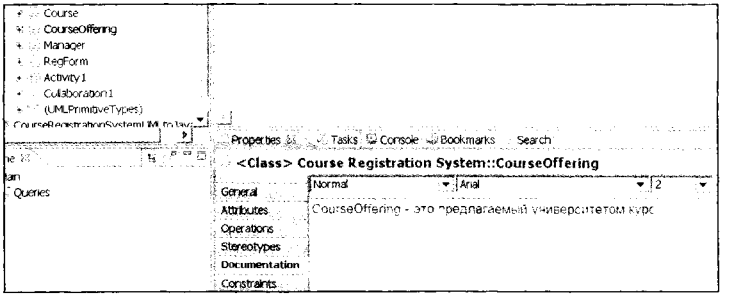


Рис. 19.32. Пример документирования класса

Следующий шаг — построение диаграммы классов. Открытие диаграммы классов производится аналогично открытию двух предыдущих диаграмм (рис. 19.33).

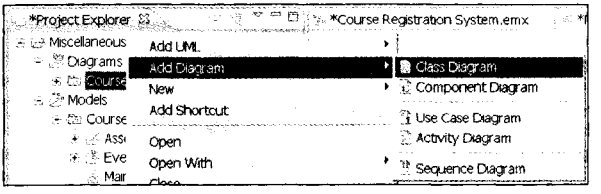


Рис. 19.33. Создание диаграммы классов

На экране появится окно диаграммы классов, а слева от него — соответствующая палитра инструментов. Теперь перенесем классы из браузера в диаграмму классов:

1. Удерживая нажатой клавишу Ctrl, последовательно отметим в браузере требуемые классы, после чего перетащим их в окно диаграммы.
2. Переупорядочим классы в диаграмме (выделяя конкретный класс и перетаскивая его на новое место).

ПРИМЕЧАНИЕ

Классы можно добавлять в диаграмму перетаскиванием их из окна браузера по одному.

На данном этапе диаграмма классов имеет вид, показанный на рис. 19.34.

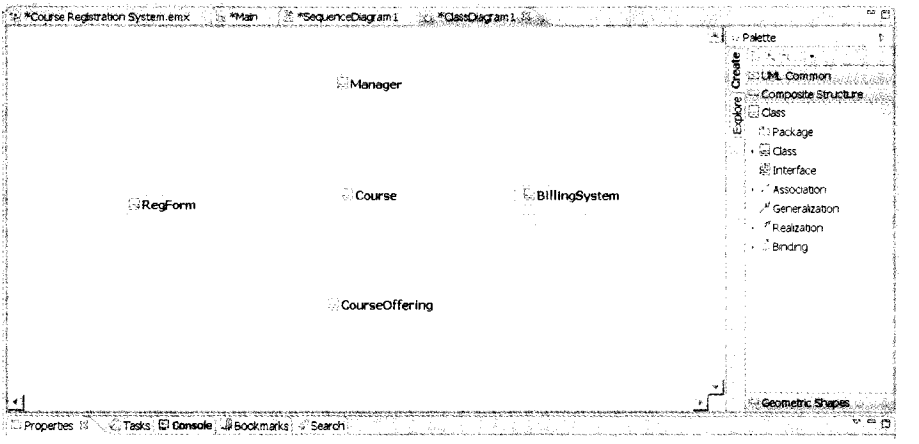


Рис. 19.34. Окно и палитра инструментов диаграммы классов

Для создания новых типов моделирующих элементов в UML используется понятие стереотипа. С помощью стереотипа можно «нагрузить» элемент новым смыслом. Используем предопределенный стереотип Interface для класса BillingSystem, так как этот класс определяет только интерфейс к внешней учетной системе (billing system).

1. Выделим на диаграмме класс BillingSystem.
2. На странице Properties выбираем закладку Stereotypes и в поле Keywords заносим слово-стереотип Interface.

Обратим внимание на то, что новый стереотип класса отразится в виде соответствующей надписи на изображении класса в диаграмме (рис. 19.35).

Для определения взаимодействия объектов нужно указать отношения между соответствующими классами. Для того чтобы увидеть, как объекты должны «разговаривать» друг с другом, исследуются диаграммы последовательности. Если объекты должны «разговаривать», то должен быть путь для коммуникации между их классами. Двумя типами структурных отношений являются ассоциации и агрегации.

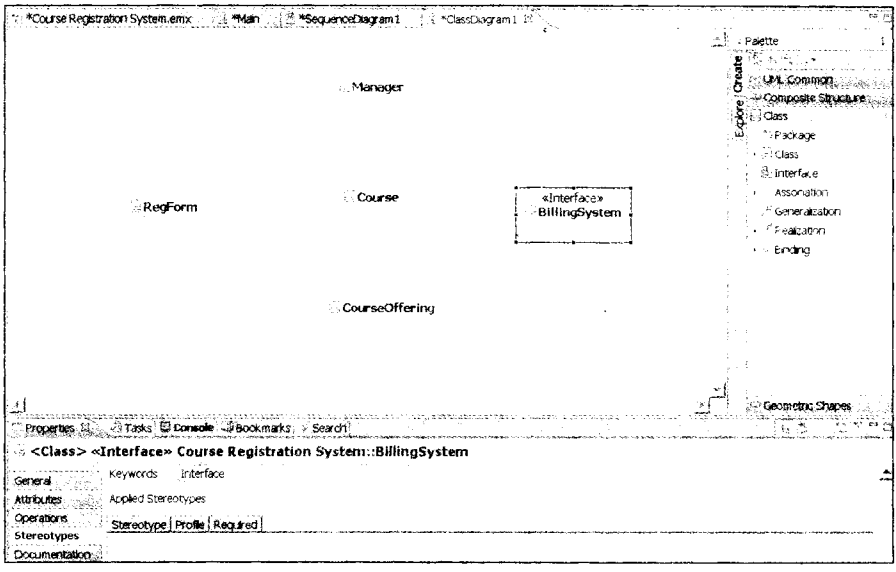


Рис. 19.35. Класс Billing System после введения стереотипа Interface

Ассоциация определяет соединение между классами. Исследуя диаграмму последовательности Add a Course, мы можем определить существование следующих ассоциаций: от RegForm к Manager, от Manager к Course и от Manager к BillingSystem (рис. 19.36).

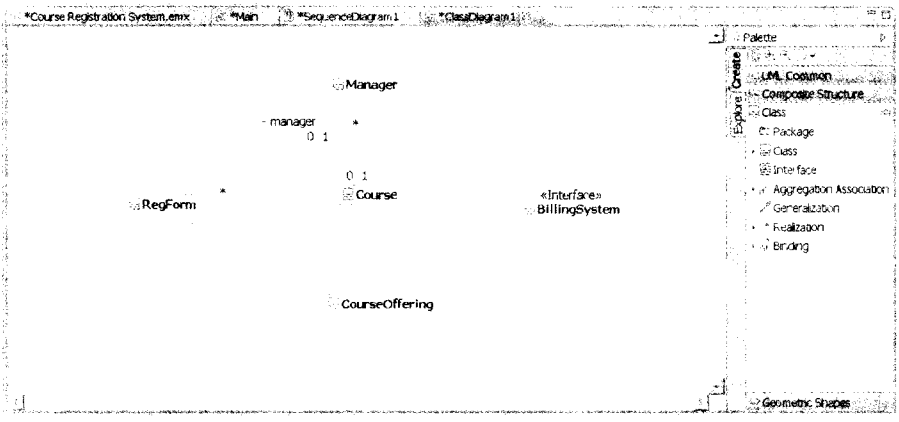


Рис. 19.36. Ассоциации между классами

1. На палитре инструментов располагается значок ненаправленной ассоциации. Поскольку этот тип ассоциации нас не устраивает, щелкнем по треугольнику слева. В результате откроется меню вариантов (рис. 19.37), в котором выбираем нужную направленную ассоциацию (Directed Association).

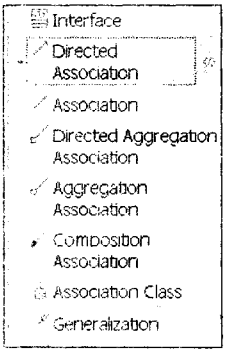


Рис. 19.37. Варианты ассоциаций между классами

2. Щелкнем по классу RegForm и перетащим линию ассоциации на класс Manager.
3. Повторим предыдущие шаги для ввода следующих отношений:

- от Manager к Course;
- от Manager к BillingSystem.

Ассоциации задают пути между объектами-партнерами одинакового уровня.

Агрегация фиксирует неравноправные связи. Она показывает отношение между целым и его частями. Создадим отношение агрегации между классом Course и классом CourseOffering (рис. 19.38), так как предложение курса CourseOfferings является частью агрегата — класса Course.

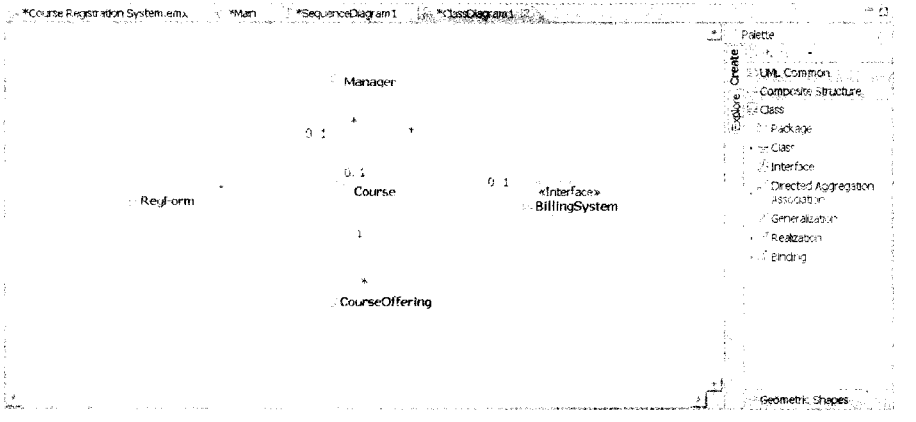


Рис. 19.38. Отношение агрегации

1. На палитре инструментов выберем значок направленной агрегации (Directed Agregation Association).
2. Щелкнем по классу, представляющему целое — Course.
3. Перетащим линию агрегации на класс CourseOffering, представляющий часть.

Объем информации, характеризующей стрелки отношений, может быть различным, как это видно на рис. 19.36, 19.38. Желательная степень подробности задается пунктом Filters контекстного меню, открывающегося при щелчке правой кнопкой по линии отношения (рис. 19.39).

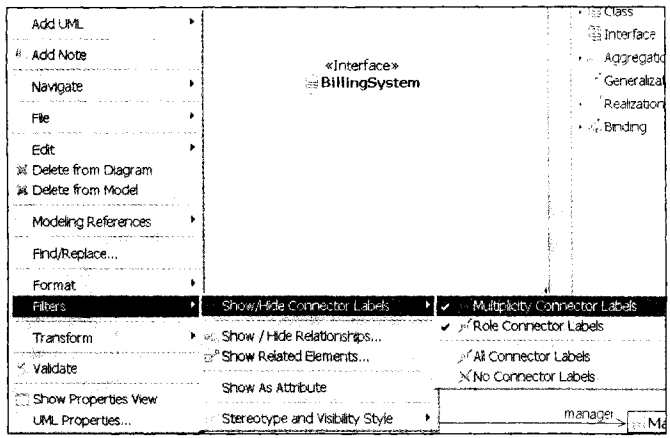


Рис. 19.39. Выбор степени подробности информации об отношении

Для отображения того, «как много» объектов участвует в отношении, к ассоциациям и агрегациям диаграммы могут добавляться индикаторы мощности. Порядок добавления индикаторов мощности:

1. Щелкнуть левой кнопкой по линии агрегации между классами Course и CourseOffering.
2. На странице Properties под окном диаграммы (рис. 19.40) выбрать закладку General.

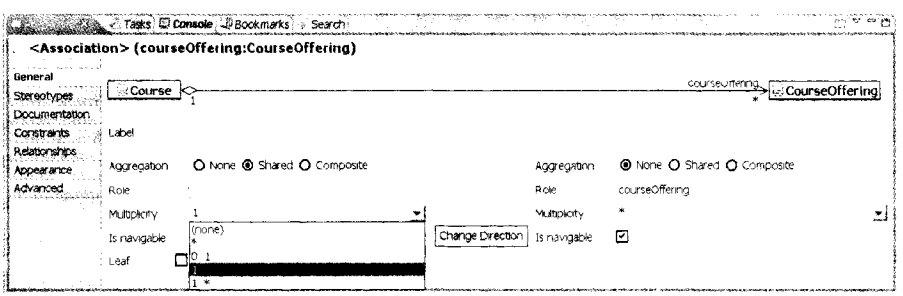


Рис. 19.40. Индикаторы мощности

3. В левом раскрывающемся списке Multiplicity задать мощность на левом полюсе агрегации.
4. В правом раскрывающемся списке Multiplicity задать мощность на правом полюсе агрегации.

5. Заметим, что в этой закладке можно задавать и другие параметры. Например, роли для каждого полюса.

Вспомним, что задание имени — это первый из трех шагов определения класса. Любой класс должен инкапсулировать в себе структуру данных и поведение, которое определяет возможности обработки этой структуры. Примем, что на фиксацию структуры ориентируется второй шаг, а на фиксацию поведения — третий шаг.

Структура класса представляется набором его атрибутов. Структура находится путем исследования предметных требований и соглашений между разработчиками и заказчиками. В нашей модели каждое предложение курса (CourseOffering) является атрибутом (attribute) класса-агрегата Course.

Конечно, класс CourseOffering тоже имеет атрибуты. Определим один из них — количество студентов (numberStudents):

1. В диаграмме классов щелкнем по классу CourseOffering.
2. В появляющемся графическом контекстном меню (рис. 19.41) выберем левый значок (Add Attribute). Это приведет к добавлению в класс атрибута.

Рис. 19.41. Добавление атрибута

3. Пока новый атрибут остается выделенным, введем его имя — numberStudents. Введенный атрибут появится в изображении класса на диаграмме (рис. 19.42).

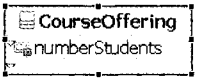


Рис. 19.42. Класс CourseOffering с добавленным атрибутом numberStudents

Итак, два шага формирования класса сделаны. Перейдем к третьему шагу — заданию поведения класса.

Поведение класса представляется набором его операций. Исходная информация об операциях класса находится в диаграммах последовательности. В операции отображаются сообщения из диаграмм последовательности. При этом обычно сообщения переименовываются — производится согласование имени сообщения и имени операции Причины переименования просты и понятны. Во-первых, имя операции должно отражать ее принадлежность к классу (а не к источнику соответствующего сообщения). Во-вторых, имя операции должно указывать на ее обязанность, а не на способ ее реализации. В-третьих, имя должно быть допустимым с точки зрения синтаксиса языка программирования, который будет использоваться для кодирования класса.

Для примера наполним операциями класс CourseOffering. Как видно из диаграммы последовательности, классу присущи две операции, представленные сообщениями add John и accepting students?. Сначала создадим операцию, соответствующую первому из этих сообщений, присвоив ей имя add. Процесс похож на наполнение класса атрибутами.

1. В диаграмме классов щелкнем по классу CourseOffering.
2. В появляющемся графическом контекстном меню (рис. 19.43) выберем правый значок (Add Operation). Это приведет к добавлению в класс операции.

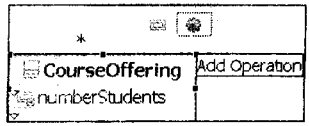


Рис. 19.43. Добавление операции

3. Пока новая операция остается выделенной, вводим ее имя — add.
4. Действуя аналогично, дополним класс второй операцией, представляющей сообщение accepting students?, присвоив этой операции имя offeringOpen.

Теперь изображение класса CourseOffering примет следующий вид (рис. 19.44).

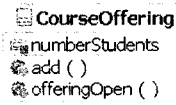


Рис. 19.44. Класс CourseOffering с добавленными атрибутом и операциями

Теперь мы можем привязать обобщенные объекты из диаграммы последовательности к конкретным классам.

Откроем диаграмму последовательности. Будем поочередно перетаскивать значки классов (из браузера проектов) на прямоугольники соответствующих обобщенных объектов. Результат перетаскивания демонстрирует рис. 19.45.

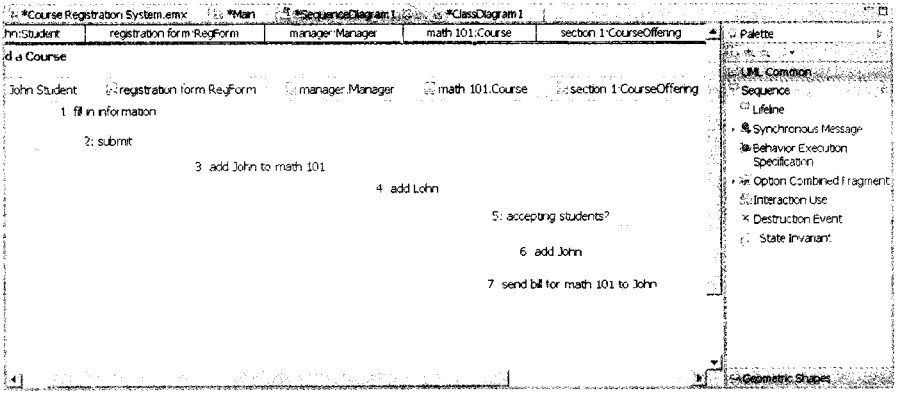


Рис. 19.45. Привязка обобщенных объектов к классам

Видим, что имя каждого обобщенного объекта удлинилось, в нем появились две части, разделенные двоеточием. Слева от двоеточия записывается собственное имя объекта, а справа — имя класса.

В заключение согласуем имена сообщений и операций классов. Продемонстрируем согласование на примере сообщения accepting students?.

1. Выполним двойной щелчок по имени сообщения accepting students?. Появится диалоговое окошко, предлагающее два варианта: создать новую операцию (Create New Operation), выбрать существующую операцию (Select Existing Operation). Для варианта «существующей операции» указаны две операции класса CourseOffering.

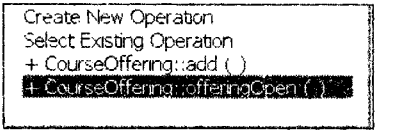


Рис. 19.46. Привязка к операции

2. Как было определено ранее, этому сообщению соответствует операция offeringOpen() класса CourseOffering (последняя строка в диалоговом окошке на рис. 19.46). Выполним щелчок по этой строке. Имя сообщения на диаграмме последовательности заменяется именем операции (рис. 19.47).

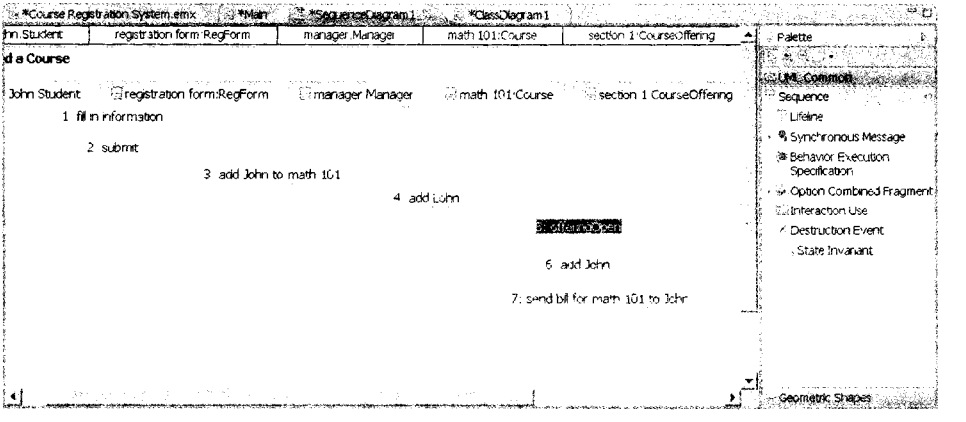


Рис. 19.47. Диаграмма последовательности после привязки к операции класса

Аналогично поступаем с остальными сообщениями в диаграмме последовательности.

Генерация программного кода

Прежде чем станет возможным преобразование UML-модели в программный код, нужно создать так называемую *конфигурацию трансформации* (transformation configuration). Она включает информацию, используемую в процессе трансформации: уникальное имя, источник и целевой объект трансформации, а также ряд свойств, специфичных для данного вида трансформации.

В нашем примере мы рассмотрим трансформацию разработанной модели UML в программный код на языке Java, в результате чего элементы модели отображаются в текст Java-классов. Генерацию программного кода начнем с создания конфигурации трансформации.

1. Выбираем в браузере проектов модель Course Registration System, щелкнув по ней правой кнопкой мыши.
2. В открывшемся меню выбираем пункт New, а в последующем подменю — пункт Other (рис. 19.48).

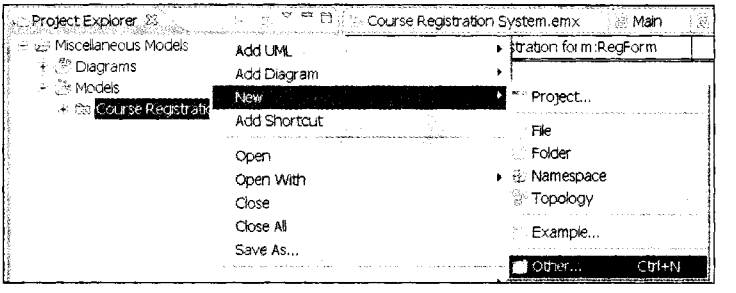


Рис. 19.48. Переход к созданию конфигурации трансформации

3. В появляющемся на экране окне Select a Wizard выбираем пункт Transformations > Transformation Configuration (рис. 19.49) и делаем щелчок по Next.

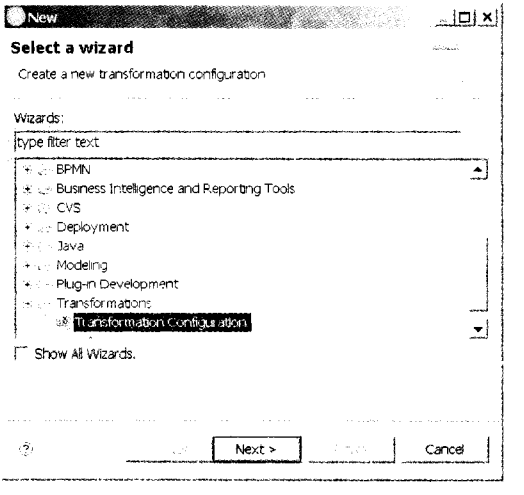


Рис. 19.49. Первый шаг создания конфигурации трансформации

4. В появившемся окне Specify a Configuration Name and Transformation (рис. 19.50) в списке Transformation выбираем Java Transformations > UML to Java, а в поле Name заносим имя конфигурации трансформации CourseRegistrationSystemUMLtoJava. Делаем щелчок по Next.

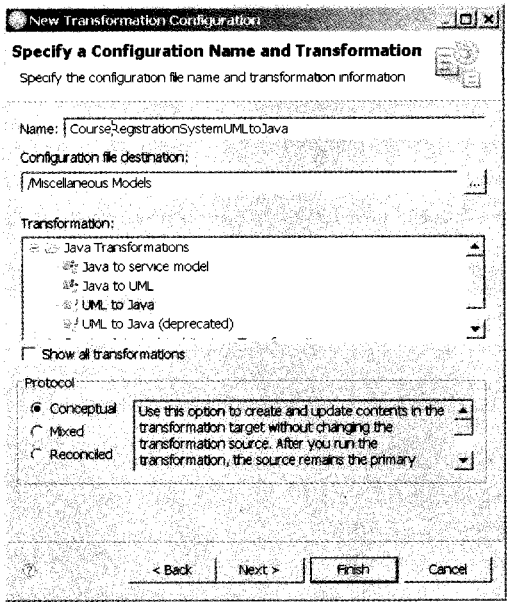


Рис. 19.50. Определение имени и типа конфигурации трансформации

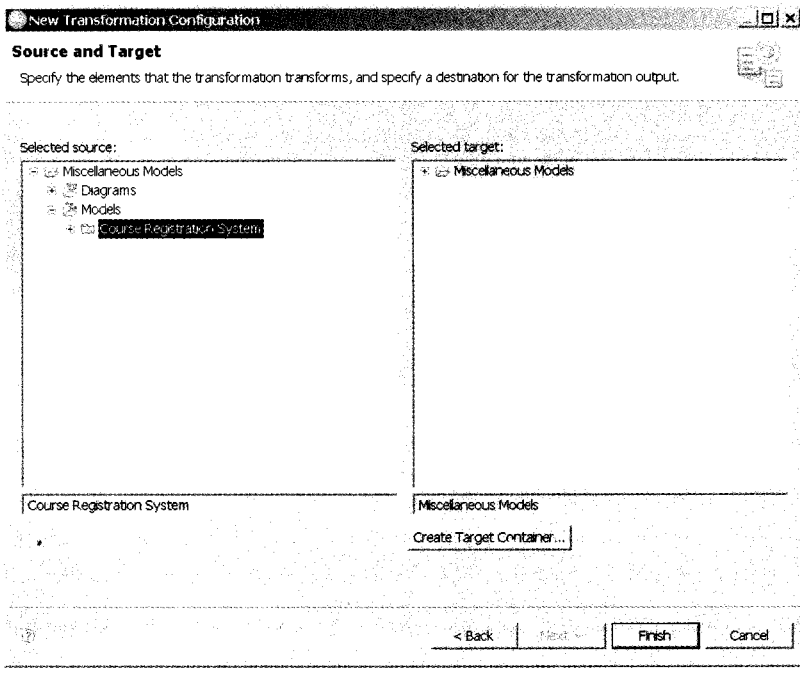


Рис. 19.51. Определение преобразуемой модели UML и местоположения конфигурации трансформации

5. В появляющемся окне Source and Target (рис. 19.51) в качестве источника выбираем нашу модель UML Course Registration Model. Для хранения конфигурации трансформации создадим контейнер, для чего щелкнем по кнопке Create Target Container... .
6. В очередном окне (Create Java Project) указываем имя создаваемого Java-проекта (назовем его CourseRegisterSystemJavaProject) и щелкаем по Finish (рис. 19.52). После возврата к окну Source and Target также щелкаем по Finish.

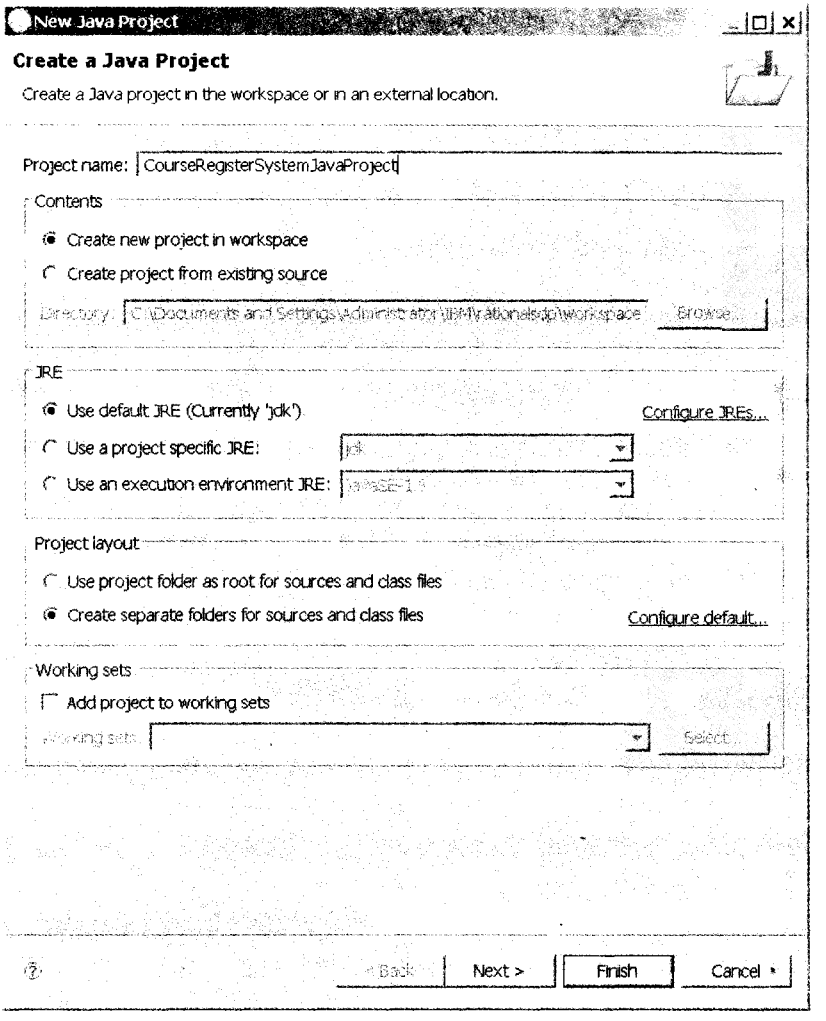


Рис. 19.52. Завершение формирования конфигурации трансформации

Мы создали конфигурацию трансформации, которая имеет название, а также CourseRegisterSystemJavaProject, но файлы Java еще не сгенерированы. Поэтому следующий шаг — это преобразование модели UML в Java-код. После выполненных шагов изображение на экране соответствует рис. 19.53.

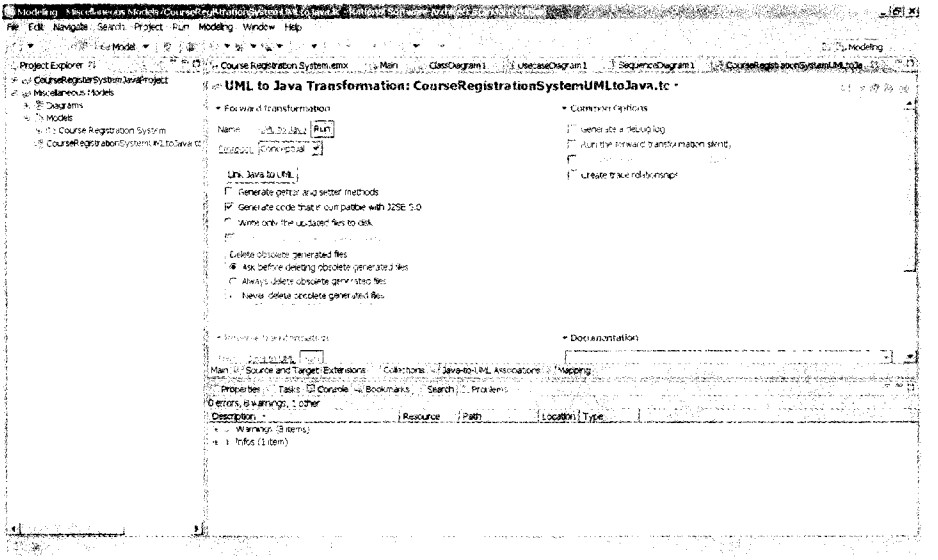


Рис. 19.53. Экран RSA после завершения создания конфигурации трансформации

Для выполнения трансформации (рис. 19.54):

1) сделаем правый щелчок по названию файла конфигурации CourseRegistration-SystemUMLtoJava.tc в браузере проектов;
2) в появившемся меню выберем пункт Transforn > UML to Java.

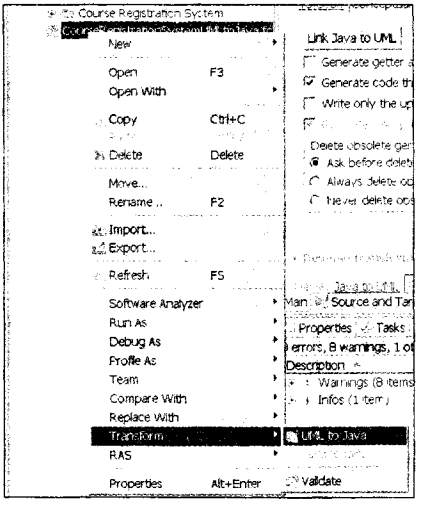


Рис. 19.54. Переход к операции трансформации

В результате будет сгенерирован код Java, который можно посмотреть в проекте Java. На рис. 19.55 показан код Java, соответствующий классу CourseOffering из модели UML.

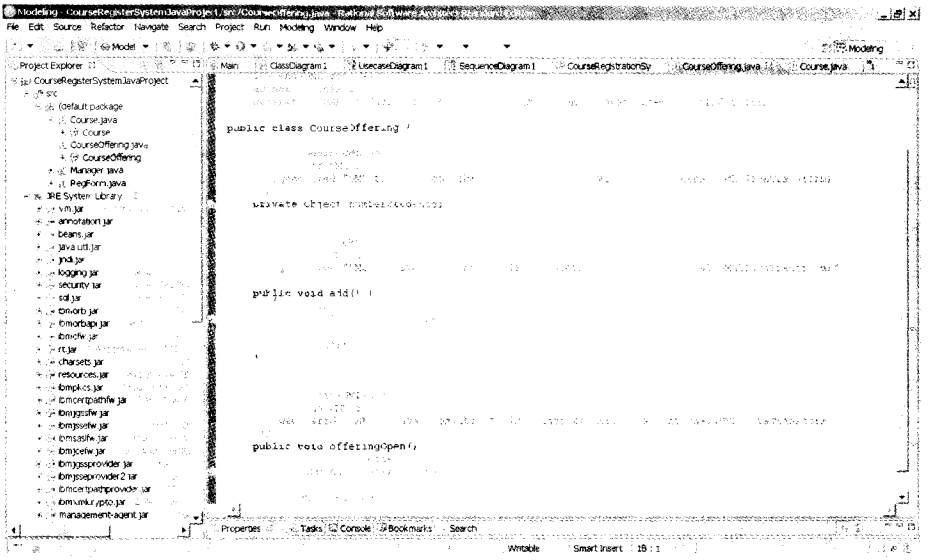


Рис. 19.55. Сгенерированный код Java для класса CourseOffering

В процессе генерации RSA отображает логическое описание класса в каркас программного кода — в коде появляются языковые описания имени класса, атрибутов класса и заголовки методов. Кроме того, для описания тела каждого метода формируется программная заготовка. Появляются и программные связи классов. Подразумевается, что программист будет дополнять этот код, работая в конкретной среде программирования. В качестве примера в листинге 19.1 приведен сгенерированный каркас кода для класса CourseOffering.

Листинг 19.1. Программный код для класса CourseOffering

```
public class CourseOffering {
    private Object numberStudents;
     * <!-- begin-UML-doc -->
     * <!-- end-UML-doc -->
    public void add() {
        // begin-user-code
        // TODO Auto-generated method stub
        // end-user-code
    }
    public void offeringOpen() {
        // begin-user-code
        // TODO Auto-generated method stub
        // end-user-code
    }
}
```

Видим, что в коде класса CourseOffering предусмотрен один приватный атрибут и два публичных метода.

Для поддержки однонаправленных ассоциаций RSA создает атрибут в классе-источнике сообщения. Например, в рассмотренной модели предусмотрена направленная ассоциация из класса RegForm в класс Manager, поэтому в коде класса RegForm обнаруживаем приватный атрибут manager:

```
public class RegForm {

    private Manager manager;
}
```

От класса Manager поддержка этой ассоциации не требуется.

Трансформация программного кода в модель UML

Теперь выполним обратное преобразование Java-кода в модель UML, внеся небольшое изменение в текст для класса CourseOffering.

Например, добавим параметр и определим тип возвращаемого значения для метода add() (новый заголовок примет вид public int add(int a)), а также введем новый метод public void sub(int b).

Сформируем новую конфигурацию трансформации.

1. Выбираем в браузере проектов пункт CourseRegisterJavaProject, щелкнув по нему правой кнопкой мыши.
2. В открывшемся меню выбираем пункт New, а в последующем подменю — пункт Other.
3. В появляющемся на экране окне Select a Wizard выбираем пункт Transformations > Transformation Configuration и делаем щелчок по Next.
4. В появившемся окне Specify a Configuration Name and Transformation в списке Transformation выбираем Java Transformations > Java to UML, а в поле Name заносим имя конфигурации трансформации CourseRegistrationSystemJavatoUML. Делаем щелчок по Next. Следствием наших действий становится появление окна Source and Target.
5. В окне Source and Target в качестве источника выбираем CourseRegisterJavaProject. Для хранения формируемой конфигурации создадим контейнер, для чего щелкнем на кнопке Create Target Container... .
6. В появляющемся окне Create Model выбираем Standard Template (стандартный шаблон) и щелкаем по Next.
7. Выводится следующее диалоговое окно с тем же названием Create Model. Оно позволяет выбрать категорию создаваемой модели (раздел Categories) и один из шаблонов, предлагаемых для выбранной категории (раздел Templates). Для нашего примера выберем категорию Analysis and Design и шаблон Blank Analysis Package (пустой пакет анализа). В поле File name наберем название файла с нашей моделью Course Registration System UML Reverse Model.
8. Щелкнем по кнопке Finish. В итоге вернемся к окну Source and Target, которое теперь будет иметь вид, представленный на рис. 19.56.
9. В разделе Select Target выберем пункт Course Registration System UML Reverse Model и щелкнем по Finish. В результате создается файл CourseRegistrationSystemJavatoUML.tc, который можно увидеть в окне браузера проектов.

После создания конфигурации приступим к самой трансформации:

1. Сделаем правый щелчок по названию файла конфигурации CourseRegistrationSystemJavatoUML.tc в браузере проектов.
2. В появившемся меню выберем пункт Transforn > Java to UML.

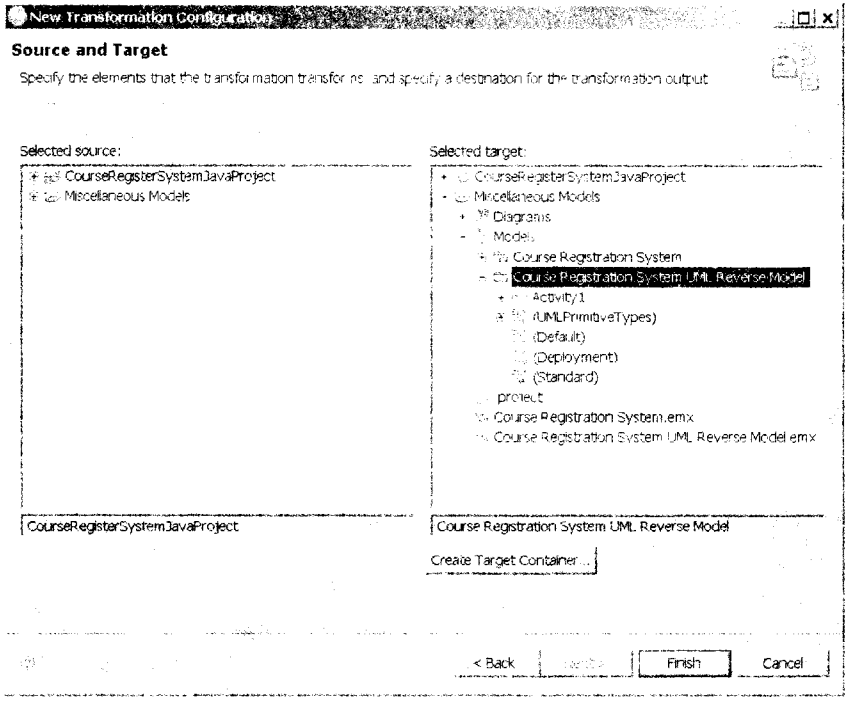


Рис. 19.56. Вид окна Source and Target после описанных действий

3. На экране появляется окно Merge trasnformed model. К этому моменту сформирована промежуточная модель. В левом разделе окна показаны изменения, которые предлагается перенести в целевую модель UML. Убирая галочки в соответствующих позициях левого раздела, можно отказаться от переноса отдельных изменений.
4. Соглашаемся на полный перенос (все галочки установлены) и щелкаем по OK.

В результате наших действий создана модель Course Registration System UML Reverse Model, что подтверждает строка в браузере проектов (рис. 19.57).

Как видно, изменения Java-кода отражены в модели UML. Об этом свидетельствует и модифицированное представление класса (рис. 19.58).

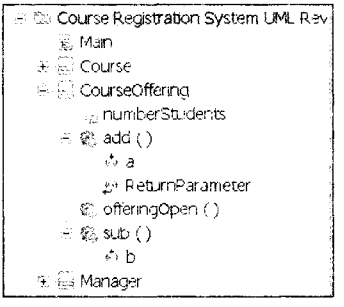


Рис. 19.57. Модель UML, созданная в результате трансформации

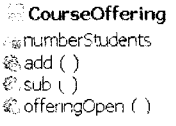


Рис. 19.58. Модифицированный класс CourseOffering

Заключение

Современная программная инженерия (Software Engineering) – молодая и быстро развивающая область знаний и практик. Она ориентирована на комплексное решение задач, связанных с разработкой особой разновидности сложных систем – программных систем.

Программные системы – самые необычные и удивительные создания рук человеческих. Они не имеют физических тел, их нельзя потрогать, ощутить одним из человеческих чувств. Они не подвергаются физическому износу, их нельзя изготовить в обычном инженерном смысле, как автомобиль на заводе. И вместе с тем разработка программных систем является самой сложной из задач, которые приходилось когда-либо решать человеку-инженеру. В пределе это задача создания рукотворной системы, сопоставимой по сложности самому творцу.

Многие стереотипы и приемы, разработанные в физическом мире, оказались неприменимы к инженерии программных систем. Приходилось многое изобретать и придумывать. Все это теперь история. Программные инженеры начинали с полного неприятия инструментария других инженерных дисциплин, уверовав в свою кастовость «жрецов в белых халатах», и совершили эволюционный круг, вернувшись в лоно общечеловеческой инженерии.

Впрочем, были времена, когда и другие «кланы» людей относились к «программерам» с большим подозрением: им мало платили, унижая материально, не находили для них ласковых слов (а употребляли, большей частью, ругательные). Где эти люди? И где их прежнее отношение?

Современное общество впадает во все большую зависимость от программных технологий. Программного инженера стали любить, охотно приглашать в гости, хорошо кормить, обувать и одевать. Словом, стали лелеять и холить (правда, время от времени продолжают сжигать на костре и предавать анафеме).

Современная программная инженерия почти достигла уровня зрелости – об этом свидетельствуют современные тенденции – она разворачивается от сердитого отношения к своим разработчикам к дружелюбному, снисходительному пониманию человеческих слабостей.

Базис современной программной инженерии образуют следующие составляющие:

- процессы разработки ПО;
- метрический аппарат, обеспечивающий измерения процессов и продуктов;
- аппарат формирования исходных требований к разработкам;
- аппарат анализа и проектирования ПО;

- аппарат визуального моделирования ПО;
- аппарат тестирования программных продуктов.

Все эти составляющие рассмотрены в данном учебнике. Конечно, многое осталось за кадром. Особенности разработки веб-приложений, сервис-ориентированная архитектура, наследуемые системы, фабрики программного обеспечения, облачные вычисления, реинжиниринг систем — вот неполный перечень тем, обсудить которые не позволили ресурсные ограничения. Хотелось бы обратить внимание на новейшие пост-объектные методологии — порождающее, ментальное и многомерное проектирование и программирование. Они представляют собой новую высоту стремительного полета в компьютерный космос. Но это — тема следующей работы. Впереди длинная и интересная дорога познаний. Как хочется подольше шагать по этой дороге!

В заключение «родился теплый лирический тост» — за программистов всех стран! А если серьезно, друзья, вы — строители целого виртуального мира, мы верим в вас, мы гордимся вами. Дерзайте, творите, разочаровывайтесь и очаровывайтесь! Уверены, вы построите достойное информационное обеспечение человеческого общества!

Приложение А

Факторы затрат пост-архитектурной модели COCOMO II

Значительную часть времени при использовании модели COCOMO II занимает работа с факторами затрат. Это приложение содержит описание таблиц Боэма, обеспечивающих оценку факторов затрат.

Факторы продукта

Таблица А.1. Требуемая надежность ПО (Required Software Reliability) RELY

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
RELY	Легкое беспокойство	Низкая, легко восстанавливаемые потери	Умеренная, легко восстанавливаемые потери	Высокая, финансовые потери	Риск для человеческой жизни	

Таблица А.2. Размер базы данных (Data Base Size) DATA

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
DATA		Байты БД/ LOCпрогр.< 10	$10 \leq D/P < 100$	$100 \leq D/P < 1000$	$D/P \geq 1000$	

ПРИМЕЧАНИЕ

Фактор DATA определяется делением размера БД (D) на длину кода программы (P). Длина программы представляется в LOC-оценках.

Сложность продукта (Product Complexity) CPLX

Сложность продукта определяют по двум следующим таблицам. Выделяют 5 областей применения продукта: операции управления, вычислительные операции,

операции с приборами (устройствами), операции управления данными, операции управления пользовательским интерфейсом. Выбирается область или комбинация областей, которые характеризуют продукт или подсистему продукта. Сложность рассматривается как взвешенное среднее значение для этих областей.

Таблица А.3. Сложность модуля в зависимости от области применения

CPLX	Операции управления	Вычислительные операции	Операции с приборами
Очень низкий	Последовательный код с небольшим количеством структурированных операторов: DO, CASE, IF-THEN-ELSE. Простая композиция модулей с помощью вызовов процедур и простых сценариев	Вычисление простых выражений, например A = B + C*(D - E)	Простые операторы чтения и записи, использующие простые форматы
Низкий	Несложная вложенность структурированных операторов. В основном простые предикаты	Вычисление выражений средней сложности, например D = = SQRT(B**2 - - 4*A*C)	Не требуется знание характеристик конкретного процессора или устройства ввода-вывода. Ввод-вывод выполняется на уровне GET/PUT
Номинальный	В основном простая вложенность. Некоторое межмодульное управление. Таблицы решений. Простые обратные вызовы (callbacks) или передачи сообщений, включение среднего уровня — поддержка распределенной обработки	Использование стандартных математических и статистических подпрограмм. Базовые матричные/векторные операции	Обработка ввода-вывода, включающая выбор устройства, проверку состояния и обработку ошибок
Высокий	Высокая вложенность операторов с составными предикатами. Управление очередями и стеками. Однородная распределенная обработка. Управление ПО реального времени на единственном процессоре	Базовый численный анализ: мультивариантная интерполяция, обычные дифференциальные уравнения. Базисное усечение, учет потерь точности	Операции ввода-вывода физического уровня (определение адресов физической памяти; поиски, чтения и т. д.). Оптимизированный совмещенный ввод-вывод
Очень высокий	Реентерабельное и рекурсивное программирование. Обработка прерываний с фиксированными приоритетами. Синхронизация задач, сложные обратные вызовы, гетерогенная распределенная обработка. Управление однопроцессорной системой в реальном времени	Сложный, но структурированный численный анализ: уравнения с плохо обусловленными матрицами, уравнения в частных производных. Простой параллелизм	Процедуры для диагностики по прерыванию, обслуживание и маскирование прерываний. Обслуживание линий связи. Высокопроизводительные встроенные системы
Сверхвысокий	Планирование множественных ресурсов с динамически изменяющимися приоритетами. Управление на уровне микропрограмм. Управление распределенной аппаратурой в реальном времени	Сложный и неструктурированный численный анализ: высокоточный анализ стохастических данных с большим количеством шумов. Сложный параллелизм	Программирование с учетом временных характеристик приборов, микропрограммные операции. Критические к производительности встроенные системы

Таблица А.4. Сложность модуля в зависимости от области применения

CPLX	Операции управления данными	Операции управления пользовательским интерфейсом
Очень низкий	Простые массивы в оперативной памяти. Простые запросы к БД, обновления	Простые входные формы, генераторы отчетов
Низкий	Использование одного файла без изменения структуры данных, без редактирования и промежуточных файлов. Умеренно сложные запросы к БД, обновления	Использование билдеров для простых графических интерфейсов
Номинальный	Ввод из нескольких файлов и вывод в один файл. Простые структурные изменения, простое редактирование. Сложные запросы БД, обновления	Простое использование набора графических объектов (widgets)
Высокий	Простые триггеры, активизируемые содержимым потока данных. Сложное изменение структуры данных	Разработка набора графических объектов, его расширение. Простой голосовой ввод-вывод, мультимедиа
Очень высокий	Координация распределенных БД. Сложные триггеры. Оптимизация поиска	Умеренно сложная 2D/3D-графика, динамическая графика, мультимедиа
Сверхвысокий	Динамические реляционные и объектные структуры с высоким сцеплением. Управление данными с помощью естественного языка	Сложные мультимедиа, виртуальная реальность

Таблица А.5. Требуемая повторная используемость (Required Reusability) RUSE

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
RUSE		Нет	На уровне проекта	На уровне программы	На уровне семейства продуктов	На уровне нескольких семейств продуктов

Таблица А.6. Документирование требований жизненного цикла (Documentation match to life-cycle needs) DOCU

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
DOCU	Многие требования жизненного цикла не учтены	Некоторые требования жизненного цикла не учтены	Оптимизированы к требованиям жизненного цикла	Избыточны по отношению к требованиям жизненного цикла	Очень избыточны по отношению к требованиям жизненного цикла	

Факторы платформы (виртуальной машины)

Таблица А.7. Ограничения времени выполнения (Execution Time Constraint) TIME

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
TIME			Используется $\leq$ 50% возможного времени выполнения	70%	85%	95%

Таблица А.8. Ограничения оперативной памяти (Main Storage Constraint) STOR

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверх высокий
STOR			Используется ≤ 50% доступной памяти	70%	85%	95%

Таблица А.9. Изменчивость платформы (Platform Volatility) PVOL

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
PVOL		Значительные изменения — каждые 12 мес.; незначительные — каждый месяц	Значительные изменения — каждые 6 мес.; незначительные — каждые 2 недели	Значительные изменения — 2 мес.; незначительные — 1 неделя	Значительные изменения — 2 нед.; незначительные — 2 дня	

Факторы персонала

Таблица А.10. Возможности аналитика (Analyst Capability) ACAP

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
ACAP	15%	35%	55%	75%	90%	

Таблица А.11. Возможности программиста (Programmer Capability) PCAP

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
PCAP	15%	35%	55%	75%	90%	

Таблица А.12. Опыт работы с приложением (Applications Experience) AEXP

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
AEXP	2 месяца	6 месяцев	1 год	3 года	6 лет	

Таблица А.13. Опыт работы с платформой (Platform Experience) PEXP

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
PEXP	2 месяца	6 месяцев	1 год	3 года	6 лет	

Таблица А.14. Опыт работы с языком и утилитами (Language and Tool Experience) LTEX

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
LTEX	2 месяца	6 месяцев	1 год	3 года	6 лет	

Таблица А.15. Непрерывность персонала (Personnel Continuity) PCON

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
PCON	48%/год	24%/год	12%/год	6%/год	3%/год	

ПРИМЕЧАНИЕ

С помощью фактора PCON учитывается процент смены персонала.

Факторы проекта

Таблица А.16. Использование программных утилит (Use of Software Tools) TOOL

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
TOOL	Редактирование, кодирование, отладка	Простая входная, выходная CASE-утилита, малая интеграция	Базовые утилиты жизненного цикла, умеренная интеграция	Развитые утилиты жизненного цикла, умеренная интеграция	Развитые утилиты жизненного цикла, хорошо интегрированные с процессами, методами, повторным использованием	

Таблица А.17. Мультисетевая разработка (Multisite Development) SITE

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
SITE: коммуникации	Один телефон, почта	Индивидуальные телефоны. FAX	Узкополосный e-mail	Широкополосные электронные коммуникации	Широкополосные электронные коммуникации, видеоконференции от случая к случаю	Интерактивные мультимедиа

Таблица А.18. Требуемый график разработки (Required Development Schedule) SCED

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
SCED	75% от номинального срока	85%	100%	130%	160%	

Таблица А.19. Числовые значения множителей затрат

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
RELY	Легкое беспокойство 0,75	Низкая, легко восстанавливаемые потери 0,88	Умеренная, легко восстанавливаемые потери 1,00	Высокая, финансовые потери 1,15	Риск для человеческой жизни 1,39	

продолжение

Таблица А.19 (*продолжение*)

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
DATA		Байты БД/LOC прогр. <10 0,93	10 ≤ D/P <100 1,00	100 ≤ D/P <1000 1,09	D/P ≥ 1000 1,19	
CPLX	0,75	0,88	1,00	1,15	1,30	1,66
RUSE		Нет 0,91	На уровне проекта 1,00	На уровне программы 1,14	На уровне семейства продуктов 1,29	На уровне нескольких семейств продуктов 1,49
DOCU	Многие требования жизненного цикла не учтены 0,89	Некоторые требования жизненного цикла не учтены 0,95	Оптимизированы к требованиям жизненного цикла 1,00	Избыточны по отношению к требованиям жизненного цикла 1,06	Очень избыточны по отношению к требованиям жизненного цикла 1,13	
TIME			Используется ≤ 50% возможного времени выполнения 1,00	70% 1,11	85% 1,31	95% 1,67
STOR			Используется ≤ 50% доступной памяти 1,00	70% 1,06	85% 1,21	95% 1,57
PVOL		Значительные изменения – через 1 год; незначительные – через 1 мес. 0,87	Значительные – через 6 мес.; незначительные – через 2 недели 1,00	Значительные – через 2 мес.; незначительные – через 1 неделю 1,15	Значительные – через 2 нед.; незначительные – через 2 дня 1,30	
ACAP	15% 1,50	35% 1,22	55% 1,00	75% 0,83	90% 0,67	
PCAP	15% 1,37	35% 1,16	55% 1,00	75% 0,87	90% 0,74	
PCON	48%/год 1,24	24%/год 1,10	12%/год 1,00	6%/год 0,92	3%/год 0,84	
AEXP	≤2 месяцев 1,22	6 месяцев 1,10	1 год 1,00	3 года 0,89	6 лет 0,81	
PEXP	≤2 месяцев 1,25	6 месяцев 1,12	1 год 1,00	3 года 0,88	6 лет 0,81	

Фактор	Очень низкий	Низкий	Номинальный	Высокий	Очень высокий	Сверхвысокий
LTEX	≤2 месяцев 1,22	6 месяцев 1,10	1 год 1.00	3 года 0,91	6 лет 0,84	
TOOL	Редактирование, кодирование, отладка 1.24	Простая входная, выходная CASE утилита, малая интеграция 1,12	Базовые утилиты жизненного цикла, умеренная интеграция 1,00	Развитые утилиты жизненного цикла, умеренная интеграция 0,86	Развитые утилиты жизненного цикла, хорошо интегрированы с процессами, методами, повторным использованием 0,72	
SITE коммуникации	Один телефон, почта 1,25	Индивид. телефоны, FAX 1,10	Узкополосный e-mail 1,00	Широкополос. коммуникации 0,92	Широкополос. коммуникации, иногда видеоконф. 0,84	Интеракт. мультимедиа 0,78
SCED	75% от номин. 1,29	85% 1,10	100% 1,00	130% 1,00	160% 1,00	

Таблица А.19 обеспечивает перевод оценок факторов затрат в числовые значения множителей затрат. Порядок использования таблицы чрезвычайно прост. Имя фактора определяет строку таблицы, оценка фактора — столбец. На пересечении строки и столбца находим числовое значение множителя. Например, фактору TIME с оценкой Высокий соответствует множитель со значением 1,11.

Список литературы

1. *Боэм Б. У.* Инженерное проектирование программного обеспечения. М.: Радио и связь, 1985. 511 с.
2. *Брауде Э. Дж.* Технология разработки программного обеспечения. СПб.: Питер, 2004. 655 с.
3. *Дейкстра Э.* Дисциплина программирования. М.: Мир, 1978. 277 с.
4. *Демарко Т., Листер Т.* Человеческий фактор: успешные проекты и команды. 2-е изд. СПб.: Символ-Плюс; 2005. 256 с.
5. *Константайн Л., Локвуд Л.* Разработка программного обеспечения. СПб.: Питер, 2004. 592 с.
6. *Брукс Ф.* Мифический человеко-месяц, или Как создаются программные системы. СПб.: Символ-Плюс, 1999. 304 с.
7. *Кватрани Т., Палистрат Дж.* Визуальное моделирование с помощью IBM Rational Software Architect и UML. М.: Кудиц-пресс, 2007. 192 с.
8. *Липаев В. В.* Отладка сложных программ: Методы, средства, технология. М.: Энергоатомиздат, 1993. 384 с.
9. *Мартин Р.* Быстрая разработка программ: принципы, примеры, практика. М.: Вильямс, 2004. 752 с.
10. *Майерс Г.* Искусство тестирования программ. М.: Финансы и статистика, 1982. 176 с.
11. *Мюллер Р. Д.* Базы данных и UML: Проектирование. М.: Лори, 2002. 420 с.
12. *Нейбург Э. Д.*, Максимчук Р. А. Проектирование баз данных с помощью UML. М.: Вильямс, 2002. 288 с.
13. *Орлов С. А.* Принципы объектно-ориентированного и параллельного программирования на языке Ада 95. Рига: TSI, 2001. 327 с.
14. Руководство к своду знаний по управлению проектами (руководство PMBOK). 4-е изд. Project Management Institute, 2008. 463 с.
15. *Соммервилл И.* Инженерия программного обеспечения. 6-е изд. М: Вильямс, 2002. 624 с.
16. *Фаулер М.* Рефакторинг: улучшение существующего кода. СПб.: Символ-Плюс, 2003. 432 с.
17. *Чеппел Д.* Технологии ActiveX и OLE. М.: Русская редакция, 1997. 320 с.
18. *Шнейдерман Б.* Психология программирования: Человеческие факторы в вычислительных и информационных системах. М.: Радио и связь, 1984. 304 с.
19. Abreu, F. B., Esteves, R., Goulao, M. The Design of Eiffel Programs: Quantitative Evaluation Using the MOOD metrics. Proceedings of the TOOLS'96, Santa Barbara, California, 20 pp., July 1996.

20. Albrecht, A. J. Measuring Application Development Productivity. Proc. IBM Application Development Symposium, Oct. 1979, pp. 83–92.
21. Ambler, S. W. Agile Database Techniques: Effective Strategies for the Agile Software Developer. John Wiley & Sons, 2003. 416 pp.
22. Ambler, S. W. The Object Primer. 3rd Edition. Cambrige University Press, 2004. 572 pp.
23. Anda, Bente, et al. Effort Estimation of Use Cases for Incremental Large-Scale Software Development. 27th International Conference on Software Engineering, St Louis, MO, 15-21 May 2005, pp. 303-311.
24. Beck, K., and Cunningham, W. A Laboratory for Teaching Object-oriented Thinking. SIGPLAN Notices vol. 24 (10), October 1989, pp. 1–7.
25. Beck, K. Embracing Change with Extreme Programming. IEEE Computer, Vol. 32, No. 10, October 1999, pp. 70–77.
26. Beck, K. Extreme Programming Explained. Embrace Change. Addison-Wesley, 1999. 211 pp. (Русский перевод: Бек К. Экстремальное программирование. СПб.: Питер, 2002. 224 с.)
27. Beck, K., Andres, C. Extreme Programming Explained. Embrace Change. 2nd ed. Addison-Wesley, 2004. 224 pp.
28. Beck, K, Fowler, M. Planning Extreme Programming. Addison-Wesley, 2001. 156 pp. (Русский перевод: Бек К., Фаулер М. Экстремальное программирование: планирование. СПб.: Питер, 2003. 144 с.)
29. Beizer, B. Software Testing Techniques, 2nd ed. New York: International Thomson Computer Press, 1990. 503 pp.
30. Beizer, B. Black-Box Testing: Techniques for Functional Testing of Software and Systems. New York: John Wiley & Sons, 1995. 320 pp. (Русский перевод: Бейзер Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем. СПб.: Питер, 2004. 318 с.)
31. Bieman, J. M. and Kang, B-K. Cohesion and Reuse in an Object-Oriented System. Proc. ACM Symposium on Software Reusability (SSR'95), pp. 259–262, April 1995.
32. Binder, R. V. Testing object-oriented systems: a status report. American Programmer 7 (4), April 1994, pp. 22–28.
33. Binder, R. V. Design for Testability in Object-Oriented Systems. Communications of the ACM, vol. 37, No 9, September 1994, pp. 87–101.
34. Binder, R. V. Testing Object-Oriented Systems. Models, Patterns, and Tools. Addison-Wesley, 1999. 1298 pp.
35. Boehm, B. W. A spiral model of software development and enhancement. IEEE Computer, 21 (5), 1988, pp. 61–72.
36. Boehm, B. W. Software Risk Management: Principles and Practices. IEEE Software, January 1991, pp. 32–41.
37. Boehm, B. W. et al. Software Cost Estimation with Cocomo II. Prentice Hall, 2001. 502 pp.
38. Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen J., Houston, K. Object-Oriented analysis and design with applications. 3rd Edition. Addison-Wesley, 2007. 691 pp. (Русский перевод: Буч Г., Максимчук Р., Энгл М., Янг Б., Коналлен Дж., Хьюстон К. Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е изд. М.: И. Д. Вильямс, 2008. 720 с.)
39. Booch, G., Rumbaugh, J., Jacobcon, I. The Unified Modeling Language User Guide. 2nd Edition. Addison-Wesley, 2005. 496 pp. (Русский перевод: Буч Г., Рамбо Д., Якобсон И. Язык UML. Руководство пользователя. 2-е изд. М.: ДМК Пресс, 2006. 496 с.)

40. Braude, E., Bernstein, M. Software Engineering. Modern Approaches. 2nd Edition. John Wiley & Sons, 2011. 782 pp.
41. Card, D., Glass, R. Measuring Software Design Quality. Prentice Hall, 1990.
42. Carroll, Edward R. Estimating Software Based on Use Case Points. 2005 Object-Oriented, Programming, Systems, Languages, and Applications (OOPSLA) Conference, San Diego, CA, 2005.
43. Clemmons, R.. Project Estimation with Use Case Points, CrossTalk, February 2006, pp. 18-22.
44. Chidamber, S. R. and Kemerer, C. F. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, vol. 20, pp. 476–493. No. 6, June 1994.
45. Cockburn, A. Agile Software Development. Addison-Wesley, 2001. 220 pp. (Русский перевод: Коберн А. Быстрая разработка программного обеспечения. М.: Лори, 2002. 314 с.)
46. Coplien, J. O. Multi-Paradigm Design for C++. Addison-Wesley, 1999. 297 pp. (Русский перевод: Коплиен Дж. Мультипарадигменное проектирование для C++. СПб.: Питер, 2005. 235 с.)
47. DeMarco, T. Structured Analysis and System Specification. Englewood Cliffs, NJ: Prentice-Hall, 1979.
48. Fenton, N. E., Pfleeger S. L. Software Metrics: A Rigorous & Practical Approach. 2nd Edition . International Thomson Computer Press, 1997. 647 pp.
49. Fowler, M. The New Methodology. http://www.martinfowler.com, 2001.
50. Fowler, M. Is Design Dead? Proceedings of the XP 2000 conference, the Mediterranean island of Sardinia, 11 pp., June 2000.
51. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley, 1995. 410 pp. (Русский перевод: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер. 2001. 368 с.)
52. Graham, I. Object-Oriented Methods. Principles & Practice. 3rd Edition. Addison-Wesley. 2001. 853 pp.
53. Halstead, M. H. Elements of Software Science. New York, Elsevier North-Holland, 1977. (Русский перевод: Холстед М. Х. Начала науки о программах. М.: Финансы и статистика, 1981. 128 с.)
54. Hatley, D., and Pirbhai, I. Strategies for Real-Time System Specification. New York, NY: Dorset House, 1988.
55. Henderson-Sellers, B., Constantine, L., Graham, I. Coupling and Cohesion: towards a valid metrics suite for object-oriented analysis and design. Object-oriented Systems 3(3), 1996, p. 143–158.
56. Henry, S. and Kafura, D. Software Structure Metrics Based on Information Flow. IEEE Transactions on Software Engineering, vol. 7, No. 5, pp. 510-518, Sept. 1981.
57. Highsmith, J. A. Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. Dorset House Publishing, 2000. 392 pp.
58. Highsmith, J. A. Extreme programming. e-business Application Delivery, vol. XII, No. 2; February 2000, pp 1–16.
59. Hitz, M., Montazeri, B. Measuring Coupling and Cohesion in Object-Oriented Systems. Proc. Int'l Symp. Applied Corporate Computing (ISACC '95), Monterrey, Mexico, Oct. 25–27, 1995.
60. Hitz, M., Montazeri, B. Measuring Coupling in Object-Oriented Systems. Object Currents, vol. 2, 17 pp., No 4, Apr 1996.
61. Jackson, M. A. Principles of Program Design. London: Academic Press, 1975.

62. Jacobcon, I., Booch, G., Rumbaugh, J. The Unified Software Development Process. Addison-Wesley, 1999 463 pp. (Русский перевод: Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения. СПб.: Питер, 2002. 496 с.)
63. Jacobcon, I., Christerson, M., Jonsson, P., Overgaard, G. J. Object-Oriented Software Engineering. Addison-Wesley, 1993. 528 pp.
64. Jorgensen, P. C. and Erickson, C. Object Oriented Integration. Communications of the ACM, vol. 37, No 9, September 1994, pp. 30–38.
65. Karner, Gustav. Resource Estimation for Objectory Projects. Objective Systems SF AB, 1993. 9 pp.
66. Kirani, S. and Tsai, W. T. Specification and Verification of Object-Oriented programs, Technical Report TR 94–64 Computer Science Department University of Minnesota, December 1994. 99 pp.
67. Kruchten, Phillipe B. The 4+1 View Model of Architecture. IEEE Software, Vol. 12 (6), November 1995, pp. 42–50.
68. Liskov, B. Data Abstraction and Hierarchy. SIGPLAN Notices, vol. 23, no. 5, May 1988.
69. Lorenz, M. and Kidd, J. Object-Oriented Software Metrics. Prentice Hall, 1994. 146 pp.
70. Mapping Object to Data Models with the UML. – Rational Software White Paper, TP-185, 2000. 13 pp.
71. Marick, B. Notes on Object-Oriented Testing. Part 1: Fault-Based Test Design. Part 2: Scenario-Based Test Design. Testing Foundations Inc., 1995. 11 pp.
72. Martin, Robert C. RUP/XP Guidelines: Test-first Design and Refactoring. Rational Software White Paper, 2000, 17 pp.
73. McCabe, T. J. A Complexity Measure. IEEE Transactions on Software Engineering, vol. 2: pp. 308-320. No.4, Apr 1976.
74. McGregor, J. D. and Korson, T. D. Integrated Object Oriented testing and Development Processes. Communications of the ACM, vol. 37, No 9, September 1994, pp. 59–77.
75. McGregor, J. D., Sykes, D. A. A Practical Guide to Testing Object-Oriented Software. Addison-Wesley, 2001. 407 pp..(Русский перевод: МакГрегор Дж., Сайкс Д. Тестирование объектно-ориентированного программного обеспечения. Киев: DiaSoft, 2002. 432 с.)
76. Myers, G. Composite Structured Design. New York, NY: Van Nostrand Reinhold, 1978.
77. OMG Unified Modeling Language Superstructure. Version 2.3. Object Management Group, Inc., 2010. 758 pp.
78. Orr, K. T. Structured Systems Analysis. Englewood Cliffs, NJ: Yourdon Press, 1977.
79. Ott, L., Bieman, J. M., Kang, B-K., Mehra, B. Developing Measures of Class Cohesion for Object-Oriented Software. Proc. Annual Oregon Workshop on Software Merics (AOWSM'95). 11 pp., June 1995.
80. Oviedo, E. I. Control Flow, Data Flow and Program Complexity. Proc. IEEE COMPSAC, Nov. 1980, pp. 146–152.
81. Quatrani, T. Visual Modeling with Rational Rose and UML. Addison-Wesley, 1998. 222 pp.. (Русский перевод 2-го изд.: Кватрани Т. Rational Rose 2000 и UML. Визуальное моделирование. М.: ДМК Пресс, 2001. 176 с.)
82. Page-Jones, M. The Practical Guide to Structured Systems Design. Englewood Cliffs, NY: Yourdon Press, 1988.
83. Page-Jones, M. Fundamentals of Object-Oriented Design in UML. Addison–Wesley, 2001. 479 pp.
84. Parnas, D. On the Criteria to the Be Used in Decomposing Systems into Modules. Communications of the ACM vol. 15 (12), December, 1972, pp. 1053–1058.

85. Paulk, M. C., B. Curtis, M. B. Chrissis, and C. V. Weber. Capability Maturity Model, Version 1.1. IEEE Software, 10, 4, July 1993, pp. 18–27.
86. Paulk, M. C. Extreme Programming from a CMM Perspective. XP Universe, Raleigh, NC, 23–25 July 2001, 8 pp.
87. Poston, R. M. Automated Testing from Object Models. Communications of the ACM, vol. 37, No 9, September 1994, pp. 48–58.
88. Pressman, R. S. Software Engineering: A Practioner's Approach. 7th ed. McGraw-Hill, 2010. 895 pp.
89. Royce, Walker W. Managing the development of large software systems: concepts and techniques. Proc. IEEE WESTCON, Los Angeles, August 1970, pp. 1–9.
90. Rumbaugh J., Blaha M. Object Oriented Modeling and Design with UML. 2nd ed. Prentice Hall, 2004. 496 pp. (Русский перевод: Рамбо Дж., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка. 2-е изд. СПб.: Питер, 2007. 544 с.)
91. Rumbaugh, J., Jacobcon, I., Booch, G., The Unified Modeling Language Reference Manual. 2nd ed. Addison-Wesley. 2005. 721 pp. (Русский перевод: Рамбо Дж., Якобсон А., Буч Г. UML. 2-е изд. СПб.: Питер, 2006. 736 с.)
92. Shalloway, A., Trott, J. R. Design Patterns Explained. A New Perspective on Object-Oriented Design. Addison–Wesley, 2002. 361 pp. (Русский перевод: Шаллоуей А., Тротт Дж. Р. Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию. М.: Изд. д. Вильямс, 2002. 288 с.)
93. Software Engineering. Report on a conference sponsored by the NATO Science Committee. Garmisch, Germany, 7th to 11th October 1968. 136 pp.
94. Sommerville, I. Software Engineering. 9th ed. Addison-Wesley, 2011. 773 pp.
95. Stevens, W.. Myers, G., and Constantine, L. 1979. Structured Design. IBM Systems Journal, Vol. 13 (2), 1974, pp. 115–139.
96. Vliet, J. C. van. Software Engineering: Principles and Practice. John Wiley & Sons, 1993. 558 pp.
97. Tai, K., and Su, H. Test Generation for Boolean Expressions. Proc. COMPSAC'87, October 1987, pp. 278–283.
98. The UML and Data Modeling. Rational Software White Paper, TP-180, 2000. 11 pp.
99. Ward, P., and Mellor, S. Structured Development for Real-Time Systems: Introduction and Tools. Vols. 1, 2, and 3. Englewood Cliffs, NJ: Yourdon Press, 1985.
100. Warnier, J. D. Logical Construction of Programs. New York: Van Nostrand Reinhold, 1974.
101. Wells, J. D. Extreme Programming: A gentle introduction. http:// www.extremeprogramming.org, 2001.
102. Wirfs-Brock, R., Wilkerson, B., and Wiener, L. Designing Object-oriented Software. Englewood Cliffs, New Jersey: Prentice Hall, 1990. 341 pp.
103. Wirth, N. Program Development by Stepwise Refinement, CACM, vol. 14, no. 4, 1971. pp. 221–227.
104. Yourdon, E., and Constantine, L. Structured Design: fundamentals of a discipline of computer program and systems design. Englewood Cliffs, NJ: Prentice-Hall, 1979.

Алфавитный указатель

К

Л

М

Н

О

П

P

C

Т

САЛД

САНКТ-ПЕТЕРБУРГСКАЯ
АНТИВИРУСНАЯ
ЛАБОРАТОРИЯ
ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

Антивирусные
программные продукты

Базовый курс для студентов, обучающихся в высших учебных заведениях по специальностям направления «Информатика и вычислительная техника», преподавателей и профессионалов, чья деятельность связана с программной инженерией.

Учебник посвящен систематическому изложению принципов, моделей и методов, используемых в инженерном цикле разработки сложных программных продуктов.

Авторы стремились к достижению четырех целей:

- изложить классические основы, отражающие накопленный отечественный и мировой опыт программной инженерии;
- показать последние научные и практические достижения, характеризующие динамику развития в области Software Engineering;
- обеспечить комплексный охват наиболее важных вопросов, возникающих в больших и средних программных проектах;
- обобщить и отразить 20-летний университетский опыт преподавания соответствующих дисциплин.

Допущено Министерством образования и науки Российской Федерации в качестве учебника для студентов высших учебных заведений, обучающихся по специальности «Программное обеспечение вычислительной техники и автоматизированных систем» направлений подготовки дипломированных специалистов «Информатика и вычислительная техника».

ISBN: 978-5-459-01101-2

ПИТЕР

Заказ книг:

197198, Санкт-Петербург, а/я 127
тел.: (812) 703-73-74, postbook@piter.com

61093, Харьков-93, а/я 9130
тел.: (057) 758-41-45, 751-10-02, piter@kharkov.piter.com

www.piter.com — вся информация о книгах и веб-магазин