

ООЧ
о-66

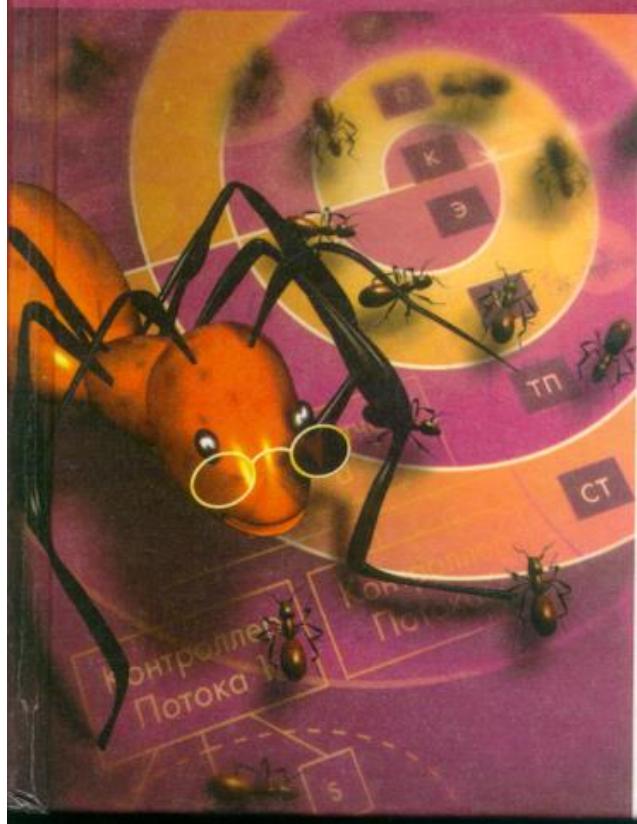
Орлов

ПИТЕР®

ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

РАЗРАБОТКА СЛОЖНЫХ ПРОГРАММНЫХ СИСТЕМ

УЧЕБНИК / ДЛЯ ВУЗОВ



- для студентов и преподавателей высших учебных заведений направления «Информатика и вычислительная техника»
- фундаментальный курс по программной инженерии



УЧЕБНИК / ДЛЯ ВУЗОВ

С. А. Орлов

ТЕХНОЛОГИИ РАЗРАБОТКИ
ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

РАЗРАБОТКА СЛОЖНЫХ ПРОГРАММНЫХ СИСТЕМ

Допущено Министерством образования Российской Федерации
в качестве учебного пособия для студентов высших учебных заведений,
обучающихся по направлению подготовки бакалавров и магистров
«Информатика и вычислительная техника»



Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Ростов-на-Дону • Екатеринбург • Самара
Киев • Харьков • Минск
2002

ББК 32.973-018я7
УДК 681.3.06(075)
О-66

Рецензенты:

Филиппович Ю. Н., канд. техн. наук, доцент Московского государственного Университета печати
Ревунков Г. И., канд. техн. наук, доцент Московского государственного технического Университета
им. Н Э. Баумана

О-66 Технологии разработки программного обеспечения: Учебник/ С. Орлов. — СПб.: Питер, 2002.
— 464 с.: ил.

ISBN 5-94723-145-X

Учебник посвящен систематическому изложению принципов, моделей и методов, 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 больших программных проектах. В основу материала положен двенадцатилетний опыт преподавания автором соответствующих дисциплин.

Книга допущена Министерством образования РФ в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению подготовки бакалавров и магистров «Информатика и вычислительная техника».

ББК 32.973-018я7
УДК 681.3.06(075)

ISBN 5-94723-145-X © ЗАО Издательский дом «Питер», 2002

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 промышленного конструирования программных систем (ПС).

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

В основу материала положен двенадцатилетний опыт постановки и преподавания автором соответствующих дисциплин в Рижском авиационном университете и Рижском институте транспорта и связи. Базовый курс «Технология конструирования программного обеспечения» прослушали больше тысячи студентов, работающих теперь в инфраструктуре мировой программной индустрии, в самых разных странах и на самых разных континентах.

Автор стремился к достижению трех целей:

- изложить классические основы, отражающие накопленный 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 программных проектах.

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. Одними из результатов такой работы являются международный стандарт по компьютерному образованию Computing Curricula 2001 — Computer Science и международный стандарт по программной инженерии IEEE/ACM Software Engineering Body of Knowledge SWEBOK 2001.

Содержание данного учебника отвечает рекомендациям этих стандартов. Учебник состоит из 17 глав.

Первая глава посвящена организации классических, современных и перспективных процессов разработки ПО.

Вторая глава знакомит с вопросами руководства программными проектами — планированием, оценкой затрат. Вводятся размерно-ориентированные и функционально-ориентированные метрики затрат, обсуждается методика их применения, описывается наиболее популярная модель для оценивания затрат — СОСМО II. Приводятся примеры предварительной оценки программного проекта и анализа чувствительности проекта к изменению условий разработки.

Третья глава рассматривает классические методы анализа при разработке ПО.

Четвертая глава отведена основам проектирования программных систем. Здесь обсуждаются архитектурные модели ПО, основные проектные характеристики: модульность, информационная закрытость, сложность, связность, сцепление и метрики для их оценки.

Пятая глава описывает классические методы проектирования ПО.

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

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

Восьмая 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).

Девятая глава посвящена принципам объектно-ориентированного представления программных систем — особенностям их абстрагирования, инкапсуляции, модульности, построения иерархии. Обсуждаются характеристики основных строительных элементов объектно-ориентированного ПО — объектов и классов, а также отношения между ними.

В десятой главе дается сжатое изложение базовых понятий языка визуального моделирования — UML, рассматривается его современная версия 1.4.

Однадцатая глава представляет инструментарий UML для задания статических моделей, описывающих структуру объектно-ориентированных программных систем.

Двенадцатая глава отображает самый многочисленный инструментарий UML — инструментарий для задания динамических моделей, описывающих поведение объектно-ориентированных программных систем. Впрочем, здесь излагаются и смежные вопросы: формирование модели требований к разработке ПО с помощью аппарата Use Case, фиксация комплексных динамических решений с помощью коопераций и паттернов, бизнес-моделирование как средство предпроектного анализа организации-заказчика.

Тринадцатая глава отведена моделям реализации, описывающим формы представления объектно-ориентированных программных систем в физическом мире. Помимо компонентов, узлов и соответствующих диаграмм их обитания здесь приводится пример современной компонентной модели от Microsoft — СОМ.

В четырнадцатой главе обсуждается метрический аппарат для оценки качества объектно-ориентированных проектных решений: метрики оценки объектно-ориентированной связности, сцепления; широко известные наборы метрик Чидамбера и Кемерера, Фернандо Абреу, Лоренца и Кинда; описывается методика их применения.

Пятнадцатая глава решает задачу презентации унифицированного процесса разработки объектно-ориентированных программных систем, на конкретном примере обучает методике применения этого процесса. Кроме того, здесь рассматриваются методика управления риском разработки, процесс разработки в стиле «экстремальное программирование».

Шестнадцатая глава обучает особенностям объектно-ориентированного тестирования, проведению такого тестирования на уровне визуальных моделей, уровне методов, уровне классов и уровне взаимодействия классов.

Семнадцатая глава демонстрирует возможности применения CASE-системы Rational Rose к решению задач автоматизации формирования требований, анализа, проектирования, компонентной упаковки и программирования программного продукта.

Учебник предназначен для студентов 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, мои слова искренней любви родителям — Нине Николаевне и Александру Ивановичу Орловым (светлая им память).

Самые теплые слова благодарности моей семье, родным, любимым и близким мне людям — Лизе, Иванне, Жене. Без их долготерпения, внимания, поддержки, доброжелательности и сердечной заботы эта книга никогда не была бы написана. Моя признательность также и верному сеттеру Эльфу — это он внимательно следил за всеми моими ночных бдениями и клал мне лапу на плечо в особо трудные минуты.

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

Хочется отметить, что корабль-учебник не прибыл бы в порт назначения без опытного капитана (руководителя проекта) Андрея Васильева. Автор искренне признателен талантливым сотрудникам издательства «Питер».

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

От издательства

Ваши замечания, предложения, вопросы вы можете отправить по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция). Мы будем рады узнать ваше мнение.

Все исходные тексты, приведенные в книге, вы можете найти по адресу <http://www.piter.com/download>.

На web-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1. ОРГАНИЗАЦИЯ ПРОЦЕССА КОНСТРУИРОВАНИЯ

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

Определение технологии конструирования программного обеспечения

Технология конструирования программного обеспечения (ТКПО) — система инженерных принципов для создания экономичного ПО, которое надежно и эффективно работает в реальных компьютерах [64], [69], [71].

Различают методы, средства и процедуры ТКПО.

Методы обеспечивают решение следующих задач:

- планирование и оценка проекта;
- анализ системных и программных требований;
- проектирование алгоритмов, структур данных и программных структур;
- кодирование;
- тестирование;
- сопровождение.

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

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

Процесс конструирования программного обеспечения состоит из последовательности шагов, использующих методы, утилиты и процедуры. Эти последовательности шагов часто называют парадигмами ТКПО.

Применение парадигм ТКПО гарантирует систематический, упорядоченный подход к промышленной разработке, использованию и сопровождению ПО. Фактически, парадигмы вносят в процесс создания ПО организующее инженерное начало, необходимость которого трудно переоценить.

Рассмотрим наиболее популярные парадигмы ТКПО.

Классический жизненный цикл

Старейшей парадигмой процесса разработки ПО является классический жизненный цикл (автор Уинстон Ройс, 1970) [65].

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).

Охарактеризуем содержание основных этапов.

Подразумевается, что разработка начинается на системном уровне и проходит через анализ, проектирование, кодирование, тестирование и сопровождение. При этом моделируются действия стандартного инженерного цикла.

Системный анализ задает роль каждого элемента в компьютерной системе, взаимодействие элементов друг с другом. Поскольку ПО является лишь частью большой системы, то анализ начинается с определения требований ко всем системным элементам и назначения подмножества этих требований программному «элементу». Необходимость системного подхода явно проявляется, когда формируется интерфейс ПО с другими элементами (аппаратуой, людьми, базами данных). На этом же этапе начинается решение задачи планирования проекта ПО. В ходе планирования проекта определяются объем проектных работ и их риск, необходимые трудозатраты, формируются рабочие задачи и планграфик работ.

Анализ требований относится к программному элементу — программному обеспечению. Уточняются и детализируются его функции, характеристики и интерфейс.

Все определения документируются в *спецификации анализа*. Здесь же завершается решение задачи планирования проекта.

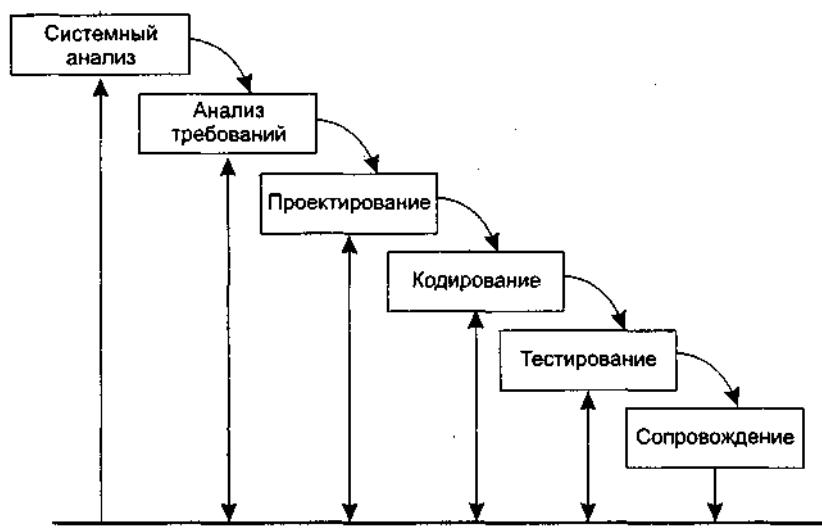


Рис. 1.1. Классический жизненный цикл разработки ПО

Проектирование состоит в создании представлений:

- архитектуры ПО;
- модульной структуры ПО;
- алгоритмической структуры ПО;
- структуры данных;
- входного и выходного интерфейса (входных и выходных форм данных).

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

Кодирование состоит в переводе результатов проектирования в текст на языке программирования.

Тестирование — выполнение программы для выявления дефектов в функциях, логике и форме реализации программного продукта.

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

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

Сопровождение ПО состоит в повторном применении каждого из предшествующих шагов (этапов) жизненного цикла к существующей программе но не в разработке новой программы.

Как и любая инженерная схема, классический жизненный цикл имеет достоинства и недостатки.

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

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 приспособляемости 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.

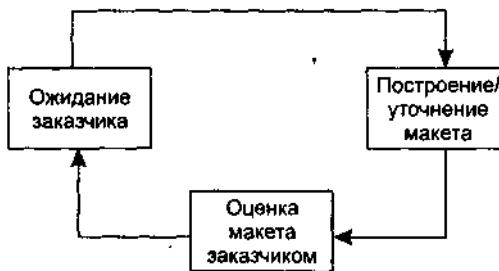


Рис. 1.2. Макетирование

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 сосредоточивается 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.

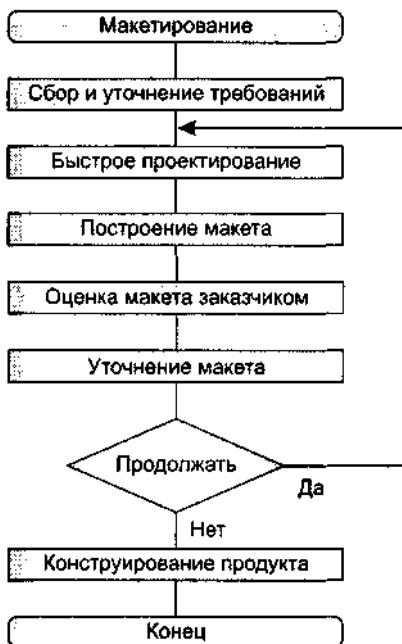


Рис. 1.3. Последовательность действий при макетировании

С другой стороны, для быстрого получения работающего макета разработчик часто идет на определенные компромиссы. Могут использоваться не самые подходящие языки 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 конструирования ПО

Существуют 3 стратегии конструирования ПО:

- однократный проход** (водопадная стратегия) — линейная последовательность этапов конструирования;
- инкрементная стратегия**. В начале процесса определяются все пользовательские и системные требования, оставшаяся часть конструирования 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 конструирования ПО в соответствии с требованиями стандарта IEEE/EIA 12207.2 приведены в табл. 1.1.

Таблица 1.1. Характеристики стратегий конструирования

Стратегия конструирования	В начале процесса определены все требования?	Множество циклов конструирования?	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 конструирования (рис. 1.4). Она объединяет элементы последовательной водопадной модели с итерационной философией макетирования.

Каждая линейная последовательность здесь вырабатывает поставляемый инкремент 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, обеспечивающую дополнительные характеристики и функциональность.

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

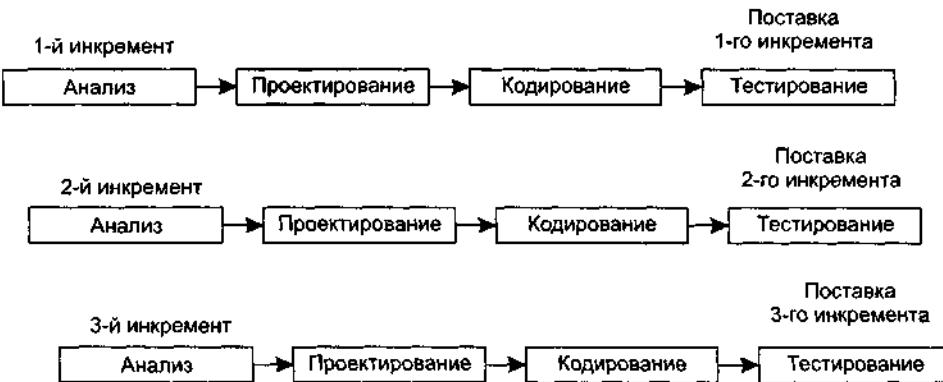


Рис. 1.4. Инкрементная модель

Забегая вперед, отметим, что современная реализация инкрементного подхода — экстремальное программирование XP (Кент Бек, 1999) [10]. Оно ориентировано на очень малые приращения функциональности.

Быстрая разработка приложений

Модель быстрой разработки приложений (Rapid Application Development) — второй пример применения инкрементной стратегии конструирования (рис. 1.5).

RAD-модель обеспечивает экстремально короткий цикл разработки. RAD — высокоскоростная адаптация линейной последовательной модели, в которой быстрая разработка достигается за счет использования компонентно-ориентированного конструирования. Если требования полностью определены, а проектная область ограничена, RAD-процесс позволяет группе создать полностью функциональную систему за очень короткое время (60-90 дней). RAD-подход ориентирован на разработку информационных систем и выделяет следующие этапы:

- **бизнес-моделирование.** Моделируется информационный поток между бизнес-функциями. Ищется ответ на следующие вопросы: Какая информация руководит бизнес-процессом? Какая генерируется информация? Кто генерирует ее? Где информация применяется? Кто обрабатывает

ее?

- **моделирование данных.** Информационный поток, определенный на этапе бизнес-моделирования, отображается в набор объектов данных, которые требуются для поддержки бизнеса. Идентифицируются характеристики (свойства, атрибуты) каждого объекта, определяются отношения между объектами;
- **моделирование обработки.** Определяются преобразования объектов данных, обеспечивающие реализацию бизнес-функций. Создаются описания обработки для добавления, модификации, удаления или нахождения (исправления) объектов данных;
- **генерация приложения.** Предполагается использование методов, ориентированных на языки программирования 4-го поколения. Вместо создания ПО с помощью языков программирования 3-го поколения, RAD-процесс работает с повторно используемыми программными компонентами или создает повторно используемые компоненты. Для обеспечения конструирования используются утилиты автоматизации;
- **тестирование и объединение.** Поскольку применяются повторно используемые компоненты, многие программные элементы уже протестированы. Это уменьшает время тестирования (хотя все новые элементы должны быть протестированы).

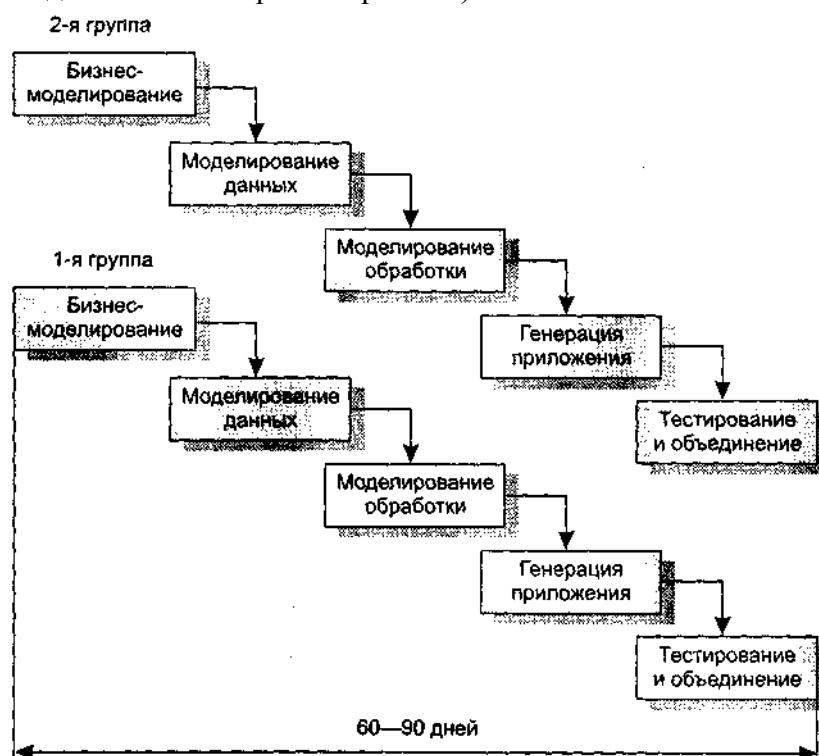


Рис. 1.5. Модель быстрой разработки приложений

Применение RAD возможно в том случае, когда каждая главная функция может быть завершена за 3 месяца. Каждая главная функция адресуется отдельной группе разработчиков, а затем интегрируется в целую систему.

Применение RAD имеет- и свои недостатки, и ограничения.

1. Для больших проектов в RAD требуются существенные людские ресурсы (необходимо создать достаточное количество групп).
2. RAD применима только для таких приложений, которые могут декомпозироваться на отдельные модули и в которых производительность не является критической величиной.
3. RAD не применима в условиях высоких технических рисков (то есть при использовании новой технологии).

Сpirальная модель

Сpirальная модель — классический пример применения эволюционной стратегии конструирования.

Сpirальная модель (автор Барри Боэм, 1988) базируется на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент — анализ риска,

отсутствующий в этих парадигмах [19].



Рис. 1.6. Спиральная модель: 1 — начальный сбор требований и планирование проекта; 2 — таже работа, но на основе рекомендаций заказчика; 3 — анализ риска на основе начальных требований; 4 — анализ риска на основе реакции заказчика; 5 — переход к комплексной системе; 6 — начальный макет системы; 7 — следующий уровень макета; 8 — сконструированная система; 9 — оценивание заказчиком

Как показано на рис. 1.6, модель определяет четыре действия, представляемые четырьмя квадрантами спирали.

1. Планирование — определение целей, вариантов и ограничений.
2. Анализ риска — анализ вариантов и распознавание/выбор риска.
3. Конструирование — разработка продукта следующего уровня.
4. Оценивание — оценка заказчиком текущих результатов конструирования.

Интегрирующий аспект спиральной модели очевиден при учете радиального измерения спирали. С каждой итерацией по спирали (продвижением от центра к периферии) строятся все более полные версии ПО.

В первом витке спирали определяются начальные цели, варианты и ограничения, распознается и анализируется риск. Если анализ риска показывает неопределенность требований, на помощь разработчику и заказчику приходит макетирование (используемое в квадранте конструирования). Для дальнейшего определения проблемных и уточненных требований может быть использовано моделирование. Заказчик оценивает инженерную (конструкторскую) работу и вносит предложения по модификации (квадрант оценки заказчиком). Следующая фаза 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 модели системы. В каждом цикле по спирали требуется конструирование (нижний правый квадрант), которое может быть реализовано классическим жизненным циклом или макетированием. Заметим, что количество действий по разработке (происходящих в правом нижнем квадранте) возрастает по мере продвижения от центра спирали.

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 шаг системного подхода в итерационную структуру разработки;
- 4) использует моделирование для уменьшения риска и совершенствования программного изделия.

Недостатки спиральной модели:

- 1) новизна (отсутствует достаточная статистика эффективности модели);
- 2) повышенные требования к заказчику;
- 3) трудности контроля и управления временем разработки.

Компонентно-ориентированная модель

Компонентно-ориентированная модель является развитием спиральной модели и тоже основывается

на эволюционной стратегии конструирования. В этой модели конкретизируется содержание квадранта конструирования — оно отражает тот факт, что в современных условиях новая разработка должна основываться на повторном использовании существующих программных компонентов (рис. 1.7).

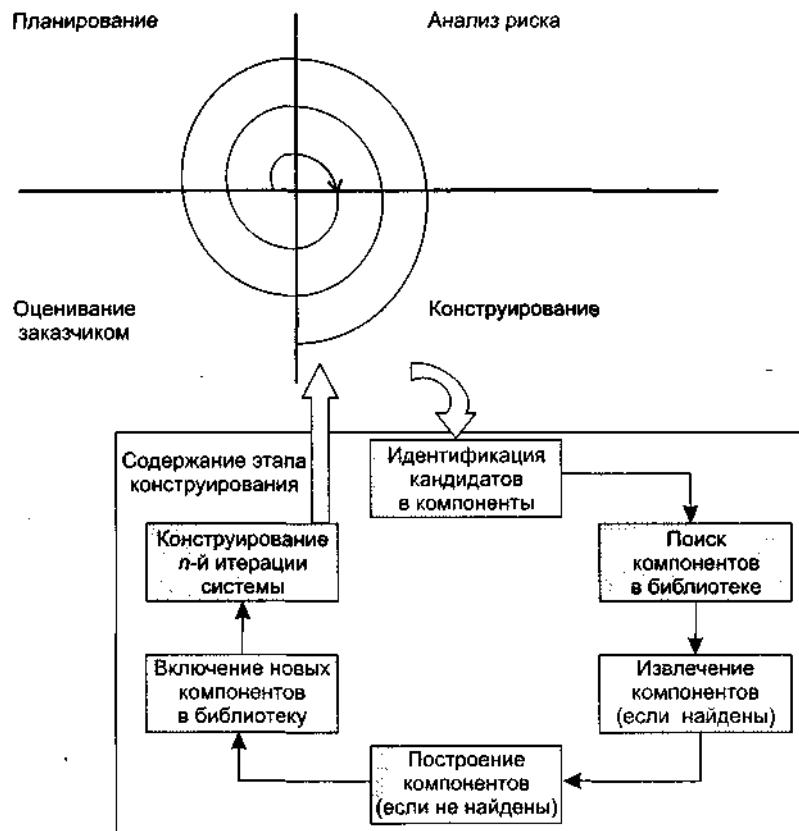


Рис. 1.7. Компонентно-ориентированная модель

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 XXI 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 [29]. Теперь их называют подвижными (agile) процессами [8], [25], [36]. 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 от

разработчиков.

Подвижные 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, подвижные 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, подвижные процессы адаптируют изменения требований и даже выигрывают от этого. Словом, подвижные процессы имеют адаптивную природу.

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

- семейство прогнозирующих (тяжеловесных) процессов;
- семейство адаптивных (подвижных, 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 (подвижный) процесс (или методология), главный автор которого — Кент Бек (1999) [11]. 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 — устраниить высокую стоимость изменения, характерную для приложений с использованием объектов, паттернов* и реляционных баз данных. Поэтому XP-процесс должен быть высокодинамичным 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: кодирование, тестирование, выслушивание заказчика и проектирование. Динамизм обеспечивается с помощью четырех характеристик: непрерывной связи с заказчиком (и в пределах группы), простоты (всегда выбирается минимальное решение), быстрой обратной связи (с помощью модульного и функционального тестирования), смелости в проведении профилактики возможных проблем.

* Паттерн является решением типичной проблемы в определенном контексте.

Большинство принципов, поддерживаемых в XP (минимальность, простота, эволюционный цикл разработки, малая длительность итерации, участие пользователя, оптимальные стандарты кодирования 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».

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	Тестирование модуля, функциональное тестирование
Проектирование	Проектирование является частью ежедневной деятельности каждого разработчика	Реорганизация (refactoring)
Простота	Для системы выбирается простейшее проектное решение, поддерживающее ее текущую	Самая простая вещь, которая могла бы работать

функциональность

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

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 XP образуют перечисленные ниже двенадцать методов.

1. Игра планирования (Planning game) — быстрое определение области действия следующей реализации путем объединения деловых приоритетов и технических оценок. Заказчик формирует область действия, приоритетность и сроки с точки зрения бизнеса, а разработчики оценивают и прослеживают продвижение (прогресс).
2. Частая смена версий (Small releases) — быстрый запуск в производство простой системы. Новые версии реализуются в очень коротком (двухнедельном) цикле.
3. Метафора (Metaphor) — вся разработка проводится на основе простой, общедоступной истории о том, как работает вся система.
4. Простое проектирование (Simple design) — проектирование выполняется настолько просто, насколько это возможно в данный момент.
5. Тестирование (Testing) — непрерывное написание тестов для модулей, которые должны выполняться безупречно; заказчики пишут тесты для демонстрации законченности функций. «Тестируй, а затем кодируй» означает, что входным критерием для написания кода является «отказавший» тестовый вариант.
6. Реорганизация (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 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. 40-часовая неделя (40-hour week) — как правило, работают не более 40 часов в неделю. Нельзя удваивать рабочую неделю за счет сверхурочных работ.
11. Локальный заказчик (On-site customer) — в группе все время должен находиться представитель заказчика, действительно готовый отвечать на вопросы разработчиков.
12. Стандарты кодирования (Coding standards) — должны выдерживаться правила, обеспечивающие одинаковое представление программного кода во всех частях программной системы.

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 о том, какие истории будут осуществлены в пределах двух недель. Полную функциональность, требуемую заказчику, характеризует пул историй; но для следующей двухнедельной итерации из пула выбирается подмножество историй, наиболее важное 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 управляет. Локальный заказчик обеспечивает поддержку этого стиля итерационной разработки.

«Метафора» обеспечивает глобальное «видение» проекта. Она могла бы рассматриваться как высокоуровневая архитектура, но XP подчеркивает желательность проектирования при минимизации проектной документации. Точнее говоря, XP предлагает непрерывное перепроектирование (с помощью реорганизации), при котором нет нужды в детализированной проектной документации, а для инженеров сопровождения единственным надежным источником информации является программный код. Обычно после написания кода проектная документация выбрасывается. Проектная документация 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 реорганизации приводит к реализации простейшего решения, удовлетворяющего текущую потребность. Изменения в требованиях заставляют отказываться от всех «общих решений».

Парное программирование — один из наиболее спорных методов в XP, оно влияет на ресурсы, что важно для менеджеров, решающих, будет ли проект использовать XP. Может показаться, что парное программирование удваивает ресурсы, но исследования 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 повышение 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 тестовые варианты разрабатываются параллельно анализу требований, хотя традиционный подход состоит в тестировании «черного ящика». Размышление о тестировании в начале цикла жизни — хорошо известная практика конструирования ПО (правда, редко осуществляемая практически).

Основным средством управления XP является метрика, а среда метрик — «большая визуальная диаграмма». Обычно используют 3-4 метрики, причем такие, которые видны всей группе. Рекомендуемой в XP метрикой является «скорость проекта» — количество историй заданного размера, которые могут быть реализованы в итерации.

При принятии XP рекомендуется осваивать его методы по одному, каждый раз выбирая метод, ориентированный на самую трудную проблему группы. Конечно, все эти методы 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 методов.

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 — исследование, блокировка, регулирование. Исследование (exploration) — это поиск новых требований (историй, задач), которые должна выполнять система. Блокировка (commitment) — выбор для реализации конкретного подмножества из всех возможных требований (иными словами, планирование). Регулирование (steering) — 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. 1.8.

Процесс инициируется начальной исследовательской фазой.

Фаза исследования, 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.

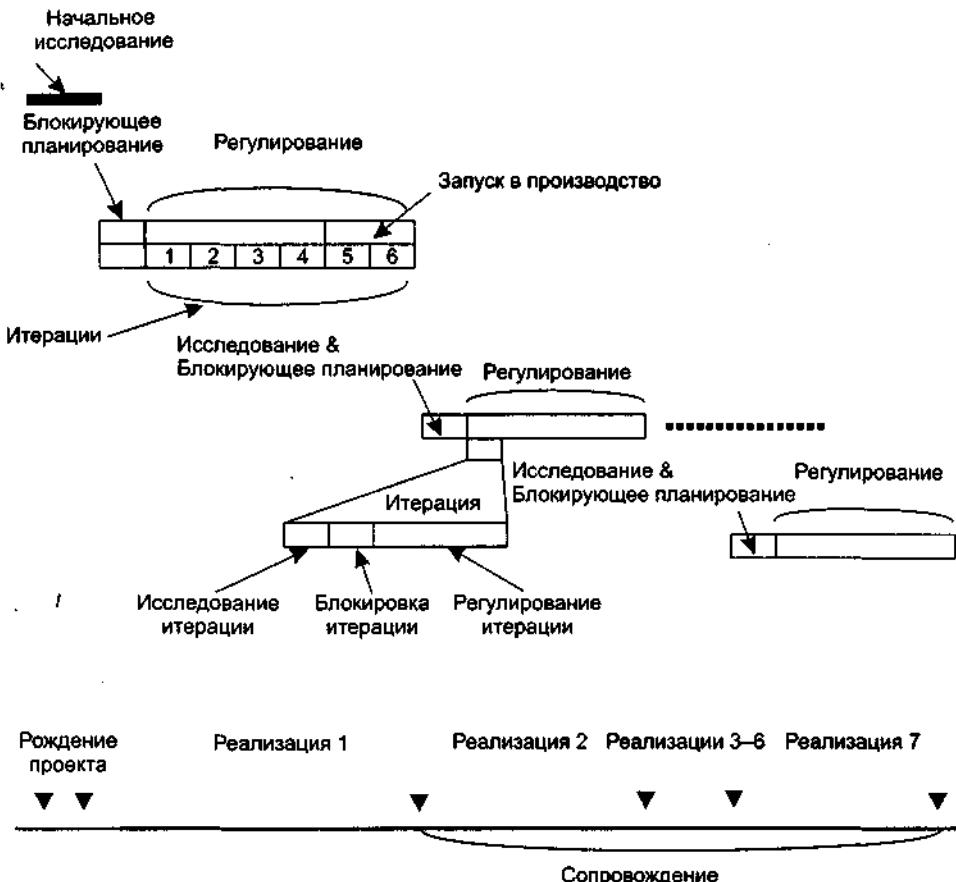


Рис. 1.8. Идеальный XP-процесс

Модели качества процессов конструирования

В современных условиях, условиях жесткой конкуренции, очень важно гарантировать высокое качество вашего процесса конструирования 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 конструирования ПО (Capability Maturity Model — СММ) 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 СММ.

Базовым понятием модели СММ считается зрелость компании [61], [62]. Незрелой называют компанию, где процесс конструирования 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 СММ 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.

Очень важно отметить, что модель СММ ориентирована на построение системы постоянного улучшения процессов. В ней зафиксированы пять уровней зрелости (рис. 1.9) и предусмотрен плавный, поэтапный подход к совершенствованию процессов — можно поэтапно получать подтверждения об улучшении процессов после каждого уровня зрелости.



Рис. 1.9. Пять уровней зрелости модели СММ

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 конструирования. 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 СММ 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 СММ.

Контрольные вопросы

1. Дайте определение технологии конструирования программного обеспечения.
2. Какие этапы классического жизненного цикла вы знаете?
3. Охарактеризуйте содержание этапов классического жизненного цикла.
4. Объясните достоинства и недостатки классического жизненного цикла.
5. Чем отличается классический жизненный цикл от макетирования?
6. Какие существуют формы макетирования?
7. Чем отличаются друг от друга стратегии конструирования ПО?
8. Укажите сходства и различия классического жизненного цикла и инкрементной модели.
9. Объясните достоинства и недостатки инкрементной модели.
10. Чем отличается модель быстрой разработки приложений от инкрементной модели?
11. Объясните достоинства и недостатки модели быстрой разработки приложений.
12. Укажите сходства и различия спиральной модели и классического жизненного цикла.
13. В чем состоит главная особенность спиральной модели?
14. Чем отличается компонентно-ориентированная модель от спиральной модели и классического жизненного цикла?
15. Перечислите достоинства и недостатки компонентно-ориентированной модели.
16. Чем отличаются тяжеловесные процессы от облегченных процессов?
17. Чем отличаются тяжеловесные процессы от прогнозирующих процессов?
18. Чем отличаются подвижные процессы от облегченных процессов?
19. Перечислите достоинства и недостатки тяжеловесных процессов.
20. Перечислите достоинства и недостатки облегченных процессов.
21. Приведите примеры тяжеловесных процессов.
22. Приведите примеры облегченных процессов.
23. Перечислите характеристики XP-процесса.
24. Перечислите методы XP-процесса.
25. В чем состоит главная особенность XP-процесса?
26. Охарактеризуйте содержание игры планирования в XP-процессе.
27. Охарактеризуйте назначение метафоры в XP-процессе.
28. Какова особенность проектирования в XP-процессе?
29. Какова особенность программирования в XP-процессе?
30. Что такое реорганизация?
31. Что такое коллективное владение?
32. Какова особенность тестирования в XP-процессе?
33. Чем отличается XP-реализация от XP-итерации?
34. Чем XP-реализация похожа на XP-итерацию?
35. Какова длительность XP-реализации?
36. Какова длительность XP-итерации?
37. Какова максимальная численность группы XP-разработчиков?
38. Какие модели качества процессов конструирования вы знаете?
39. Охарактеризуйте модель СММ.
40. Охарактеризуйте уровень зрелости знакомой вам фирмы.

ГЛАВА 2. РУКОВОДСТВО ПРОГРАММНЫМ ПРОЕКТОМ

В этой главе детально рассматривается такой элемент процесса конструирования ПО, как руководство программным проектом. Читатель знакомится с вопросами планирования проекта, оценки затрат проекта. В данной главе обсуждаются размерно-ориентированные и функционально-ориентированные метрики затрат, методика их применения. Достаточно подробно описывается наиболее популярная модель для оценивания затрат — СОСМО II. В качестве иллюстраций

приводятся примеры предварительного оценивания проекта, анализа влияния на проект конкретных условий разработки.

Процесс руководства проектом

Руководство программным проектом — первый слой процесса конструирования ПО. Термин «слой» подчеркивает, что руководство определяет сущность процесса разработки от его начала до конца. Принцип руководства иллюстрирует рис. 2.1.

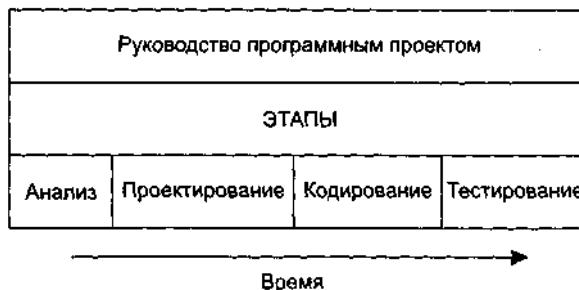


Рис. 2.1. Руководство в процессе конструирования ПО

На этом рисунке прямоугольник обозначает процесс конструирования, в нем выделены этапы, а вверху, над каждым из этапов, размещен слой деятельности «руководство программным проектом».

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 [32], [64], [69].

Начало проекта

Перед планированием проекта следует:

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

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, дающим числовые значения и называемым *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.

Процесс оценки

При планировании программного проекта надо оценить людские ресурсы (в человеко-месяцах), продолжительность (в календарных датах), стоимость (в тысячах долларов). 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.

Анализ риска

На этой стадии исследуется область неопределенности, имеющаяся в наличии перед созданием программного продукта. Анализируется ее влияние на проект. Нет ли скрытых от внимания трудных технических проблем? Не станут ли изменения, проявившиеся в ходе проектирования, причиной недопустимого отставания по срокам? В результате принимается решение — выполнять проект или нет.

Планирование

Определяется набор проектных задач. Устанавливаются связи между задачами, оценивается сложность каждой задачи. Определяются людские и другие ресурсы. Создается сетевой график задач, проводится его времененная 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 конструирования. 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.

Планирование проектных задач

Основной задачей при планировании является определение WBS — Work Breakdown Structure (структурь распределения работ). Она составляется с помощью утилиты планирования проекта. Типовая WBS приведена на рис. 2.2.

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

Системный анализ проводится с целью:

- 1) выяснения потребностей заказчика;
- 2) оценки выполнимости системы;
- 3) выполнения экономического и технического анализа;
- 4) распределения функций по элементам компьютерной системы (аппаратуре, программам, людям, базам данных и т. д.);
- 5) определения стоимости и ограничений планирования;
- 6) создания системной спецификации.

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

Анализ требований дает возможность:

- 1) определить 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.

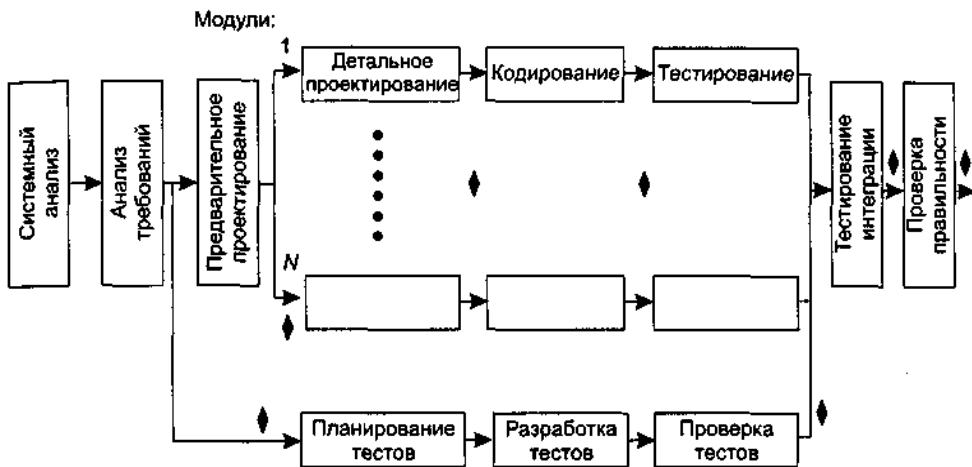


Рис. 2.2. Типовая структура распределения проектных работ

Результаты анализа сводятся в *спецификацию требований* 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. 2.2 обозначены вехи — процедуры контроля промежуточных результатов. Очень важно, чтобы вехи были расставлены через регулярные интервалы (вдоль всего процесса разработки ПО). Это дает руководителю возможность 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, руководитель 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} .

$$T_{\min}^{out} + T_{\min}^{in} + T_{\text{пew}}.$$

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

$$T_{\max}^{out} + T_{\max}^{in} + T_{\text{пew}}.$$

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

Все эти значения позволяют руководителю (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 системный анализ — 5%);
- на кодирование — 20%;
- на тестирование и отладку — 40%.

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. 2.1).

Таблица 2.1. Исходные данные для расчета LOC-метрик

Проект	Затраты, чел.-мес	Стоимость, тыс. \$	KLOC, тыс. LOC	Прогр. док-ты, 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 человека-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):

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

$$\text{Качество} = \frac{\text{Ошибки}}{\text{Длина}} \left[\frac{\text{Единиц}}{\text{тыс. LOC}} \right];$$

$$\text{Удельная стоимость} = \frac{\text{Стоимость}}{\text{Длина}} \left[\frac{\text{Тыс\$}}{\text{LOC}} \right];$$

$$\text{Документированность} = \frac{\text{Страниц Документа}}{\text{Длина}} \left[\frac{\text{Страниц}}{\text{тыс. LOC}} \right].$$

Достоинства размерно-ориентированных метрик:

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

Недостатки размерно-ориентированных метрик:

- 1) зависят 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 характеристики 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 (неповторяющееся) поле, распознаваемое пользователем. В качестве примера рассмотрим табл. 2.2.

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.

Таблица 2.2. 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

Примеры элементов данных для различных характеристик приведены в табл. 2.3, а табл. 2.4

содержит правила учета элементов данных из графического интерфейса пользователя (GUI).

Таблица 2.3. 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 2.4. 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
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. 2.5-2.9 (числовая оценка указана в круглых скобках). Использовать их очень просто. Например, внешнему вводу, который ссылается на 2 файла и имеет 7 элементов данных, по табл. 2.5 назначается средний ранг и оценка сложности 4.

Таблица 2.5. 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 2.6. Ранг и оценка сложности внешних выводов

Ссылки на файлы	Элементы данных		
	1-4	5-19	>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 2.7. Ранг и оценка сложности внешних запросов

Ссылки на файлы	Элементы данных		
	1-4	5-19	>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 2.8. 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 2.9. 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) [7]. Исходные данные для расчета сводятся в табл. 2.10.

Таблица 2.10. Исходные данные для расчета FP-метрик

Имя характеристики	Ранг, сложность, количество			
	Низкий	Средний	Высокий	Итого
Внешние вводы	0x3 = __	0x4 = __	0x6 = __	= 0
Внешние выводы	0x4 = __	0x5 = __	0x7 = __	= 0
Внешние запросы	0x3 = __	0x4 = __	0x6 = __	= 0
Внутренние логические файлы	0x7 = __	0x10 = __	0x15 = __	= 0
Внешние интерфейсные файлы	0x5 = __	0x7 = __	0x10 = __	= 0
Общее количество				= 0

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

$$FP = \text{Общее количество} \times (0,65 + 0,01 \times \sum_{i=1}^{14} F_i), \quad (2.1)$$

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. 2.11).

Таблица 2.11. Определение системных параметров приложения

№	Системный параметр	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.:

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

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

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

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

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. 2.12.

Таблица 2.12. Исходные данные для расчета указателя свойств

№	Характеристика	Количество	Сложность	Итого
1	Вводы	0	x4	= 0
2	Выходы	0	x5	= 0
3	Запросы	0	x4	= 0
4	Логические файлы	0	x7	= 0
5	Интерфейсные файлы	0	x7	= 0
6	Количество алгоритмов	0	x3	= 0
Общее количество				= 0

После заполнения таблицы по формуле (2.1) 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. 2.13, результаты пересчета зависят от языка программирования, используемого для реализации ПО.

Таблица 2.13. 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
HTML3	15
LISP	64
Prolog	64
Miranda	40
Haskell	38

Выполнение оценки в ходе руководства проектом

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 позволяют:

- 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:

$$f_1, f_2, \dots, f_n.$$

- *Шаг 2.* Для каждой функции f_i , планировщик формирует лучшую LOC_{лучши} (FP_{лучши}), худшую LOC_{худши} (FP_{худши}) и вероятную оценку LOC_{вероятни} (FP_{вероятни}). Используются опытные данные (из метрического базиса) или интуиция. Диапазон значения оценок соответствует степени предусмотренной неопределенности.
- *Шаг 3.* Для каждой функции/ в соответствии с β -распределением вычисляется ожидаемое значение LOC- (или FP-) оценки:

$$\text{LOC}_{\text{ожи}} = (\text{LOC}_{\text{лучши}} + \text{LOC}_{\text{худши}} + 4 \times \text{LOC}_{\text{вероятни}}) / 6.$$

- *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_{ср}, взятая из метрического базиса;
- 2) для i -й функции на основе метрики средней производительности вычисляется настраиваемая величина производительности:

$$\text{ПРОИЗВ}_i = \text{ПРОИЗВ}_{\text{ср}} \times (\text{LOC}_{\text{ср}} / \text{LOC}_{\text{ожи}}),$$

где LOC_{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:

$$\text{ПРОИЗВ}_i = \text{ПРОИЗВ}_{\text{ан}_i} \times (\text{LOC}_{\text{ан}_i} / \text{LOC}_{\text{ожи}}).$$

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

$$\text{ЗАТРАТЫ} = \left(\sum_{i=1}^n \text{LOC}_{\text{ожи}} \right) / \text{ПРОИЗВ}_{\text{ср}} [\text{чел. - мес}];$$

для второго и третьего подходов

$$\text{ЗАТРАТЫ} = \sum_{i=1}^n \left(\text{LOC}_{\text{ожи}} / \text{ПРОИЗВ}_i \right) [\text{чел. - мес}].$$

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

$$\text{СТОИМОСТЬ} = \left(\sum_{i=1}^n \text{LOC}_{\text{ожи}} \right) \times \text{УД_СТОИМОСТЬ}_{\text{ср}},$$

где УД_СТОИМОСТЬ_{ср} — метрика средней стоимости одной строки, взятая из метрического базиса.
для третьего подхода

$$\text{СТОИМОСТЬ} = \sum_{i=1}^n \left(\text{LOC}_{\text{ожи}} \times \text{УД_СТОИМОСТЬ}_{\text{ан}_i} \right),$$

где УД_СТОИМОСТЬ_{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 СОСМО 81 (Constructive Cost Model) и ввел в ее состав три разные по сложности статистические подмодели [1].

Иерархию подмоделей Боэма (версии 1981 года) образуют:

- базисная СОСМО — статическая модель, вычисляет затраты разработки и ее стоимость как функцию размера программы;
- промежуточная СОСМО — дополнительно учитывает атрибуты стоимости, включающие основные оценки продукта, аппаратуры, персонала и проектной среды;
- усовершенствованная СОСМО — 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 СОСМО 81 могут применяться к трем типам программных проектов. По терминологии Боэма, их образуют:

- распространенный тип* — небольшие программные проекты, над которыми 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

$$E = a_b x (\text{KLOC})^{\frac{b_b}{d_b}} [\text{чел-мес}];$$

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

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. 2.14.

Таблица 2.14. Коэффициенты для базовой подмодели СОСМО 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 СОСМО II, ориентированную на применение в программной инженерии XXI века [21].

В состав СОСМО II входят:

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

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

Модель композиции приложения

Модель композиции используется на ранней стадии конструирования 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. 2.15, каждый объектный экземпляр (экран, отчет) относят к одному из трех уровней сложности. Здесь места подстановки измеренных и вычисленных значений отмечены прямоугольниками (прямоугольник играет роль метки-заполнителя). В свою очередь, сложность является функцией от параметров клиентских и серверных таблиц данных (см. табл.

2.16 и 2.17), которые требуются для генерации экрана и отчета, а также от количества представлений и секций, входящих в экран или отчет.

Таблица 2.15. Оценка количества объектных указателей

Тип объекта	Количество	Вес			Итого
		Простой	Средний	Сложный	
Экран	0	x1	x2	x3	= 0
Отчет	0	x2	x5	x8	= 0
3GL компонент	0			x10	= 0
Объектные указатели					= 0

Таблица 2.16. Оценка сложности экрана

Экраны	Количество серверных (срв) и клиентских (клт) таблиц данных		
	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 2.17. Оценка сложности отчета

Отчеты	Количество серверных (срв) и клиентских (клт) таблиц данных		
	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. 2.15. 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:

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

Для оценки затрат, основанной на величине NOP, надо знать скорость разработки продукта PROD. Эту скорость определяют по табл. 2.18, учитывающей уровень опыта разработчиков и зрелость среды разработки.

Проектные затраты оцениваются по формуле

$$\text{ЗАТРАТЫ} = \text{NOP} / \text{PROD} [\text{чел.-мес}],$$

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

Таблица 2.18. 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

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:

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

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. 2.19, а табл. 2.20 позволяет определить оценки этих факторов. Оценки принимают 6 значений: от очень низкой (5) до сверхвысокой (0).

Таблица 2.19. 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
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.
Зрелость процесса PMAT	Сверхвысокий, означает интегрированную группу, без проблем взаимодействия Означает зрелость процесса в организации. Вычисление этого фактора может выполняться по вопроснику СММ

В качестве иллюстрации рассмотрим компанию, которая берет проект в малознакомой проблемной 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-му уровню зрелости согласно модели СММ. 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).

Таблица 2.20. Оценка масштабных факторов

Масштабный фактор (W_i)	Очень низкий 5	Низкий 4
-----------------------------	----------------	----------

PREC	Полностью непредсказуемый проект	Главным образом, в значительной степени непредсказуемый
FLEX	Точный, строгий процесс разработки	Редкое расслабление в работе
RESL	Малое разрешение риска (20%)	Некоторое (40%)
TEAM	Очень трудное взаимодействие	Достаточно трудное взаимодействие
PREC	Полностью непредсказуемый проект	В значительной степени непредсказуемый
PMAT	Взвешенное среднее значение от количества ответов «Yes» на вопросник CMM Maturity	

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. 2.21.

Для каждого формирователя затрат определяется оценка (по 6-балльной шкале), где 1 соответствует очень низкому значению, а 6 — сверхвысокому значению. На основе оценки для каждого формирователя по таблице Бюэма 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:

$$\text{ЗАТРАТЫ}_{\text{auto}} = (\text{KALOC} \times (\text{AT} / 100)) / \text{ATPROD},$$

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.

Номинальный 3	Высокий 2	Очень высокий 1	Сверхвысокий 0
Отчасти непредсказуемый	Большой частью знакомый	В значительной степени знакомый	Полностью знакомый
Некоторое расслабление в работе	Большой частью согласованный процесс	Некоторое согласование процесса	Заказчик определил только общие цели
Частое (60%)	Большой частью (75%)	Почти всегда (90%)	Полное (100%)
Среднее взаимодействие	Главным образом кооперативность	Высокая кооперативность	Безукоризненное взаимодействие
Отчасти непредсказуемый	Большой частью знакомый	В значительной степени знакомый	Полностью знакомый
Взвешенное среднее значение от количества ответов «Yes» на вопросник CMM Maturity			

Таблица 2.21. 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)

Модель этапа постархитектуры

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

Основное уравнение постархитектурной модели является развитием уравнения предыдущей модели и имеет следующий вид:

$$\text{ЗАТРАТЫ} = A \times K_{\sim req} \times \text{РАЗМЕР}^B \times M_p + \text{ЗАТРАТЫ}_{\text{auto}} [\text{чел.-мес}],$$

где

- коэффициент $K_{\sim req}$ 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

$$K_{\sim req} = 1 + (\text{BRAK}/100),$$

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

Размер проекта и продукта определяют по выражению

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

где

- $\text{РАЗМЕР}_{\text{new}}$ — размер нового (создаваемого) программного кода;
- $\text{РАЗМЕР}_{\text{reuse}}$ — размер повторно используемого программного кода.

Формула для расчета размера повторно используемого кода записывается следующим образом:

$$\text{РАЗМЕР}_{\text{reuse}} = \text{KASLOC} \times ((100 - AT)/100) \times (AA + SU + 0,4 DM + 0,3 CM + 0,3 IM)/100,$$

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 СОСМО 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-балльной шкале). На основе оценки для каждого фактора по таблице Бюэма определяется множитель затрат EM_i . Перемножение всех множителей затрат дает множитель поправки пост-архитектурной 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:

$$\text{СТОИМОСТЬ} = \text{ЗАТРАТЫ} \times \text{РАБ_КОЭФ},$$

где среднее значение рабочего коэффициента составляет \$15 000 за человека-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:

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

где

□ B — ранее рассчитанный показатель степени;

□ SCEDPercentage — процент увеличения (уменьшения) номинального графика.

Если нужно определить номинальный график, то принимается SCEDPercentage =100 и правый множитель 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 равны 20 человеко-месяцев. Примем, что все масштабные факторы номинальны (имеют значения 3), поэтому, в соответствии с табл. 2.20, показатель степени $B=1,16$. Отсюда следует, что номинальная длительность проекта равна

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

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 устраниет миф нерадивых менеджеров в том, что добавление людей поможет ликвидировать отставание в проекте.

СОСМО 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 проблемную 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 проблемную область. Следует выделить базовые функции ПО и очеркнуть количественные границы. Очевидно, нужно определить, что такое «стандартизированный графический пользовательский интерфейс», какими должны 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 двухмерной графики А2Г.
3. Анализ трехмерной 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 вероятное значения. Ожидаемую LOC-оценку реализации функции определяем по формуле

$$LOC_{ожi} = (LOC_{лучши} + LOC_{худши} + 4 \times LOC_{вероятни}) / 6,$$

результаты расчетов заносим в табл. 2.22.

Таблица 2.22. Начальная таблица оценки проекта

Функция	Лучш. [LOC]	Вероят. [LOC]	Худш. [LOC]	Ожид. [LOC]	Уд. стоимость [\$/LOC]	Стоимость Произв. [\$]	Затраты [LOC/ 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

33360

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. 2.23.

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).

Таблица 2.23. 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

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

$$\text{СТОИМОСТЬ}_i = \text{LOC}_{\text{ож}i} \times \text{УД_СТОИМОСТЬ}_{\text{ан}i}.$$

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

$$\text{ПРОИЗВ}_i = \text{ПРОИЗВ}_{\text{ан}i} \times (\text{LOC}_{\text{ан}i} / \text{LOC}_{\text{ож}i}).$$

Соответственно, затраты на разработку каждой функции будем определять по выражению

$$\text{ЗАТРАТЫ}_i = (\text{LOC}_{\text{ож}i} / \text{ПРОИЗВ}_i) [\text{чел.-мес}].$$

Теперь мы имеем все необходимые данные для завершения расчетов. Заполним до конца таблицу оценки нашего проекта (табл. 2.24).

Таблица 2.24. Конечная таблица оценки проекта

Функция	Лучш.	Вероят.	Худш.	Ожид. [LOC]	Уд. стоимость [\$] [S/LOC]	Стоимость [\$]	Произв. [LOC/ чел.-мес]	Затраты [чел-мес]
СУПИ	1800	2400	2650	2340	14	32760	315	7,4
А2Г	4100	5200	7400	5380	20	107600	245	21,9
А3Г	4600	6900	8600	6800	20	136000	194	35,0
УБД	2950	3400	3600	3350	18	60300	240	13,9
КДГ	4050	4900	6200	4950	22	108900	200	24,7
УП	2000	2100	2450	2140	28	59920	140	15,2
МПА	6600	8500	9800	8400	18	151200	300	28,0
Итого				33360		656680		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. 2.25, 2.26.

Таблица 2.25. Оценка информационных характеристик проекта

Характеристика	Лучш.	Вероят.	Худш.	Ожид.	Сложность	Количество
Вводы	20	24	30	24	x 4	96
Выводы	12	15	22	16	x 5	80
Запросы	16	22	28	22	x 4	88

Логические файлы	4	4	5	4	x 10	40
Интерфейсные файлы	2	2	3	2	x 7	14
Общее количество						318

Таблица 2.26. Оценка системных параметров проекта

Коэффициент 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:

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

Используя значение производительности, взятое в метрическом базисе фирмы,

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

вычисляем значения затрат и стоимости:

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

$$\text{Стоимость} = \text{Затраты} \times \$4500 = \$656500.$$

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

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

и получаем:

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

Соответственно, номинальная длительность проекта равна

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

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

Анализ чувствительности программного проекта

СОСОМО 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 зарплата в команде составит \$ 6000 в месяц. Команда имеет годовой опыт работы с этой проблемной областью и полгода работает с нужной аппаратной платформой.

В терминах СОСМО II проблемную 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. 2.27.

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.

Таблица 2.27. 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 из 64 Кбайт, 70%)	Высокая	1,06
PVOL	Изменчивость платформы (каждые 6 месяцев)	Номинал.	1
ACAP	Возможности аналитика (75%)	Высокая	0,83
PCAP	Возможности программиста (75%)	Высокая	0,87
AEXP	Опыт работы с приложением (1 год)	Номинал.	1
PEXP	Опыт работы с платформой (6 месяцев)	Низкая	1,12
LTEX	Опыт работы с языком и утилитами (1 год)	Номинал.	1
PCON	Непрерывность персонала (1 2% 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_p			1,088

Рассчитаем затраты и стоимость проекта:

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

$$\text{СТОИМОСТЬ} = \text{ЗАТРАТЫ} \times \$6000 = \$234\,000.$$

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:

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

$$\text{СТОИМОСТЬ} = \text{ЗАТРАТЫ} \times \$5000 = \$270\,000,$$

$$\text{Проигрыш в стоимости} = \$36\,000.$$

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:

$$\begin{aligned} EM_{STOR} &= 1 \rightarrow M_p = 1,026, \\ \text{ЗАТРАТЫ} &= 36 \times 1,026 = 37 \text{ [чел.-мес]}, \\ \text{СТОИМОСТЬ} &= \text{ЗАТРАТЫ} \times \$6000 = \$222000, \\ \text{Выигрыш_в_стоимости} &= \$ 12 000. \end{aligned}$$

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 $EM_{TOOL} = 1,24$):

$$\begin{aligned} M_p &= (1,088 / 0,86) \times 1,22 \times 1,24 = 1,914, \\ \text{ЗАТРАТЫ} &= 36 \times 1,914 = 69 \text{ [чел.-мес]}, \\ \text{СТОИМОСТЬ} &= \text{ЗАТРАТЫ} \times \$6000 = \$414000, \\ \text{Проигрыш_в_стоимости} &= \$180000. \end{aligned}$$

Сценарий уменьшения средств на завершение проекта

Положим, что к разработке принят сценарий с наращиванием памяти:

$$\begin{aligned} \text{ЗАТРАТЫ} &= 36 \times 1,026 = 37 \text{ [чел.-мес]}, \\ \text{СТОИМОСТЬ} &= \text{ЗАТРАТЫ} \times \$6000 = \$222000. \end{aligned}$$

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:

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

Очевидно, что

$$\begin{aligned} (\text{НовыйРазмер})^{1,16} &= 12,58, \\ (\text{НовыйРазмер})^{1,16} &= 12,58^{1/1,16} = 8,872 \text{ [KLOC]}. \end{aligned}$$

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 позволяют улучшить принятное 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 СОСМО II предлагает широкий спектр факторов затрат, учитывающих большинство реальных ситуаций в «жизни» программного проекта.
3. Модель СОСМО II обеспечивает перевод качественного обоснования решения менеджера на количественные рельсы, тем самым повышая объективность принимаемого решения.

Контрольные вопросы

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

- модели раннего этапа проектирования?
- 27. Что такое факторы затрат модели этапа пост-архитектуры и как они вычисляются?
 - 28. Как определяется длительность разработки в модели СОСМО II?
 - 29. Что такое анализ чувствительности программного проекта?
 - 30. Как применить модель СОСМО II к анализу чувствительности?

ГЛАВА 3. 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) [27]. 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. 3.1. Диаграмма может использоваться для представления программного изделия на любом уровне абстракции.

Пример системы взаимосвязанных диаграмм показан на рис. 3.2.

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.

Дальнейшее уточнение (например, преобразователя F_3) приводит к диаграмме 2-го уровня. Говорят, что ПДД1 разбивается на диаграммы 2-го уровня.



Рис. 3.1. Элементы диаграммы потоков данных

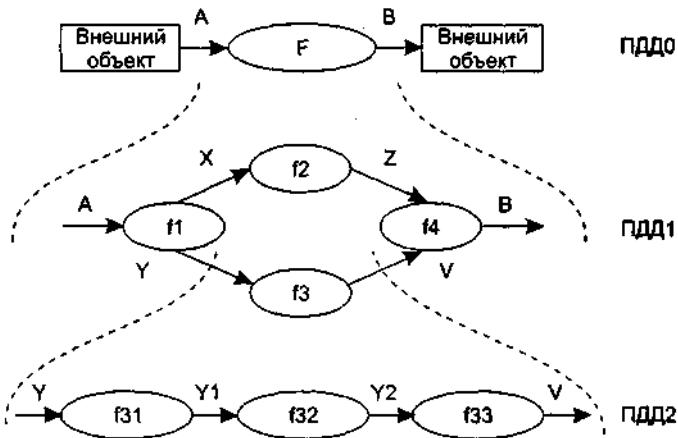


Рис. 3.2. Система взаимосвязанных диаграмм потоков данных

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 описание 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 базовую 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.

Расширения для систем реального времени

Как известно, программное изделие (ПИ) является дискретной моделью проблемной области, взаимодействующей с непрерывными процессами физического мира (рис. 3.3).

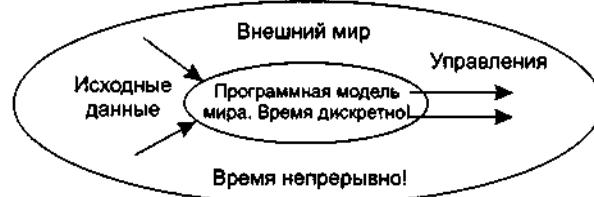


Рис. 3.3. Программное изделие как дискретная модель проблемной области

П. Вард и С. Меллор приспособили диаграммы потоков данных к следующим требованиям систем реального времени [73].

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).

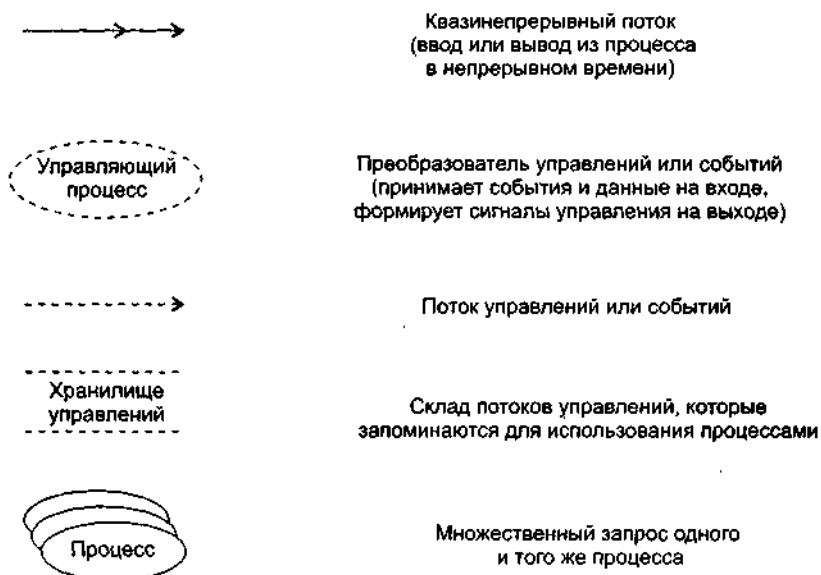


Рис. 3.4. Расширения диаграмм для систем реального времени

Новые элементы имеют обозначения, показанные на рис. 3.4.

Приведем два примера использования новых элементов.

Пример 1. Использование потоков, непрерывных во времени.

На рис. 3.5 представлена модель анализа программного изделия для системы слежения за газовой турбиной.



Рис. 3.5. 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 цифроаналоговую аппаратуру.

Во-вторых, необходимость организации высокоскоростного управления этой аппаратурой делает критичным требование к производительности системы.

Пример 2. Использование потоков управления.

Рассмотрим компьютерную систему, которая управляет роботом (рис. 3.6).



Рис. 3.6. 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, посыпается 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) посыпается в управляющий преобразователь «Начать движение», который руководит дальнейшей командной обработкой. Потоки данных посыпаются в преобразователь команд роботу при наличии события «Процесс активен».

Расширение возможностей управления

Д. Хетли и И. Пирбхай сосредоточили внимание на аспектах управления программным продуктом [34]. Они выделили системные состояния и механизм перехода из одного состояния в другое. Д. Хетли и И. Пирбхай 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. 3.7, ссылка изображается как косая пунктирная стрелка, указывающая на окно УСПЕЦ (вертикальную черту).



Рис. 3.7. 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. 3.8.

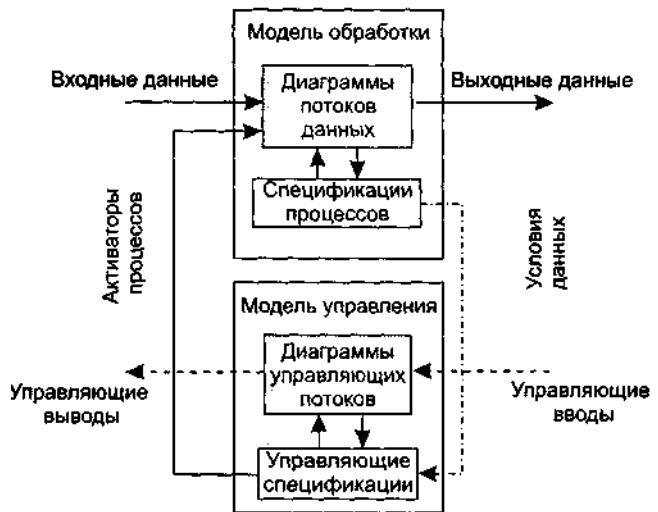


Рис. 3.8. Композиция модели обработки и управления

В модель обработки входит набор диаграмм потоков данных и набор спецификаций процессов. Модель управления образует 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. 3.9.

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.

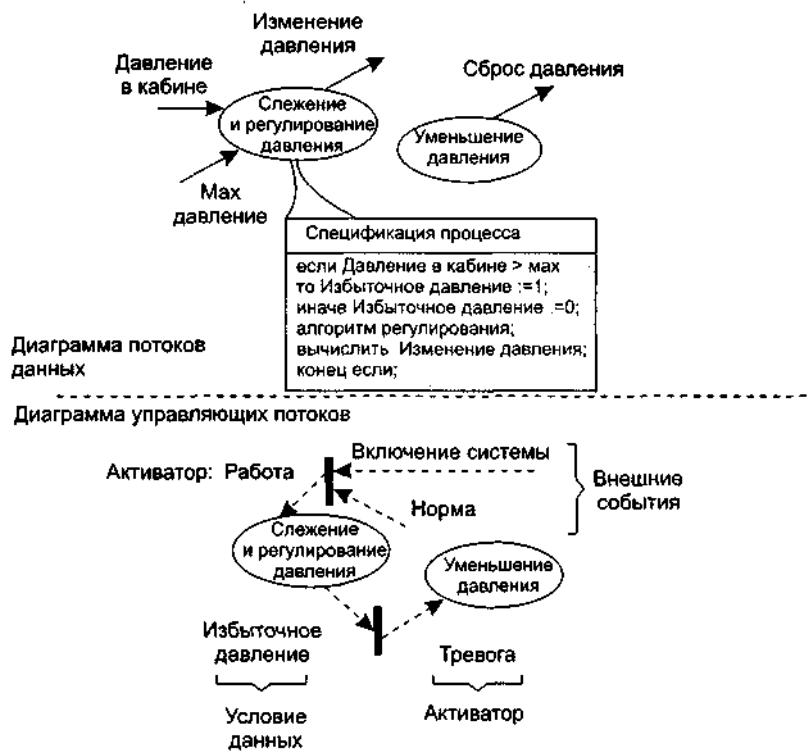


Рис. 3.9. Модель системы регулирования давления космического корабля

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:=11;
    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. 3.1.

Таблица 3.1. 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 показана на рис. 3.10.

Системные состояния показаны прямоугольниками. Стрелки показывают переходы между состояниями. Стрелки переходов подписываются следующим образом: в числителе — событие, которое вызывает переход, в знаменателе — процесс, запускаемый как результат события.

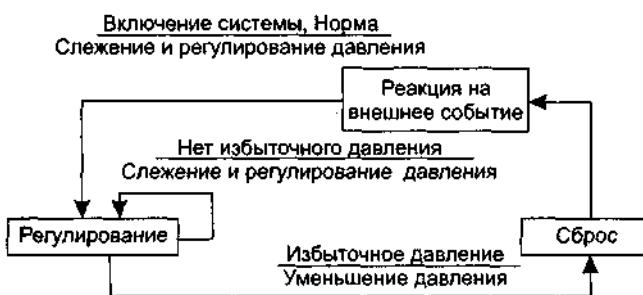


Рис. 3.10. Диаграмма переходов-состояний

Изучая ДПС, разработчик может анализировать поведение модели и установить, нет ли «дыр» в определении поведения.

Методы анализа, ориентированные на структуры данных

Элементами проблемной 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 [54].

Для построения диаграмм Варнье используют 3 базовых элемента: последовательность, выбор, повторение (рис. 3.11) [74].

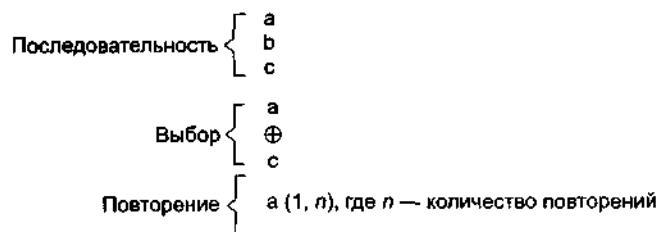


Рис. 3.11. Базовые элементы в диаграммах Варнье

Как показано на рис. 3.12, с помощью этих элементов можно строить информационные структуры с любым количеством уровней иерархии.

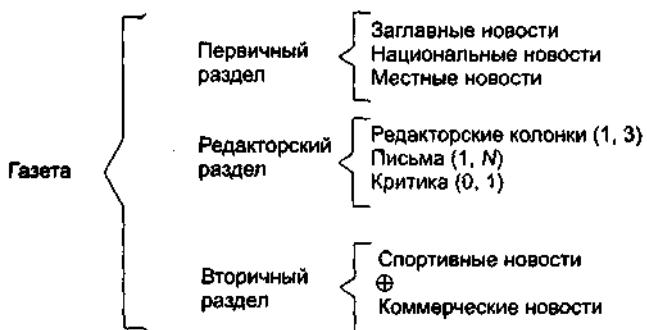


Рис. 3.12. Структура газеты в виде диаграммы Варнье

Как видим, для представления структуры газеты здесь используются три уровня иерархии.

Метод анализа Джексона

Как и метод Варнье-Оппа, метод Джексона появился в период революции структурного программирования. Фактически оба метода решали одинаковую задачу: распространить базовые структуры программирования (последовательность, выбор, повторение) на всю область конструирования сложных программных систем. Именно поэтому основные выразительные средства этих методов оказались так похожи друг на друга.

Методика Джексона

Метод Джексона (1975) включает 6 шагов [39]. 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:

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 исследовать загрузку 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 3 типа структурных диаграмм. Они показаны на рис. 3.13. В первой диаграмме к объектам применяется такое действие, как последовательность, во второй — выбор, в третьей — повторение.

Рассмотрим объектную структуру для транспорта (см. рис. 3.14). Условимся, что начало и конец истории транспорта — у первой остановки. Действиями, влияющими на объект, являются Покинуть и Прибыть.

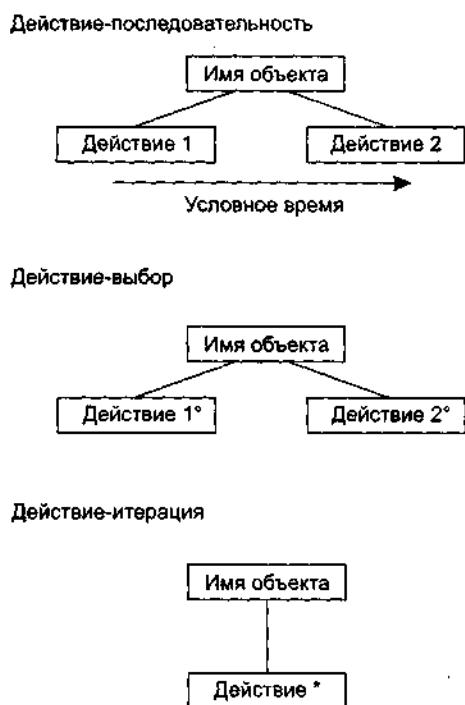


Рис. 3.13. Три типа структурных диаграмм Джексона

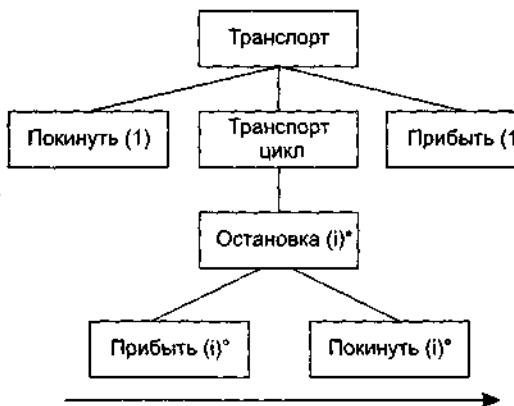


Рис. 3.14. Объектная структура для транспорта

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 представляться средствами метода. Например, «значение г в двух последовательных остановках должно быть разным».

Структурная диаграмма для объекта Кнопка показывает (рис. 3.15), что к нему многократно применяется действие Нажать.



Рис. 3.15. 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. 3.16, предусматриваются 2 вида соединений между физическими процессами и моделями.

1. Соединение потоком данных



2. Соединение по вектору состояний

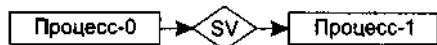


Рис. 3.16. Соединения между физическими процессами и их моделями

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. 3.17.

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.



Рис. 3.17. Диаграмма системной спецификации для системы обслуживания перевозок

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. 3.18.

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;

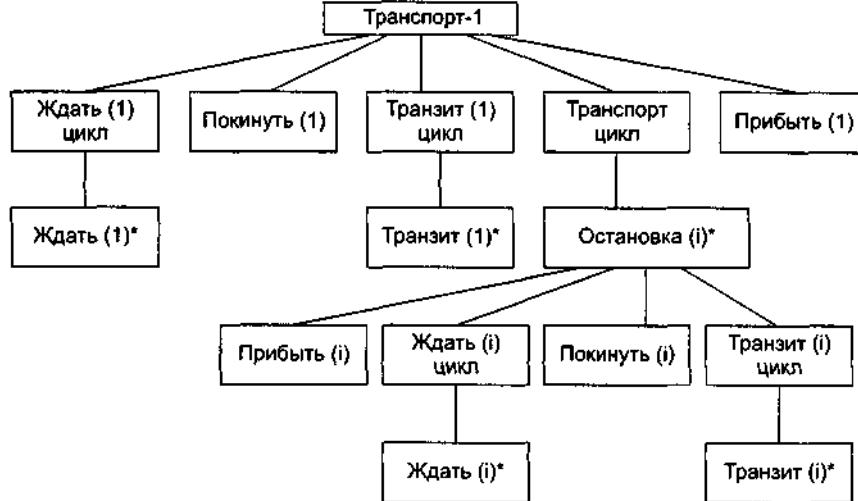


Рис. 3.18. Структурная диаграмма модели транспорта

Контрольные вопросы

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 чем связана необходимость расширения диаграмм потоков данных для систем реального времени? Какие средства расширения вы знаете?
11. Как решается проблема расширения возможностей управления на базе диаграмм потоков данных?
12. Каковы особенности диаграммы управляющих потоков?
13. Поясните понятие активатора процесса.
14. Поясните понятие условия данных.
15. Поясните понятие управляющей спецификации.
16. Поясните понятие окна управляющей спецификации.
17. Как организована спецификация процесса?
18. Поясните назначение таблицы активации процессов.
19. Поясните организацию диаграммы переходов-состояний.
20. Какие задачи решают методы анализа, ориентированные на структуры данных?
21. Какие методы анализа, ориентированные на структуры данных, вы знаете?
22. Из каких базовых элементов состоят диаграммы Варнье?
23. Какие шаги выполняет метод Джексона на этапе анализа?
24. Какие типы структурных диаграмм Джексона вы знаете?
25. Как организовано в методе Джексона обнаружение объектов?
26. Что такое структура объектов Джексона?
27. Как создается структура объектов Джексона?
28. Поясните диаграмму системной спецификации Джексона.
29. Чем отличается соединение потоком данных от соединения по вектору состояний?
30. Какова задача структурного текста Джексона?

ГЛАВА 4. ОСНОВЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ

В этой главе рассматривается содержание этапа проектирования и его место в жизненном цикле конструирования программных систем. Дается обзор архитектурных моделей ПО, обсуждаются классические проектные характеристики: модульность, информационная закрытость, сложность, связность, сцепление и метрики для их оценки.

Особенности процесса синтеза программных систем

Известно, что технологический цикл конструирования программной системы (ПС) включает три процесса — анализ, синтез и сопровождение.

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. 4.1).

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 — разработка данных, разработка архитектуры и процедурная разработка ПС.

Разработка данных — это результат преобразования информационной модели анализа в структуры данных, которые потребуются для реализации программной системы.

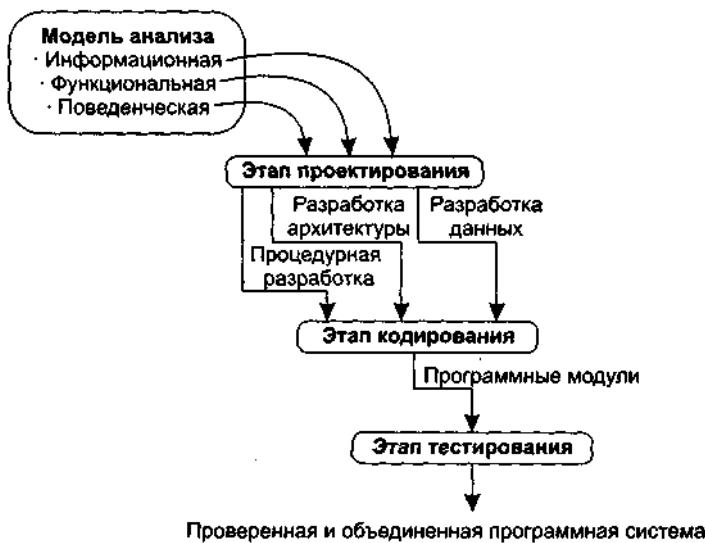


Рис. 4.1. Информационные потоки процесса синтеза ПС

Разработка архитектуры выделяет основные структурные компоненты и фиксирует связи между ними.

Процедурная разработка описывает последовательность действий в структурных компонентах, то есть определяет их содержание.

Далее создаются тексты программных модулей, проводится тестирование для объединения и проверки ПС. На проектирование, кодирование и тестирование приходится более 75% стоимости конструирования 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 «вырашивается» качество разработки ПС. 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 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: предварительное проектирование и детальное проектирование. Предварительное проектирование формирует абстракции архитектурного 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. 4.2.

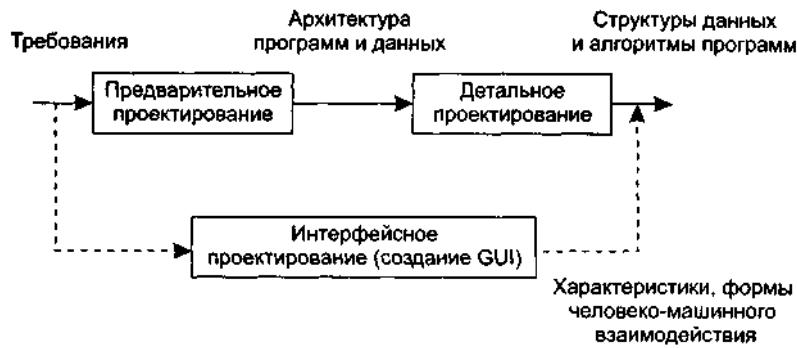


Рис. 4.2. Информационные связи процесса проектирования

Предварительное проектирование обеспечивает:

- идентификацию подсистем;
- определение основных принципов управления подсистемами, взаимодействия подсистем.

Предварительное проектирование включает три типа деятельности:

1. *Структурирование системы*. Система структурируется на несколько подсистем, где под подсистемой понимается независимый программный компонент. Определяются взаимодействия подсистем.
2. *Моделирование управления*. Определяется модель связей управления между 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

Известны четыре модели системного структурирования:

- модель хранилища данных;
- модель клиент-сервер;
- трехуровневая модель;
- модель абстрактной машины.

В модели хранилища данных (рис. 4.3) подсистемы разделяют данные, находящиеся в общей памяти. Как правило, данные образуют БД. Предусматривается система управления этой базой.

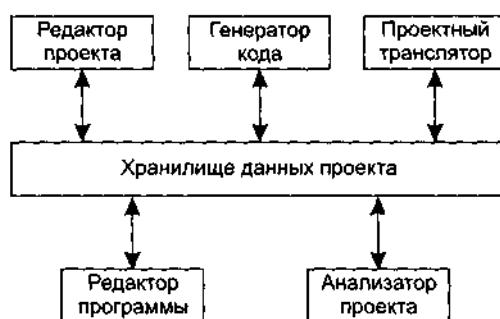


Рис. 4.3. Модель хранилища данных

Модель клиент-сервер используется для распределенных систем, где данные распределены по серверам (рис. 4.4). Для передачи данных применяют сетевой протокол, например TCP/IP.



Рис. 4.4. Модель клиент-сервер

Трехуровневая модель является развитием модели клиент-сервер (рис. 4.5).

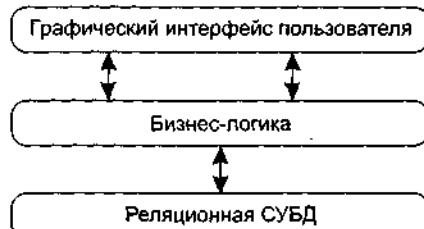


Рис. 4.5. Трехуровневая модель

Уровень графического интерфейса пользователя запускается на машине клиента. Бизнес-логику образуют модули, осуществляющие функциональные обязанности системы. Этот уровень запускается на сервере приложения. Реляционная СУБД хранит данные, требуемые уровню бизнес-логики. Этот уровень запускается на втором сервере — сервере базы данных.

Преимущества трехуровневой модели:

- упрощается такая модификация уровня, которая не влияет на другие уровни;
- отделение прикладных функций от функций управления БД упрощает оптимизацию всей системы.

Модель абстрактной машины отображает многослойную систему (рис. 4.6).

Каждый текущий слой реализуется с использованием средств, обеспечивающих слоем-фундаментом.



Рис. 4.6. Модель абстрактной машины

Моделирование управления

Известны два типа моделей управления:

- модель централизованного управления;
- модель событийного управления.

В модели централизованного управления одна подсистема выделяется как системный контроллер. Ее обязанности — руководить работой других подсистем. Различают две разновидности моделей централизованного управления: *модель вызов-возврат* (рис. 4.7) и *Модель менеджера* (рис. 4.8), которая используется в системах параллельной обработки.

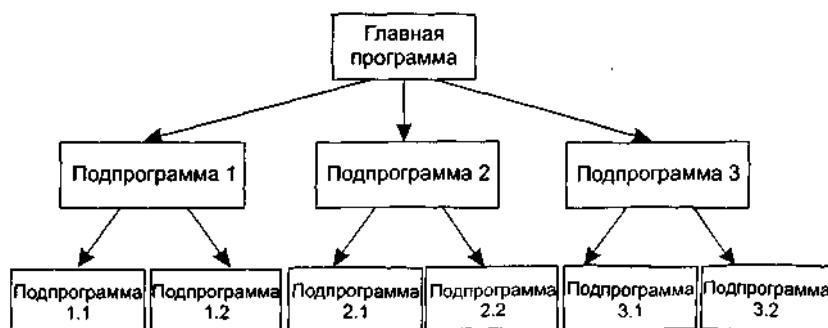


Рис. 4.7. Модель вызов-возврат

В модели событийного управления системой управляют внешние события. Используются две разновидности модели событийного управления: широковещательная модель и модель, управляемая прерываниями.

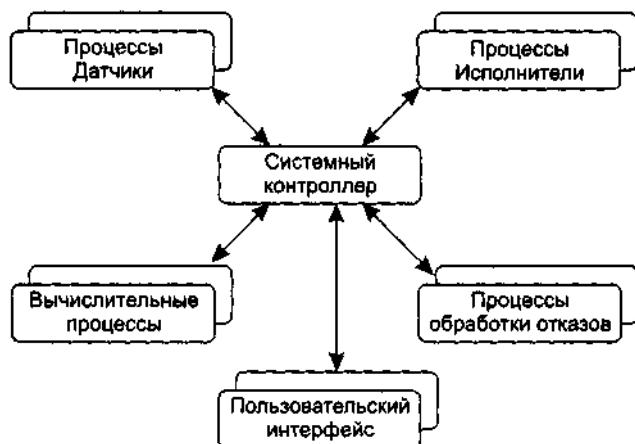


Рис. 4.8. Модель менеджера

В широковещательной модели (рис. 4.9) каждая подсистема уведомляет обработчика о своем интересе к конкретным событиям. Когда событие происходит, обработчик пересыпает его подсистеме, которая может обработать это событие. Функции управления в обработчик не встраиваются.



Рис. 4.9. Широковещательная модель

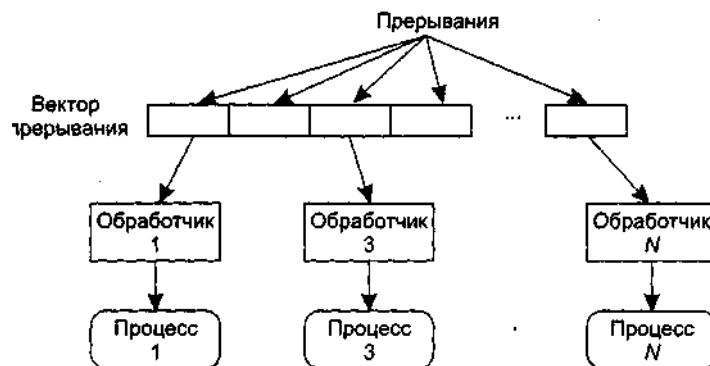


Рис. 4.10. Модель, управляемая прерываниями

В модели, управляемой прерываниями (рис. 4.10), все прерывания разбиты на группы — типы, которые образуют вектор прерываний. Для каждого типа прерывания есть свой обработчик. Каждый обработчик реагирует на свой тип прерывания и запускает свой процесс.

Декомпозиция подсистем на модули

Известны два типа моделей модульной декомпозиции:

- модель потока данных;
- модель объектов.

В основе модели потока данных лежит разбиение по функциям.

Модель объектов основана на слабо сцепленных сущностях, имеющих собственные наборы данных, состояния и наборы операций.

Очевидно, что выбор типа декомпозиции должен определяться сложностью разбиваемой подсистемы.

Модульность

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 [52]. 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

$$T(p_1) > T(p_2). \quad (4.1)$$

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 (4.1) запишем:

$$T(p_1 + p_2) > T(p_1) + T(p_2). \quad (4.2)$$

Соотношение (4.2) — это обоснование модульности. Оно приводит к заключению «разделяй и властвуй» — сложную проблему легче решить, разделив ее на управляемые части. Результат, выраженный неравенством (4.2), имеет важное значение для модульности и ПО. Фактически, это аргумент в пользу модульности.

Однако здесь отражена лишь часть реальности, ведь здесь не учитываются затраты на межмодульный интерфейс. Как показано на рис. 4.11, с увеличением количества модулей (и уменьшением их размера) эти затраты также растут.

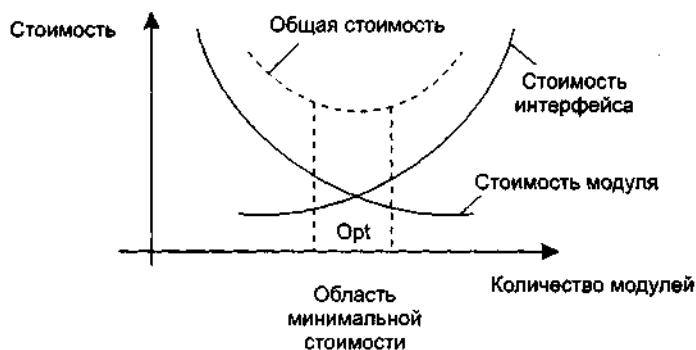


Рис. 4.11. Затраты на модульность

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 [60]. Как показано на рис. 4.12, модуль должен определяться и проектироваться так, чтобы его содержимое (процедуры и данные) было недоступно тем модулям, которые не нуждаются в такой информации (клиентам).

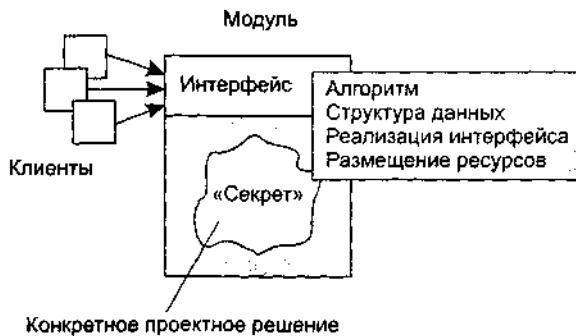


Рис. 4.12. Информационная закрытость модуля

Информационная закрытость означает следующее:

- 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 [58], [70], [77]. 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 (CC). Существует 7 типов связности:

1. **Связность по совпадению** ($CC=0$). В модуле отсутствуют явно выраженные внутренние связи.
2. **Логическая связность** ($CC=1$). 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** ($CC=3$). Части модуля не связаны, но необходимы в один и тот же период работы системы.

Недостаток: сильная взаимная связь с другими модулями, отсюда — сильная чувствительность к внесению изменений.

4. **Процедурная связность** ($CC=5$). Части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения.
5. **Коммуникативная связность** ($CC=7$). Части модуля связаны по данным (работают с одной и той же структурой данных).
6. **Информационная (последовательная) связность** ($CC=9$). Выходные данные одной части используются как входные данные в другой части модуля.
7. **Функциональная связность** ($CC=10$). Части модуля вместе реализуют одну функцию.

Отметим, что типы связности 1,2,3 — результат неправильного планирования архитектуры, а тип связности 4 — результат небрежного планирования архитектуры приложения.

Общая характеристика типов связности представлена в табл. 4.1.

Таблица 4.1. Характеристика связности модуля

Тип связности	Сопровождаемость	Роль модуля
Функциональная	Лучшая сопровождаемость	«Черный ящик»
Информационная (последовательная)		Не совсем «черный ящик»
Кэмуникативная		«Серый ящик»
Процедурная	Худшая сопровождаемость	«Белый» или «просвечивающий 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 связан.

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 3 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 связанного 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 связанного 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 связанного 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 связанном 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 сబлазняется 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 флагги, 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 темного 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 «заявлять» 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 флагков.

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 поданным [58], [70], [77]. 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. 4.13).



Рис. 4.13. Сцепление поданным

2. **Сцепление по образцу** (СЦ=3). В качестве параметров используются структуры данных (рис. 4.14).



Рис. 4.14. Сцепление по образцу

3. **Сцепление по управлению** (СЦ=4). Модуль А явно управляет функционированием модуля В (с помощью флагов или переключателей), посыпая ему управляющие данные (рис. 4.15).

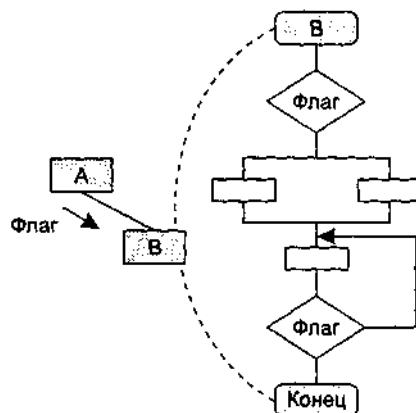


Рис. 4.15. Сцепление по управлению

4. **Сцепление по внешним ссылкам** (СЦ=5). Модули А и В 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. 4.16).

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 (рис. 4.16).

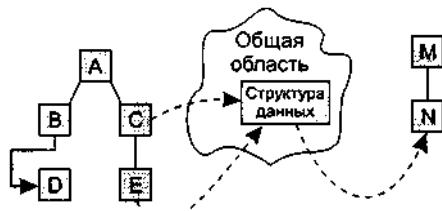


Рис. 4.16. Сцепление по общей области и содержанию

На рис. 4.16 видим, что модули В и D сцеплены по содержанию, а модули С, Е и N сцеплены по общей области.

Сложность программной системы

В простейшем случае сложность системы определяется как сумма мер сложности ее модулей. Сложность модуля может вычисляться различными способами.

Например, М. Холстед (1977) предложил меру длины N модуля [33]:

$$N \approx n_1 \log_2(n_1) + n_2 \log_2(n_2),$$

где n_1 — число различных операторов, n_2 — число различных operandов.

В качестве второй метрики М. Холстед рассматривал объем V модуля (количество символов для записи всех операторов и operandов 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 [49]. 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) [28], [56]. 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

Иерархическая структура программной системы — основной результат предварительного проектирования. 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. 4.17.

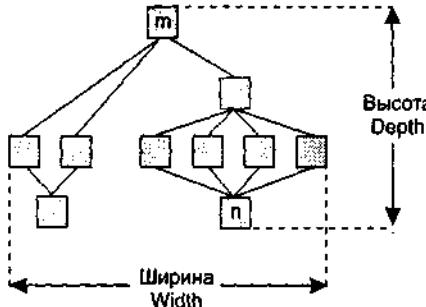


Рис. 4.17. Иерархическая структура программной системы

Первичными характеристиками являются количество вершин (модулей) и количество ребер (связей между модулями). К ним добавляются две глобальные характеристики — высота и ширина:

- **высота** — количество уровней управления;
- **ширина** — максимальное из количеств модулей, размещенных на уровнях управления.

В нашем примере высота = 4, ширина = 6.

Локальными характеристиками модулей структуры являются коэффициент объединения по входу и коэффициент разветвления по выходу.

Коэффициент объединения по входу $\text{Fan_in}(i)$ — это количество модулей, которые прямо управляют i -м модулем.

В примере для модуля n : $\text{Fan_in}(n)=4$.

Коэффициент разветвления по выходу $\text{Fan_out}(i)$ — это количество модулей, которыми прямо управляет i -й модуль.

В примере для модуля m : $\text{Fan_out}(m)=3$.

Возникает вопрос: как оценить качество структуры? 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.

Л. Констентайн и Э. Йордан (1979) предложили оценивать структуру с помощью коэффициентов $Fan_in(i)$ и $Fan_out(i)$ модулей [77].

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 $ifan_out(j)$ [35]. 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).

Контрольные вопросы

1. Какова цель синтеза программной системы? Перечислите этапы синтеза.
2. Дайте определение разработки данных, разработки архитектуры и процедурной разработки.
3. Какие особенности имеет этап проектирования?
4. Решение каких задач обеспечивает предварительное проектирование?
5. Какие модели системного структурирования вы знаете?
6. Чем отличается модель клиент-сервер от трехуровневой модели?
7. Какие типы моделей управления вы знаете?
8. Какие существуют разновидности моделей централизованного управления?
9. Поясните разновидности моделей событийного управления.
10. Поясните понятия модуля и модульности. Зачем используют модули?
11. В чем состоит принцип информационной закрытости? Какие достоинства он имеет?

12. Что такое связность модуля?
13. Какие существуют типы связности?
14. Дайте характеристику функциональной связности.
15. Дайте характеристику информационной связности.
16. Охарактеризуйте коммуникативную связность.
17. Охарактеризуйте процедурную связность.
18. Дайте характеристику временной связности.
19. Дайте характеристику логической связности.
20. Охарактеризуйте связность по совпадению.
21. Что значит «улучшать связность»?
22. Что такое сцепление модуля?
23. Какие существуют типы сцепления?
24. Дайте характеристику сцепления по данным.
25. Дайте характеристику сцепления по образцу.
26. Охарактеризуйте сцепление по управлению.
27. Охарактеризуйте сцепление по внешним ссылкам.
28. Дайте характеристику сцепления по общей области.
29. Дайте характеристику сцепления по содержанию.
30. Что значит «улучшать сцепление»?
31. Какие подходы к оценке сложности системы вы знаете?
32. Что определяет иерархическая структура программной системы?
33. Поясните первичные характеристики иерархической структуры.
34. Поясните понятия коэффициента объединения по входу и коэффициента разветвления по выходу.
35. Что определяет невязка структуры?
36. Поясните информационные коэффициенты объединения и разветвления.

ГЛАВА 5. 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

Исходными данными для метода структурного проектирования являются компоненты модели анализа ПС, которая представляется иерархией диаграмм потоков данных [34], [52], [58], [73], [77]. Результат структурного проектирования — иерархическая структура ПС. Действия структурного проектирования зависят от типа информационного потока в модели анализа.

Типы информационных потоков

Различают 2 типа информационных потоков:

- 1) поток преобразований;
- 2) поток запросов.

Как показано на рис. 5.1, в потоке преобразований выделяют 3 элемента: Входящий поток, Преобразуемый поток и Выходящий поток.

Потоки запросов имеют в своем составе особые элементы — запросы.

Назначение элемента-запроса состоит в том, чтобы запустить поток данных по одному из нескольких путей. Анализ запроса и переключение потока данных на один из путей действий происходит в центре

запросов.

Структуру потока запроса иллюстрирует рис. 5.2.



Рис. 5.1. Элементы потока преобразований

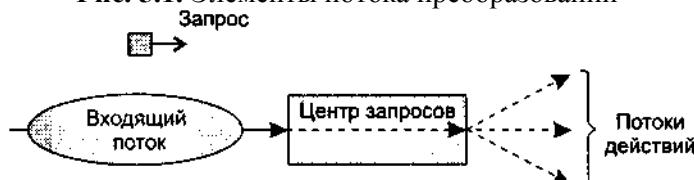


Рис. 5.2. Структура потока запроса

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 потоков, 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. 5.3.

Диаграмма потоков данных ПДД

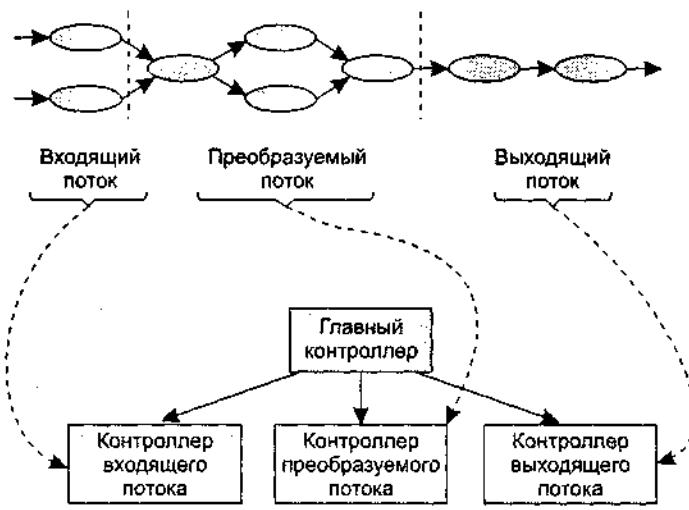


Рис. 5.3. Начальная структура ПС для потока «преобразование»

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. 5.4).

Центр преобразования ПДД отображается иначе (рис. 5.5). Каждый преобразователь отображается в модуль, непосредственно подчиненный контроллеру центра.

Проходит преобразуемый поток 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.

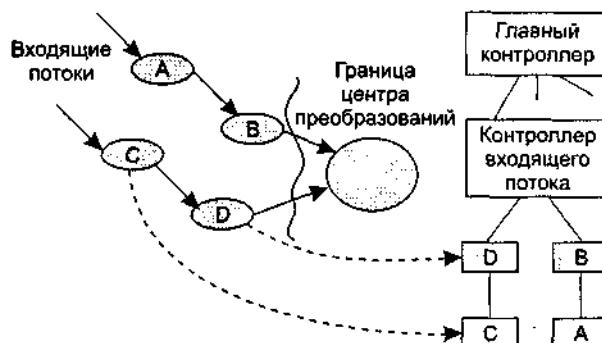


Рис. 5.4. Отображение преобразователей ПДД в модули структуры

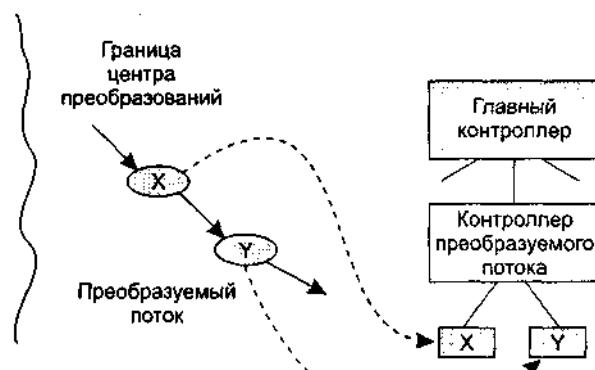


Рис. 5.5. Отображение центра преобразования ПДД

Для каждого модуля полученной структуры на базе спецификаций процессов модели анализа пишется сокращенное описание обработки.

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.

Проектирование для потока данных типа «запрос»

Шаг 1. Проверка основной системной модели. Модель включает: контекстную диаграмму ПДДО, 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.

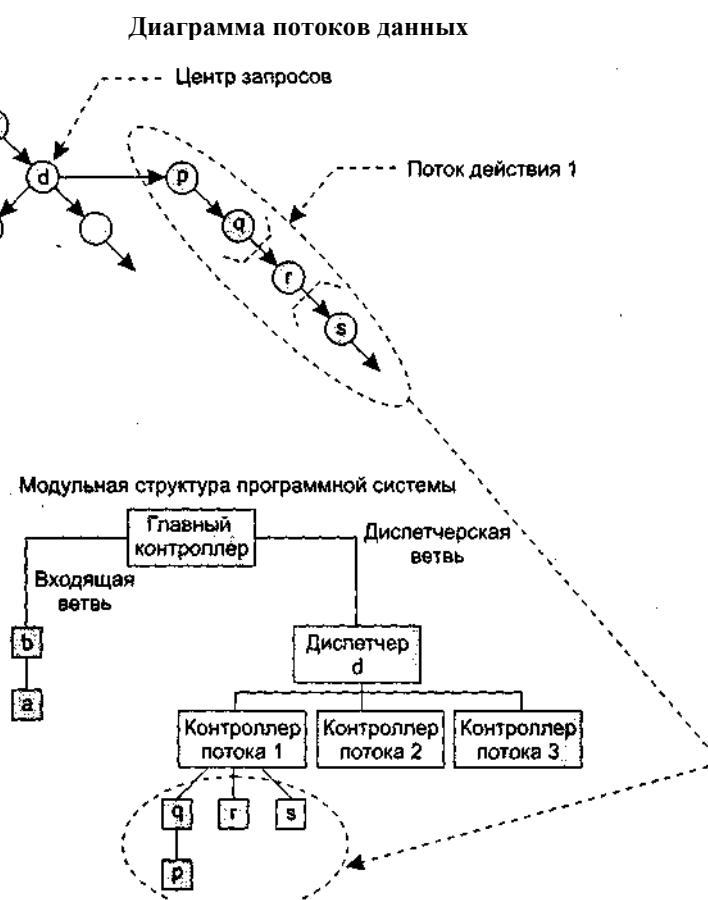


Рис. 5.6. Отображение в модульную структуру ПС потока действия 1

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 запросов. На рис. 5.6 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.

Метод Джексона включает шесть шагов [39]. 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 3 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(I);

опрос SV;

ЖДАТЬ цикл ПОКА ПРИБЫЛ(1)

 опрос SV;

 конец ЖДАТЬ;

 LOFF(I);

 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 как 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:

TRAHСПОРТ-1

LON(I);
опрос SV;

ЖДАТЬ цикл ПОКА ПРИБЫЛ(1)
 опрос SV;

конец ЖДАТЬ;

LOFF(I);

Покинуть(1);

TRAHЗИТ цикл ПОКА УБЫЛ(1)

 опрос SV;

конец ТРАНЗИТ;

TRAHСПОРТ цикл

ОСТАНОВКА;

 прибыть(i):

 записать Прибыл в S1D;

 LON(i);

 ЖДАТЬ цикл ПОКА ПРИБЫЛ(i)

 опрос SV;

 конец ЖДАТЬ;

 LOFF(i);

 покинуть(i);

 TRAHЗИТ 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. Связем 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. 5.7.



Рис. 5.7. Диаграмма дополнительного процесса КНОПКА-2

Внешние связи модели КНОПКА-2 должны включать:

- ❑ одно соединение моделью КНОПКА-1 — организуется через поток данных BID (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 В1D;

ГрНАЖ цикл

 ЖдатьНАЖ цикл ПОКА Не НАЖАТА

 читать В1D;

 конец ЖдатьНАЖ;

 Запрос := ДА;

 читать МВД;

 ЖдатьОБСЛУЖ цикл ПОКА Не ПРИБЫЛ

 читать МВД;

 конец ЖдатьОБСЛУЖ;

 Запрос := НЕТ; читать В1D;

 конец ГрНАЖ;

конец КНОПКА-2;

Диаграмма системной спецификации, отражающая все изменения, представлена на рис. 5.8.

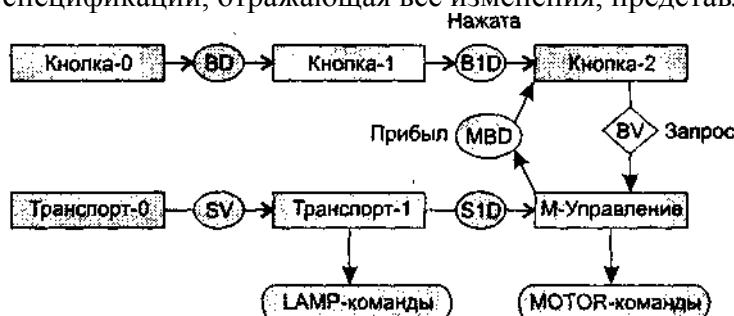


Рис. 5.8. Полная диаграмма системной спецификации

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:

- временной интервал на выработку команды STOP; он должен выбираться путем анализа скорости транспорта и ограничения мощности;
- 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

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. Какие шаги предусматривает метод Джексона на этапе проектирования?
16. В чем состоит суть развития диаграммы системной спецификации Джексона?
17. Поясните понятие встроенной функции.
18. Поясните понятие функции впечатления.
19. Поясните понятие функции диалога.
20. В чем состоит учет системного времени (в методе Джексона)?

ГЛАВА 6. 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 — 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 раскрыты 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. 6.1.



Рис. 6.1. 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 оценивания 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.

Существуют 2 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.

Как показано на рис. 6.2, основное место приложения тестов «черного ящика» — интерфейс ПО.

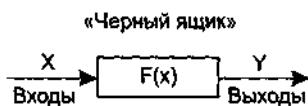


Рис. 6.2. Тестирование «черного ящика»

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 (рис. 6.3).

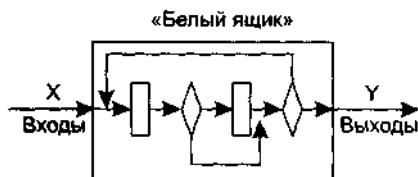


Рис. 6.3. Тестирование «белого ящика»

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 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 [2], [13]. 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;
- проходят 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 $m = 10^{14}$. 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 максимальное 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) [49].

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 график

Для представления программы используется потоковый график. 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.
4. 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).
5. Например, фрагмент программы

if a OR b

```
  then x
  else y
end if;
```

вместо прямого отображения в потоковый график вида, показанного на рис. 6.4, отображается в

преобразованный потоковый граф (рис. 6.5).

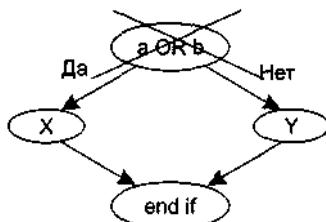


Рис. 6.4. Прямое отображение в потоковый график

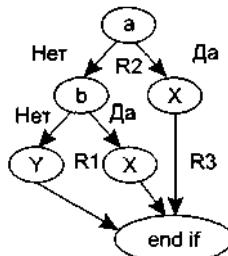


Рис. 6.5. Преобразованный потоковый график

6. Замкнутые области, образованные дугами и узлами, называют регионами.
7. Окружающая среда рассматривается как дополнительный регион. Например, показанный здесь график имеет три региона — R1, R2, R3.

Пример 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 а >= поля b
5              то удалить b;
6              иначе удалить а;
7a          конец если;
7a          конец если;
7b          конец выполнять;
8      конец сжатие;

```

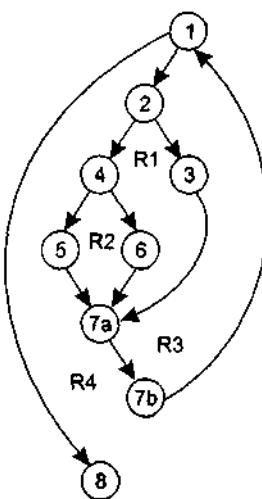


Рис. 6.6. Преобразованный потоковый график процедуры сжатия

Она отображается в потоковый график, представленный на рис. 6.6. Видим, что этот потоковый график 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 программы;
- верхнюю оценку количества тестов, которое гарантирует однократное выполнение всех операторов.

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 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.

Свойства базового множества:

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 1 каждым из трех способов:

1) потоковый график 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 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. 6.7).

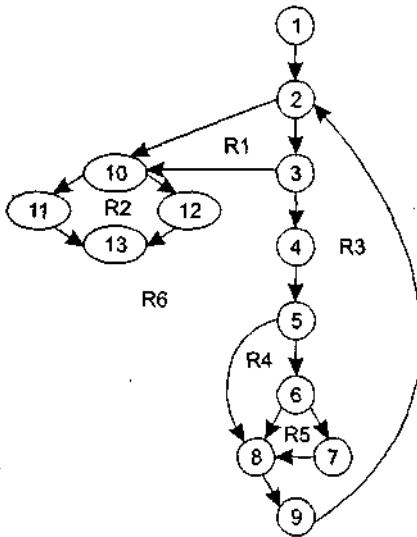


Рис. 6.7. Потоковый график процедуры вычисления среднего значения

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, 5, 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 $2 < i < 500$.

ОЖ.РЕЗ.: корректное усреднение основывается на k величинах и правильном подсчете.

ПРИМЕЧАНИЕ

Путь не может тестируться самостоятельно, а должен тестируться как часть путей 4, 5, 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 $i \leq 500$; 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 $i \leq 500$; вел(k) > макс, где $k < i$.

ОЖ.РЕЗ.: корректное усреднение основывается на n величинах и правильном подсчете.

Тестовый вариант для пути 6 **ТВ6**:

ИД: вел(i)=допустимое значение, где $i \leq 500$.

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).

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 <оператор отношения> E2,

где E1, E2 — арифметические выражения, а в качестве оператора отношения используется один из следующих операторов: $<$, $>$, $=$, \neq , \leq , \geq .

Составное условие состоит из нескольких простых условий, булевых операторов и круглых скобок. Будем применять булевые операторы OR, AND (&), NOT. Условия, не содержащие выражений отношения, называют булевыми выражениями.

Таким образом, элементами условия являются: булев оператор, булева переменная, пара скобок (заключающая простое или составное условие), оператор 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 булева оператора (наличие некорректных / отсутствующих / избыточных булевых операторов);
- ошибка булевой переменной;
- ошибка булевой скобки;
- ошибка оператора 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 А эффективен для обнаружения ошибок в условиях, содержащихся в А, то вероятно, что этот набор также эффективен для обнаружения других ошибок в А. 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

$E1 <\text{оператор отношения}> E2$

проверяется тремя тестами, которые формируют значение $E1$ большим, чем $E2$, равным $E2$ и меньшим, чем $E2$.

Если оператор отношения неправилен, а $E1$ и $E2$ корректны, то эти три теста гарантируют обнаружение ошибки оператора 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 операторов, 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 операторов отношений

Способ тестирования ветвей и операторов отношений (автор К. Таи, 1989) обнаруживает ошибки ветвления и операторов отношения в условии, для которого выполняются следующие ограничения [72]:

- все булевые переменные и операторы 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:

$$OY_c = (d_1, d_2, d_3, \dots, d_n),$$

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 OY_c (для условия *C*) покрывается выполнением *C*, если в ходе этого выполнения результат каждого простого условия в *C* удовлетворяет соответствующему ограничению в OY_c .

На основе ограничения условия OY создается ограничивающее множество ОМ, элементы которого являются сочетаниями всех возможных значений $d_1, d_2, d_3, \dots, d_n$.

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

$$(\text{true}, \text{true}, \text{true}),$$

а в терминах ограничений на значения аргументов простых условий — записи
 $(\text{true}, >, \text{true})$.

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 ОМ задает отдельный тестовый вариант. Исходные данные тестового варианта должны обеспечить соответствующую комбинацию значений простых условий, а ожидаемый результат равен значению составного условия.

Пример 1. В качестве примера рассмотрим два типовых составных условия:

$$C_{\&} = a \& b, C_{\text{or}} = a \text{ or } b,$$

где a и b — булевые переменные. Соответствующие ограничения условий принимают вид

$$\text{ОУ}_{\&} = (d_1, d_2), \text{ОУ}_{\text{or}} = (d_1, d_2),$$

где $d_1 = d_2 = (\text{true}, \text{false})$.

Ограничивающие множества удобно строить с помощью таблицы истинности (табл. 6.1).

Таблица 6.1. Таблица истинности логических операций

Вариант	a	b	$a \& b$	$a \text{ or } b$
1	false	false	false	false
2	false	true	false	true
3	true	false	false	true
4	true	true	true	true

Видим, что таблица задает в ОМ четыре элемента (и соответственно, четыре тестовых варианта). Зададим вопрос — каковы возможности минимизации? Можно ли уменьшить количество элементов в ОМ?

С точки зрения тестирования, нам надо оценивать 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 булева оператора, 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:

$$\text{ОМ}_{\&} = \{(\text{false}, \text{true}), (\text{true}, \text{false}), (\text{true}, \text{true})\};$$

- для условия типа ИЛИ ($a \text{ or } b$) варианты 2 и 3 поглощают вариант 4. Поэтому ограничивающее множество имеет вид:

$$\text{ОМ}_{\text{or}} = \{(\text{false}, \text{false}), (\text{false}, \text{true}), (\text{true}, \text{false})\}.$$

Рассмотрим шаги **способа тестирования ветвей и операторов 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 $\text{ОМ}_{\&}$ или ОМ_{or} выявленных ограничений результата;
- 4) для каждого элемента ОМ разрабатывается тестовый вариант.

Пример 2. Рассмотрим составное условие $C1$ вида:

$$B_1 \& (E_1, E_2),$$

где B_1 — булево выражение, E_1, E_2 — арифметические выражения.

Ограничение составного условия имеет вид

$$\text{ОУ}_{C_1} = (d_1, d_2),$$

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

$$\text{OM}_{\&} = \{(\text{false}, \text{true}), (\text{true}, \text{false}), (\text{true}, \text{true})\}.$$

Заметим, что *true* для $(E_1 = E_2)$ означает $=$, а *false* для $(E_1 = E_2)$ означает или $<$, или $>$. Заменяя $(\text{true}, \text{true})$ и $(\text{false}, \text{true})$, ограничениями $(\text{true}, =)$ и $(\text{false}, =)$ соответственно, а $(\text{true}, \text{false})$ — ограничениями $(\text{true}, <)$ и $(\text{true}, >)$, получаем ограничивающее множество для C_1 :

$$\text{OM}_{C_1} = \{(\text{false}, =), (\text{true}, <), (\text{true}, >), (\text{true}, =)\}.$$

Покрытие этого множества гарантирует обнаружение ошибок булевых операторов и операторов отношения в C_1 .

Пример 3. 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

$$\text{OU}_{C_2} = (d_1, d_2),$$

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} :

$$\text{OM}_{C_2} = \{ (=, =), (<, =), (>, <), (>, >), (>, =) \}.$$

Покрытие этого ограничивающего множества гарантирует обнаружение ошибок операторов отношения в C_2 .

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. 6.8. 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 6.

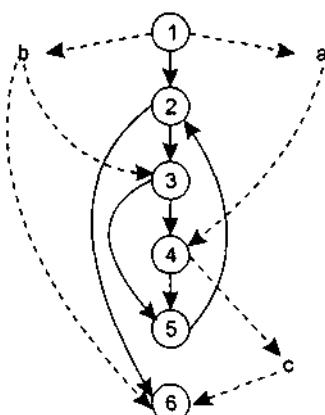


Рис. 6.8. Граф программы с управляющими и информационными связями

В общем случае для каждой вершины графа можно записать:

- ❑ множество определений данных
 $\text{DEF}(i) = \{ x \mid i - \text{я вершина содержит определение } x \};$

- ❑ множество использований данных:

$$\text{USE}(i) = \{ x \mid i - \text{я вершина использует } x \}.$$

Под *определением данных* понимают действия, изменяющие элемент данных. Признак определения — имя элемента стоит в левой части оператора присваивания:

$$x := f(\dots).$$

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).

Назовём *DU-цепочкой* (*цепочкой определения-использования*) конструкцию $[x, i, j]$, где i, j — имена вершин; x определена в i -й вершине ($x \in \text{DEF}(i)$) и используется в j -й вершине ($x \in \text{USE}(j)$).

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. 6.9);

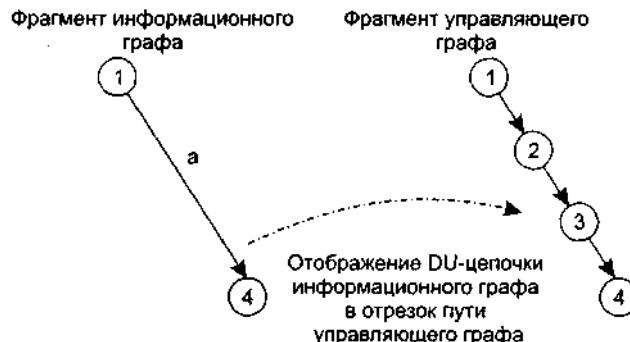


Рис. 6.9. Отображение DU-цепочки в отрезок пути

- 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.

Недостаток DU-тестирования: трудности в выборе минимального количества максимально эффективных тестов.

Область использования DU-тестирования: программы с вложенными условными операторами и операторами цикла.

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. 6.10.

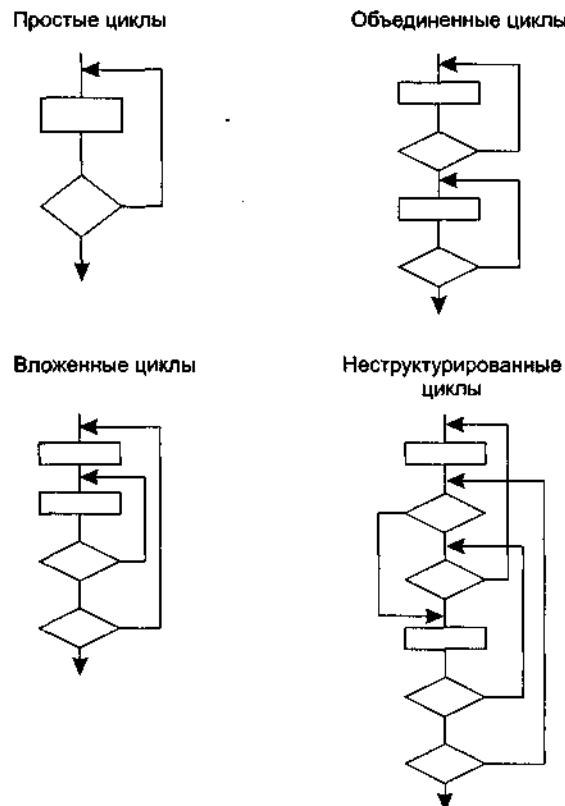


Рис. 6.10. Типовые структуры циклов

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.

Вложенные циклы

С увеличением уровня вложенности циклов количество возможных путей резко возрастает. Это приводит к нереализуемому количеству тестов [13]. Для сокращения количества тестов применяется специальная методика, в которой используются такие понятия, как объемлющий и вложенный циклы (рис. 6.11).

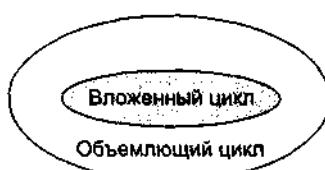


Рис. 6.11. Объемлющий и вложенный циклы

Порядок тестирования вложенных циклов иллюстрирует рис. 6.12.



Рис. 6.12. 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.

Контрольные вопросы

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 операторов 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 операторов отношений.
29. Поясните достоинства, недостатки и область применения способа тестирования ветвей и операторов 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. Поясните особенности тестирования циклов.
37. Какие методики тестирования простых циклов вы знаете?
38. Каковы шаги тестирования вложенных циклов?

ГЛАВА 7. 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 [14]. 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. 7.1, 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*.

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.

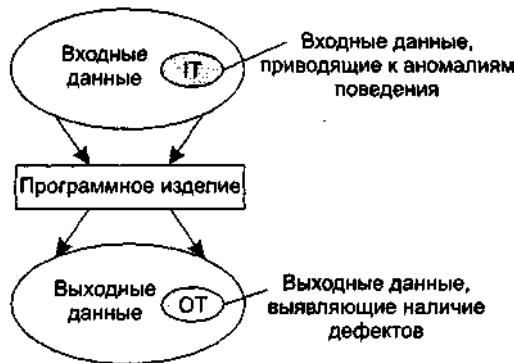


Рис. 7.1. Тестирование «черного ящика»

Принцип «черного ящика» не альтернативен принципу «белого ящика». Скорее это дополняющий подход, который обнаруживает другой класс ошибок.

Тестирование «черного ящика» обеспечивает поиск следующих категорий ошибок:

- 1) некорректных или отсутствующих функций;
- 2) ошибок интерфейса;
- 3) ошибок во внешних структурах данных или в доступе к внешней базе данных;
- 4) ошибок характеристик (необходимая емкость памяти и т. д.);
- 5) ошибок инициализации и завершения.

Подобные категории ошибок способами «белого ящика» не выявляются.

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» [3], [14].

В этом способе входная область данных программы *делится* на классы эквивалентности. Для каждого класса эквивалентности разрабатывается один тестовый вариант.

Класс эквивалентности — 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. 7.2 каждый класс эквивалентности показан эллипсом. Здесь выделены входные классы эквивалентности допустимых и недопустимых исходных данных, а также классы результатов.

Классы эквивалентности могут быть определены по спецификации на программу.



Рис. 7.2. 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 $n \dots m$, то определяются один допустимый и два недопустимых класса эквивалентности:
 - $V_Class=\{n \dots m\}$ — допустимый класс эквивалентности;
 - $Inv_Class1=\{x| \text{для любого } x: x < n\}$ — первый недопустимый класс эквивалентности;
 - $Inv_Class2=\{y| \text{для любого } y: y > m\}$ — второй недопустимый класс эквивалентности.
2. Если условие ввода задает конкретное значение a , то определяется один допустимый и два недопустимых класса эквивалентности:
 - $V_Class=\{a\}$;
 - $Inv_Class1=\{x| \text{для любого } x: x < a\}$;
 - $Inv_Class2=\{y| \text{для любого } y: y > a\}$.
3. Если условие ввода задает множество значений $\{a, b, c\}$, то определяются один допустимый и один недопустимый класс эквивалентности:
 - $V_Class=\{a, b, c\}$;
 - $Inv_Class=\{x| \text{для любого } x: (x \neq a) \& (x \neq b) \& (x \neq c)\}$.
4. Если условие ввода задает булево значение, например true, то определяются 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 [3], [14], [69]. 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 $n \dots m$, 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 $-1,0 \dots +1,0$, 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 тестиировать 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.

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, 4-м 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. 7.3.

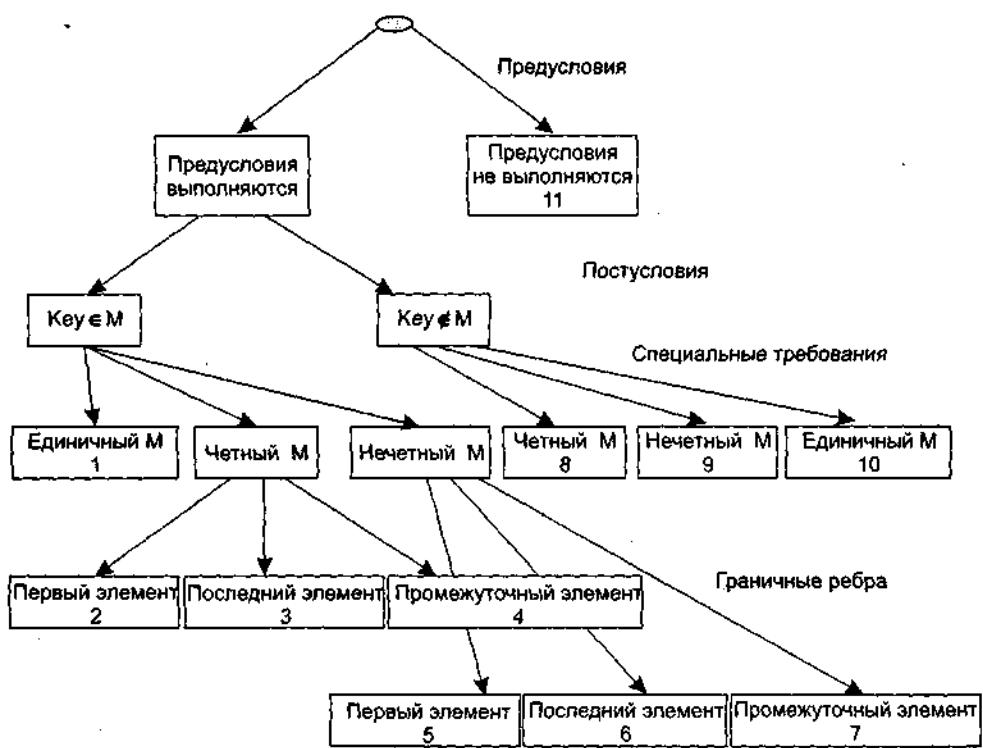


Рис. 7.3. Дерево разбиений области исходных данных бинарного поиска

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=7.

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: М=15, 10, 5, 25, 20, 40, 35; Key=35.

ОЖ.РЕЗ.: Аварийное донесение: Массив не упорядочен.

Способ диаграмм причин-следствий

Диаграммы причинно-следственных связей — способ проектирования тестовых вариантов, который обеспечивает формальную запись логических условий и соответствующих действий [3], [64]. 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. 7.4) устанавливает, что если значение c_1 есть 1, то и значение e_1 есть 1; в противном случае значение e_1 есть 0.



Рис. 7.4. Функция тождество

Функция не (рис. 7.5) устанавливает, что если значение c_1 есть 1, то значение e_1 есть 0; в противном случае значение e_1 есть 1.

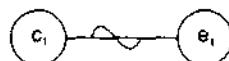


Рис. 7.5. Функция не

Функция или (рис. 7.6) устанавливает, что если c_1 или c_2 есть 1, то e_1 есть 1, в противном случае e_1 есть 0.

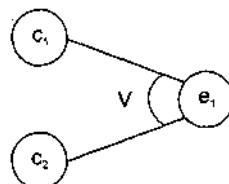


Рис. 7.6. Функция или

Функция и (рис. 7.7) устанавливает, что если и c_1 и c_2 есть 1, то e_1 есть 1, в противном случае e_1 есть 0.

Часто определенные комбинации причин невозможны из-за синтаксических или внешних ограничений. Используются перечисленные ниже обозначения ограничений.

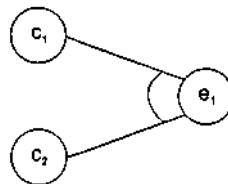


Рис. 7.7. Функция и

Ограничение Е (исключает, Exclusive, рис. 7.8) устанавливает, что Е должно быть истинным, если

хотя бы одна из причин — a или b — принимает значение 1 (a и b не могут принимать значение 1 одновременно).

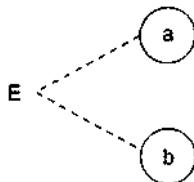


Рис. 7.8. Ограничение E (исключает, Exclusive)

Ограничение I (включает, Inclusive, рис. 7.9) 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).

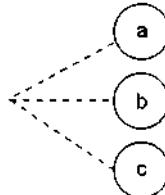


Рис. 7.9. Ограничение I (включает, Inclusive)

Ограничение O (одно и только одно, Only one, рис. 7.10) устанавливает, что одна и только одна из величин a или b должна быть равна 1.

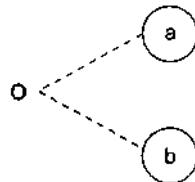


Рис. 7.10. Ограничение O (одно и только одно, Only one)

Ограничение R (требует, Requires, рис. 7.11) устанавливает, что если a принимает значение 1, то и b должна принимать значение 1 (нельзя, чтобы a было равно 1, а b - 0).

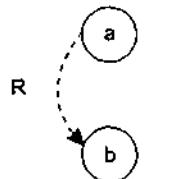


Рис. 7.11. Ограничение R (требует, Requires)

Часто возникает необходимость в ограничениях для следствий.

Ограничение M (скрывает, Masks, рис. 7.12) устанавливает, что если следствие a имеет значение 1, то следствие b должно принять значение 0.

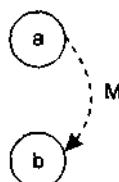


Рис. 7.12. Ограничение M (скрывает, Masks)

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. 7.13).

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.

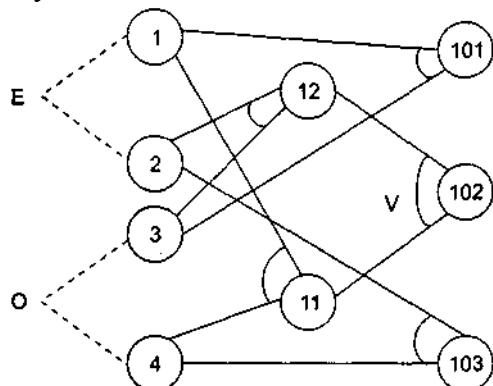


Рис. 7.13. Граф причинно-следственных связей

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 определяются 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. 7.1.

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.

Таблица 7.1. Таблица решений для расчета оплаты за электричество

Номера столбцов — >		1	2	3	4
Условия	Причины	1	1	0	1
		2	0	1	0
		3	1	1	0
		4	0	0	1
	Вторичные причины	11	0	0	1
		12	0	1	0
Действия	Следствия	101	1	0	0
		102	0	1	1
		103	0	0	1

Контрольные вопросы

1. Каковы особенности тестирования методом «черного ящика»?
2. Какие категории ошибок выявляет тестирование методом «черного ящика»?
3. Какие достоинства имеет тестирование методом «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 такая 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?

ГЛАВА 8. ОРГАНИЗАЦИЯ ПРОЦЕССА ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В этой главе излагаются вопросы, связанные с проведением тестирования на всех этапах конструирования 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 проверяются 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) [3], [13], [64], [69]. Методика тестирования ПС может быть представлена в виде разворачивающейся спирали (рис. 8.1).

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 *системного анализа* ПС.

Охарактеризуем каждый шаг процесса тестирования.

1. *Тестирование элементов*. Цель — индивидуальная проверка каждого модуля. Используются способы тестирования «белого ящика».



Рис. 8.1. Спираль процесса тестирования ПС

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%-ной 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:

$$\lambda(t) = \frac{\lambda_0}{\lambda_0 \times p \times t + 1}, \quad (8.1)$$

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 устранимых ошибок; t — время тестирования.

С помощью уравнения (8.1) можно предсказать снижение ошибок в ходе тестирования, а также время, требующееся 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 [3].

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 символьическое 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. 8.2.

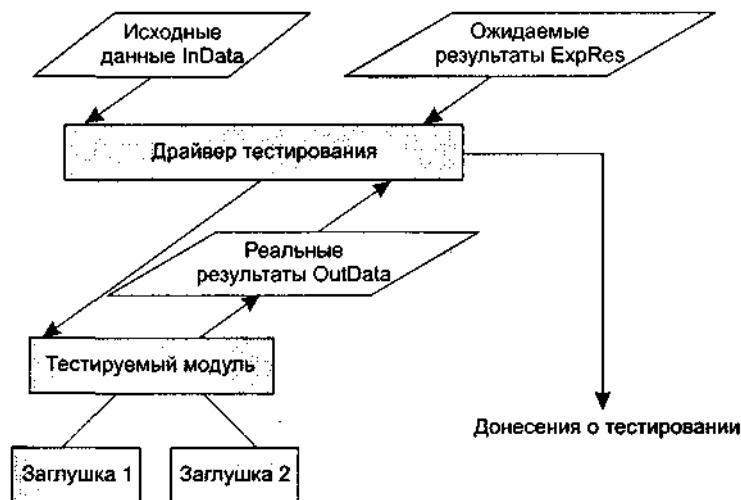


Рис. 8.2. Программная среда для тестирования модуля

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. 8.3.

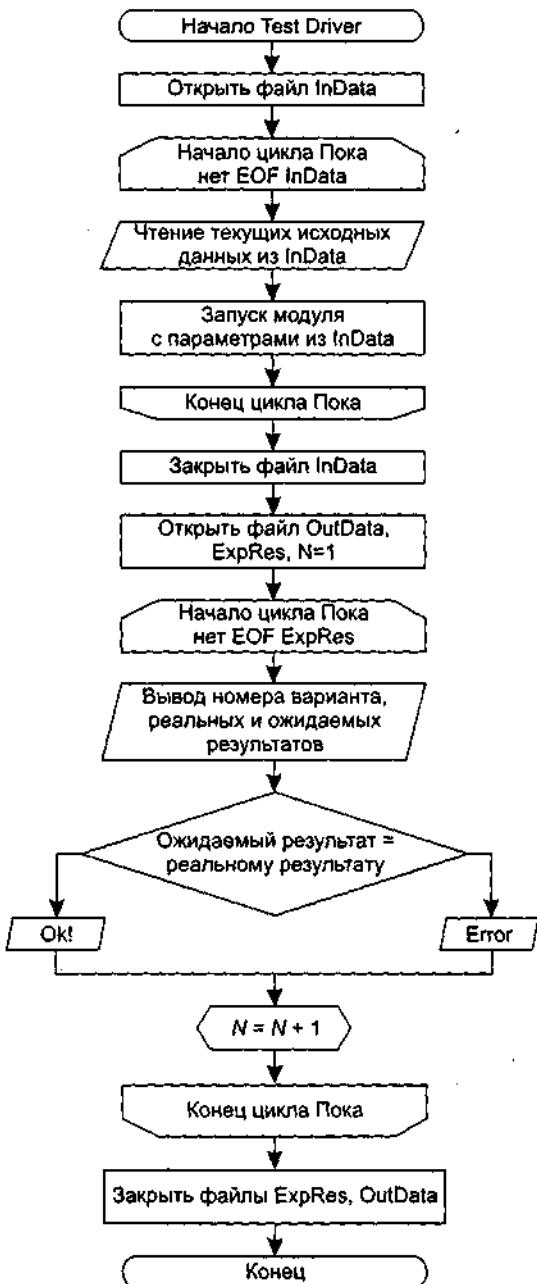


Рис. 8.3. Алгоритм работы драйвера тестирования

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 [3].

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. 8.4). Интеграция поиском в глубину будет подключать все модули, находящиеся на главном управляющем пути структуры (по вертикали). Выбор главного управляющего пути отчасти произведен и зависит от характеристик, определяемых приложением. Например, при выборе левого пути прежде всего будут подключены модули M1, M2, M5. Следующим подключается модуль M8 или Mб (если это необходимо для правильного функционирования M2). Затем строится центральный или правый управляющий путь.

При интеграции поиском в ширину структура последовательно проходит по уровням-горизонтам. 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, Mб и т. д.

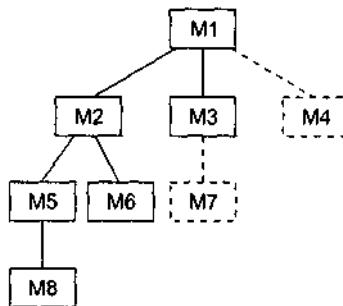


Рис. 8.4. Нисходящая интеграция системы

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 А — отображает трассируемое сообщение;
- заглушка В — отображает проходящий параметр;
- заглушка С — возвращает величину из таблицы;
- заглушка D — выполняет табличный поиск по ключу (входному параметру) и возвращает связанный с ним выходной параметр.

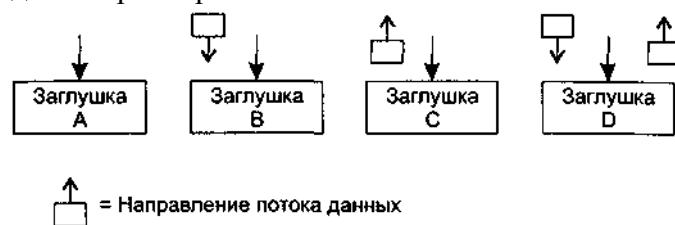


Рис. 8.5. Категории заглушек

Категории заглушек представлены на рис. 8.5.

Очевидно, что заглушка А 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 сборка и тестирование системы начинаются 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. 8.6.

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 А — вызывает подчиненный модуль;
- драйвер В — посылает элемент данных (параметр) из внутренней таблицы;
- драйвер С — отображает параметр из подчиненного модуля;
- драйвер D — является комбинацией драйверов В и С.

Очевидно, что драйвер А наиболее прост, а драйвер D наиболее сложен в реализации. Различные типы драйверов представлены на рис. 8.7.

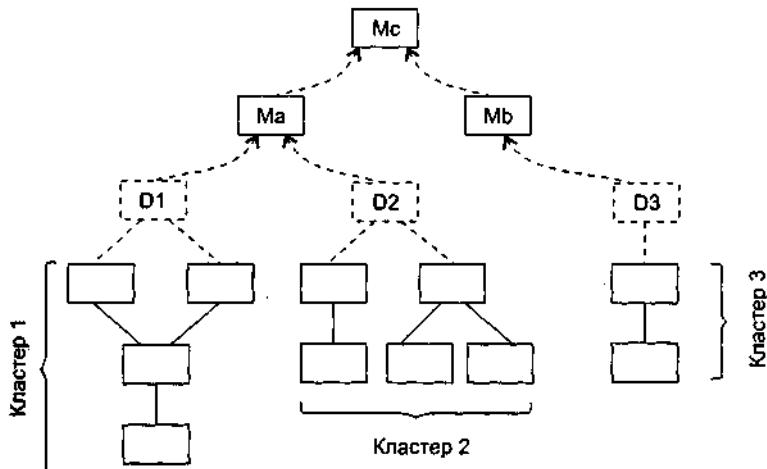


Рис. 8.6. Восходящая интеграция системы

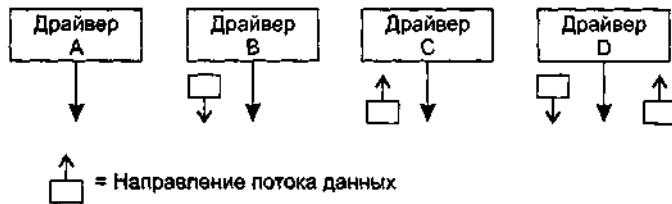


Рис. 8.7. Различные типы драйверов

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 [3], [13].

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 программа 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 [64], [69].

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. Минимальная конфигурация ПС включает следующие базовые элементы:

- 1) системную спецификацию;
- 2) план программного проекта;
- 3) спецификацию требований к ПС; работающий или бумажный макет;
- 4) предварительное руководство пользователя;
- 5) спецификация проектирования;
- 6) листинги исходных текстов программ;
- 7) план и методику тестирования; тестовые варианты и полученные результаты;
- 8) руководства по работе и инсталляции;
- 9) ехе-код выполняемой программы;
- 10) описание базы данных;
- 11) руководство пользователя по настройке;
- 12) документы сопровождения; отчеты о проблемах ПС; запросы сопровождения; отчеты о конструкторских изменениях;
- 13) стандарты и методики конструирования 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 системного элемента 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 [13], [52].

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 оцениваются 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 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. Стressовые 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 результатами. Несоответствие является симптомом скрытой причины. Процесс 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.

Возможные 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 устраниет 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» [3], [64].

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, подозреваемую 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 конструирования оно проверяет?
3. Когда и зачем выполняется тестирование интеграции? Какой этап конструирования оно проверяет?
4. Когда и зачем выполняется тестирование правильности? Какой этап конструирования оно проверяет?
5. Когда и зачем выполняется системное тестирование? Какой этап конструирования 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 9. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРЕДСТАВЛЕНИЯ ПРОГРАММНЫХ СИСТЕМ

Девятая глава вводит в круг вопросов объектно-ориентированного представления программных систем. В этой главе рассматриваются: абстрагирование понятий проблемной области, приводящее к формированию классов; инкапсуляция объектов, обеспечивающая скрытность 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.

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 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 [22], [32], [41], [59], [64], [66].

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: физический объект — датчик скорости, устанавливаемый на борту летательного аппарата

(ЛА). Создадим его абстракцию. Для этого сформулируем обязанности датчика:

- знать проекцию скорости ЛА в заданном направлении;
- показывать текущую скорость;
- подвергаться настройке.

Теперь опишем абстракцию датчика. Описание сформулируем как спецификацию класса на языке Ada 95 [4]:

```
Package Класс_ДатчикСкорости is
    subtype Скорость is Float range ...
    subtype Направление is Natural range ...
    type ДатчикСкорости is tagged private;
    function НовыйДатчик(нокер: Направление)
        return ДатчикСкорости;
    function ТекущаяСкорость (the: ДатчикСкорости)
        return Скорость;
    procedure Настраивать(the: in out ДатчикСкорости;
        ДействитСкорость: Скорость);
private — закрытая часть спецификации
-- полное описание типа ДатчикСкорости
end Класс_ДатчикСкорости;
```

Здесь Скорость и Направление — вспомогательные подтипы, обеспечивающие задание операций абстракции (НовыйДатчик, ТекущаяСкорость, Настраивать). Приведенная абстракция — это только спецификация класса датчика, настоящее его представление скрыто в приватной части спецификации и теле класса. Класс ДатчикСкорости — еще не объект. Собственно датчики — это его экземпляры, и их нужно создать, прежде чем с ними можно будет работать. Например, можно написать так:

```
ДатчикПродольнойСкорости : ДатчикСкорости;
ДатчикПоперечнойСкорости : ДатчикСкорости;
ДатчикНормальнойСкорости : ДатчикСкорости;
```

Инкапсуляция

Инкапсуляция и абстракция — взаимодополняющие понятия: абстракция выделяет внешнее поведение объекта, а инкапсуляция содержит и скрывает реализацию, которая обеспечивает это поведение. Инкапсуляция достигается с помощью информационной закрытости. Обычно скрываются структура объектов и реализация их методов.

Инкапсуляция является процессом разделения элементов абстракции на секции с различной видимостью. Инкапсуляция служит для отделения интерфейса абстракции от ее реализации.

Пример: физический объект регулятор скорости.

Обязанности регулятора:

- включаться;
- выключаться;
- увеличивать скорость;
- уменьшать скорость;
- отображать свое состояние.

Спецификация класса Регулятор скорости примет вид

```
with Класс_ДатчикСкорости. Класс_Порт;
use Класс_ДатчикСкорости. Класс_Порт;
Package Класс_РегуляторСкорости is
    type Режим is (Увеличение, Уменьшение);
    subtype Размещение is Natural range ...
    type РегуляторСкорости is tagged private;
    function НовРегуляторСкорости (номер: Размещение;
        напр: Направление; порт; Порт)
        return РегуляторСкорости;
    procedure Включить(the: in out РегуляторСкорости);
    procedure Выключить(1пе: in out РегуляторСкорости);
    procedure УвеличитьСкорость(1г1е: in out
        РегуляторСкорости);
    procedure УменьшитьСкорость(the: in out
```

```

    РегуляторСкорости);
Function ОпросСостояния(the: РегуляторСкорости)
    return Режим;
private
    type укз_наПорт is access all Порт;
    type РегуляторСкорости is tagged record
        Номер; Размещение;
        Состояние; Режим;
        Управление: укз_наПорт;
    end record;
end Класс_РегуляторСкорости;

```

Здесь вспомогательный тип Режим используется для задания основного типа класса, класс ДатчикСкорости обеспечивает класс регулятора описанием вспомогательного типа Направление, класс Порт фиксирует абстракцию порта, через который посылаются сообщения для регулятора. Три свойства: Номер, Состояние, Управление — формулируют инкапсулируемое представление основного типа класса РегуляторСкорости. При попытке клиента получить доступ к этим свойствам фиксируется семантическая ошибка.

Полное инкапсулированное представление класса РегуляторСкорости включает описание реализаций его методов — оно содержится в теле класса. Описание тела для краткости здесь опущено.

Модульность

В языках C++, Object Pascal, Ada 95 абстракции классов и объектов формируют логическую структуру системы. При производстве физической структуры эти абстракции помещаются в модули. В больших системах, где классов сотни, 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.

Определение классов и объектов выполняется в ходе логической разработки, а определение модулей — в ходе физической разработки системы. Эти действия сильно взаимосвязаны, осуществляются итеративно.

В Ada 95 мощным средством обеспечения модульности является пакет.

Пример: пусть имеется несколько программ управления полетом летательного аппарата (ЛА) — программа угловой стабилизации ЛА и программа управления движением центра масс ЛА. Нужно создать модуль, чье назначение — собрать все эти программы. Возможны два способа.

1. Присоединение с помощью указателей контекста:

```

with Класс_УгловСтабил, Класс_ДвиженЦентраMass;
use Класс_УгловСтабил, Класс_ДвиженЦентраMass;
Package Класс_УпрПолетом is
    ...

```

2. Встраивание программ управления непосредственно в объединенный модуль:

```

Package Класс_УпрПолетом is
    type УгловСтабил is tagged private;
    type ДвиженЦентраMass is tagged private;
    -----

```

Иерархическая организация

Мы рассмотрели три механизма для борьбы со сложностью:

- абстракцию (она упрощает представление физического объекта);
- инкапсуляцию (закрывает детали внутреннего представления абстракций);
- модульность (дает путь группировки логически связанных абстракций).

Прекрасным дополнением к этим механизмам является иерархическая организация — формирование

из абстракций иерархической структуры. Определением иерархии в проекте упрощаются понимание проблем заказчика и их реализация — сложная система становится обозримой человеком.

Иерархическая организация задает размещение абстракций на различных уровнях описания системы.

Двумя важными инструментами иерархической организации в объектно-ориентированных системах являются:

- структура из классов («*is a*»-иерархия);
- структура из объектов («*part of*»-иерархия).

Чаще всего «*is a*»-иерархическая структура строится с помощью наследования. 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: 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:

```
with Класс_УпрПолетом1; use Класс_УпрПолетом1;
Package Класс_УпрПолетом2 is
    type Траектория is (Гибкая. Свободная);
    type УпрПолетом2 is new УпрПолетом1 with private;
    procedure Установиться: in out УпрПолетом2:
        тип: Траектория; ориентация : Углы;
        параметры: Координаты_Скорость; команды: График);
    procedure УсловияОтделенияЗступени (the: УпрПолетом2;
        критерий:КритерийОтделения);
    function ПрогнозОтделенияЗступени (the: УпрПолетом2)
        return БортовоеВремя;
    function ИсполнениеКоманд(the: УпрПолетом2)
        return Boolean;
private
    type УпрПолетом2 is new УпрПолетом1
        with record
            типТраектории: Траектория; доОтделения: БортовоеВремя;
            выполнениеКоманд: Boolean;
        end record;
end Класс_УпрПолетом2;
```

Видим, что класс УпрПолетом2 — это «*is a*»-разновидность класса УпрПолетом1, который называется родительским классом или суперклассом.

В класс УпрПолетом2 добавлены:

- авспомогательный тип Траектория;
- три новых свойства (типТраектории, доОтделения, выполнениеКоманд);
- три новых метода (УсловияОтделенияЗступени, ПрогнозОтделенияЗступени, ИсполнениеКоманд).

Кроме того, в классе УпрПолетом2 переопределен метод суперкласса Установить. Подразумевается, что в наследство классу УпрПолетом2 достался набор методов и свойств класса УпрПолетом1. В частности, тип УпрПолетом2 включает поля типа 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 связанных структуры;
- 2) наследование позволяет легко и многократно использовать эти общие группы в других абстракциях.

Приведем пример класса ИзмерительСУ (измеритель системы управления ЛА):

```

with Класс_НастройкаДатчиков; Класс_Датчик;
use Класс_НастройкаДатчиков, Класс_Датчик;
Package Класс_ИзмерительСУ is
    type ИзмерительСУ is tagged private;
        -- описание методов
private
    type укз_наДатчик is access all Датчик'Class;
    type ИзмерительСУ is record
        датчики: атгау(1..30) of укз_наДатчик;
        процедураНастройки: НастройкаДатчиков;
    end record;
end Класс_ИзмерительСУ;

```

Очевидно, что объекты типа 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 типа ИзмерительСУ и объекты типа Датчик имеют относительную независимость. Например, время жизни измерителя и датчиков независимо друг от друга. Напротив, объект типа НастройкаДатчиков физически включается в объект типа ИзмерительСУ и независимо существовать не может. Отсюда вывод — разрушая объект типа ИзмерительСУ, мы, в свою очередь, разрушаем экземпляра НастройкаДатчиков.

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.

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» взаимозаменямы. На рис. 9.1 приведен пример объекта по имени Стол, имеющего определенный набор свойств и операций.

Индивидуальность — это характеристика объекта, которая отличает его от всех других объектов.

Состояние объекта характеризуется перечнем всех свойств объекта и текущими значениями каждого из этих свойств (рис. 9.1).

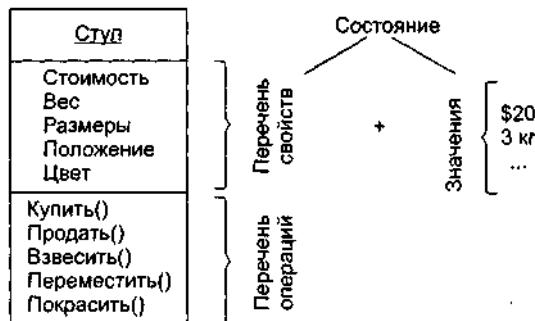


Рис. 9.1. Представление объекта с именем Стол

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 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 (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. 9.1.

Таблица 9.1. Разновидности операций

Вид операции	Пример операции
Модификатор	Пополнить (кг)
Селектор	КакойВес () : integer
Итератор	ПоказатьАссортиментТоваров () : string
Конструктор	СоздатьРобот (параметры)
Деструктор	УничтожитьРобот ()

В чистых объектно-ориентированных языках программирования операции могут объявляться только как методы — элементы классов, экземплярами которых являются объекты. Гибридные языки (C++, Ada 95) позволяют писать операции как свободные подпрограммы (вне классов). Соответствующие примеры показаны на рис. 9.2.



Рис. 9.2. Методы и свободные подпрограммы

В общем случае все методы и свободные подпрограммы, ассоциированные с конкретным объектом, образуют его *протокол*. Таким образом, протокол определяет оболочку допустимого поведения объекта и поэтому заключает в себе цельное (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. 9.3.

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.

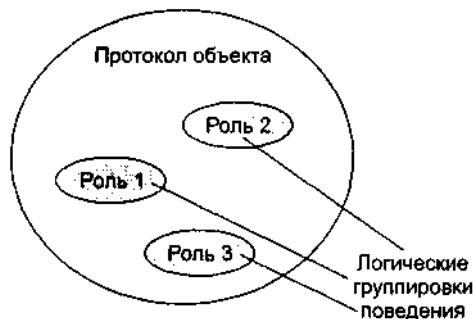


Рис. 9.3. Пространство поведения объекта

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. 9.4).

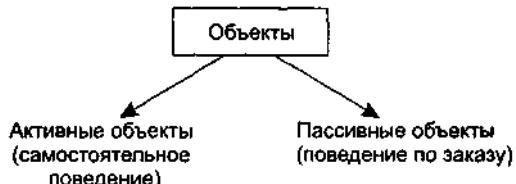


Рис.9.4. Активные и пассивные объекты

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.

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» [22]. 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. 9.5 с помощью соединительных линий. Связи представляют возможные пути для передачи сообщений. Сами сообщения показаны стрелками, отмечаяющими их направления, и помечены именами вызываемых операций.

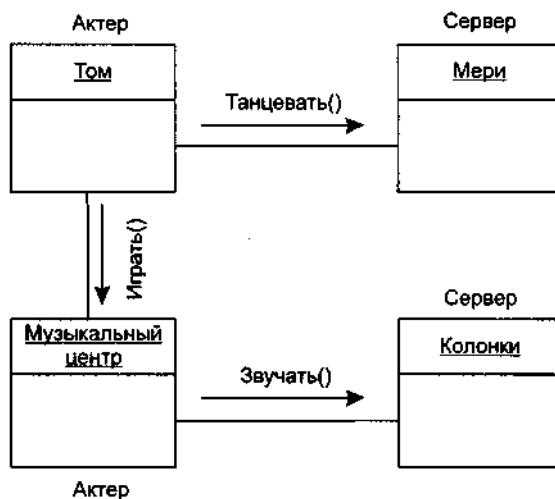


Рис. 9.5. Связи между объектами

Как участник связи объект может играть одну из трех ролей:

- актер — 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 актера или другого агента.

На рис. 9.5 Том — это актер, Мери, Колонки — серверы, Музикальный центр — агент.

Приведем пример. Допустим, что нужно обеспечить следующий график разворота первой ступени ракеты по углу тангажа, представленный на рис. 9.6.

Запишем абстракцию графика разворота:

```
with Класс_ДатчикУглаТангажа;
use Класс_ДатчикУглаТангажа;
Package Класс_ГрафикРазворота is
    subtype Секунда is Natural range ...
    type ГрафикРазворота is tagged private;
    procedure Очистить (the: in out ГрафикРазворота);
    procedure Связать (the: In out ГрафикРазворота;
        teta: Угол; si: Секунда: s2: Секунда);
    function УголНаМомент (the: ГрафикРазворота;
        s: Секунда) return Угол;
private
...
end Класс_ГрафикРазворота;
```

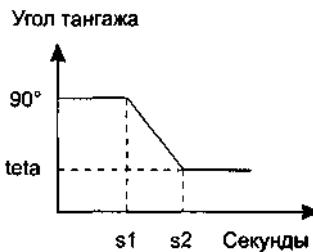


Рис. 9.6. График разворота первой ступени ракеты

Для решения задачи надо обеспечить сотрудничество трех объектов: экземпляра класса ГрафикРазворота, РегулятораУгла и КонтроллераУгла.

Описание класса КонтроллерУгла может иметь следующий вид:

```
with Класс_ГрафикРазворота. Класс_РегуляторУгла;
use Класс_ГрафикРазворота. Класс_РегуляторУгла;
Package Класс_КонтроллерУгла is
    type укз_наГрафик is access all ГрафикРазворота;
    type КонтроллерУгла is tagged private;
    procedure Обрабатывать (the: in out КонтроллерУгла;
        уг: укз_наГрафик);
    function Запланировано (the: КонтроллерУгла;
        уг: укз_наГрафик) return Секунда;
private
    type КонтроллерУгла is tagged record
        регулятор: РегуляторУгла := НовРегуляторУгла
            (1.1.10);
    ...
end Класс_КонтроллерУгла:
```

ПРИМЕЧАНИЕ

Операция Запланировано позволяет клиентам запросить у экземпляра КонтроллераУгла время обработки следующего графика.

И наконец, описание класса РегуляторУгла представим в следующей форме:

```
with Класс_ДатчикУгла. Класс_Порт;
use Класс_ДатчикУгла. Класс_Порт;
Package Класс_РегуляторУгла is
```

```

type Режим is (Увеличение, Уменьшение);
subtype Размещение is Natural range ...;
type РегуляторУгла is tagged private;
function НовРегуляторУгла (номер: Размещение;
    напр: Направление; порт: Порт)
    return РегуляторУгла;
procedure Включить(the: in out РегуляторУгла);
procedure Выключить(the: in out РегуляторУгла);
procedure УвеличитьУгол(№е: in out
    РегуляторУгла);
procedure УменьшитьУгол(the: in out
    РегуляторУгла);
function ОпросСостояния(the: РегуляторУгла)
    return Режим;
private
    type укз_наПорт is access all Порт;
    type РегуляторУгла is tagged record
        Номер: Размещение;
        Состояние: Режим;
        Управление: укз_наПорт;
    end record;
end Класс_РегуляторУгла;

```

Теперь, когда сделаны необходимые приготовления, объявим нужные экземпляры классов, то есть объекты:

```

РабочийГрафик: aliased ГрафикРазворота;
РабочийКонтроллер: aliased Контроллеругла;

```

Далее мы должны определить конкретные параметры графика разворота

```

Связь (РабочийГрафик, 30, 60, 90);

```

а затем предложить объекту-контроллеру выполнить этот график:

```

Обрабатывать (РабочийКонтроллер, РабочийГрафикAccess);

```

Рассмотрим отношение между объектом РабочийГрафик и объектом РабочийКонтроллер. 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, А и В, между которыми имеется связь. Для того чтобы объект А мог послать сообщение в объект В, надо, чтобы В был виден для А.

В примере из предыдущего подраздела объект РабочийКонтроллер должен видеть объект РабочийГрафик (чтобы иметь возможность использовать его как аргумент в операции Обрабатывать).

Различают четыре формы видимости между объектами.

1. Объект-поставщик (сервер) глобален для клиента.
2. 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 обязательно должны 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 (свойствам).

В примере из подраздела «Связи» объект РабочийКонтроллер имеет свойство 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. 9.7 приведен пример физического включения (композиции) частей (Двигателя, Сидений, Колес) в агрегат Автомобиль. В этом случае говорят, что части включены в агрегат по величине.

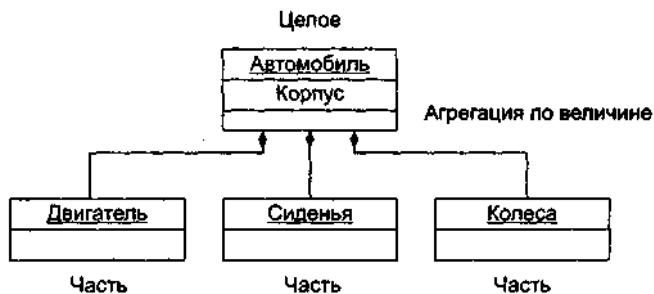


Рис. 9.7. Физическое включение частей в агрегат

На рис. 9.8 приведен пример нефизического включения частей (Студента, Преподавателя) в агрегат Вуз. Очевидно, что Студент и Преподаватель являются элементами Вуза, но они не входят в него физически. В этом случае говорят, что части включены в агрегат по ссылке.

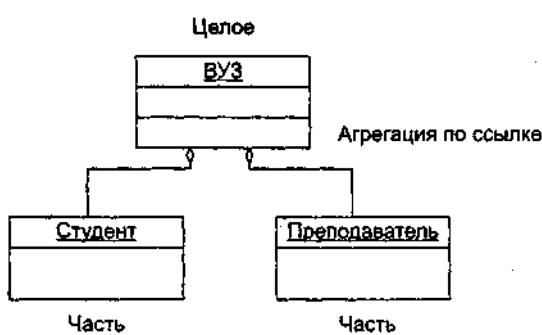


Рис. 9.8. Нефизическое включение частей в агрегат

Итак, между объектами существуют 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.

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 свойства, операции, отношения и семантику (смысл). Любой объект — просто экземпляр класса.

Как показано на рис. 9.9, 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, переменных, констант и исключений, необходимых для полноты данной абстракции.

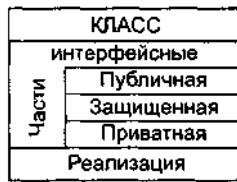


Рис. 9.9. Структуре представления класса

Интерфейс может быть разделен на 3 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.

Другом класса называют класс, который имеет доступ ко всем частям этого класса (публичной, защищенной и приватной). Иными словами, от друга у класса нет секретов.

ПРИМЕЧАНИЕ

Другом класса может быть и свободная подпрограмма.

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 проблемной областью связывают ключевые абстракции, отношения между которыми формируют структуру из классов системы.

Всего существует четыре основных вида отношений между классами:

- ассоциация (фиксирует 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 kleem, 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*, которое фиксирует отношение между клиентом, запрашивающим услугу, и сервером, предоставляющим эту услугу.

Конкретизация выражает другую разновидность отношения *обобщение-специализация*. Применяется в таких языках, как Ada 95, C++, Эйфель.

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: 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.

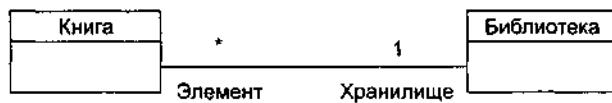


Рис. 9.10. Ассоциация

Отношение ассоциации между классами изображено на рис. 9.10. Очевидно, что ассоциация предполагает двухсторонние 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 приводим 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.

Примеры ассоциаций с различными типами мощности приведены на рис. 9.11, 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.

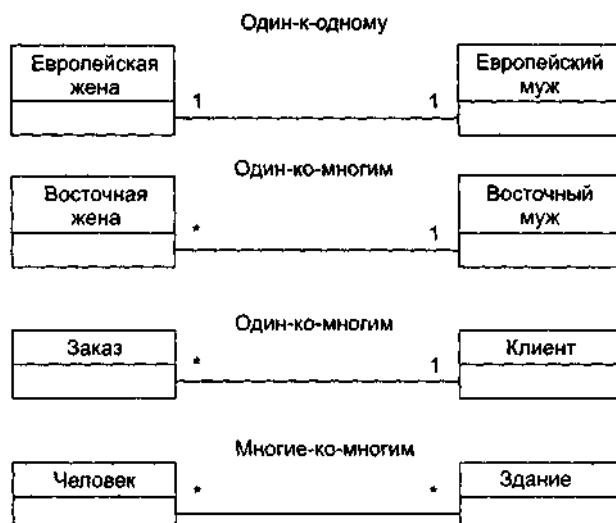


Рис. 9.11. Ассоциации с различными типами мощности

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: 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

```
with ...;...
use ...; ...
Package Класс_ПараметрыПолета is
    type ПараметрыПолета is tagged private;
    function Инициировать return ПараметрыПолета;
    procedure Записывать (the: in out ПараметрыПолета);
    function ТекущВремя (the: ПараметрыПолета)
        return БортовоеВремя;
```

```
private
    type ПараметрыПолета is tagged record
        Имя: integer;
        ОтметкаВремени: БортовоеВремя;
    end record;
```

```
end Класс_ПараметрыПолета;
```

Запись параметров кабины самолета может обеспечиваться следующим классом:

```
with Класс_ПараметрыПолета; ...
use Класс_ПараметрыПолета; ...
Package Класс_Кабина is
```

```
    type Кабина is new ПараметрыПолета with private;
    function Инициировать (Д:Давление; К:Кислород;
                           Т:Температура) return Кабина;
    procedure Записывать (the: in out Кабина);
    function ПерепадДавления (the: Кабина) return Давление;
```

```
private
    type Кабина is new ПараметрыПолета
        with record
            параметр1: Давление;
            параметр2: Кислород;
            параметр3: Температура
        end record;
```

```
end Класс_Кабина;
```

Этот класс наследует структуру и поведение класса ПараметрыПолета, но наращивает его структуру (вводит три новых элемента данных), переопределяет его поведение (процедура Записывать) и дополняет его поведение (функция ПерепадДавления).

Иерархическая структура классов системы для записи параметров полета, находящихся в отношении наследования, показана на рис. 9.12.

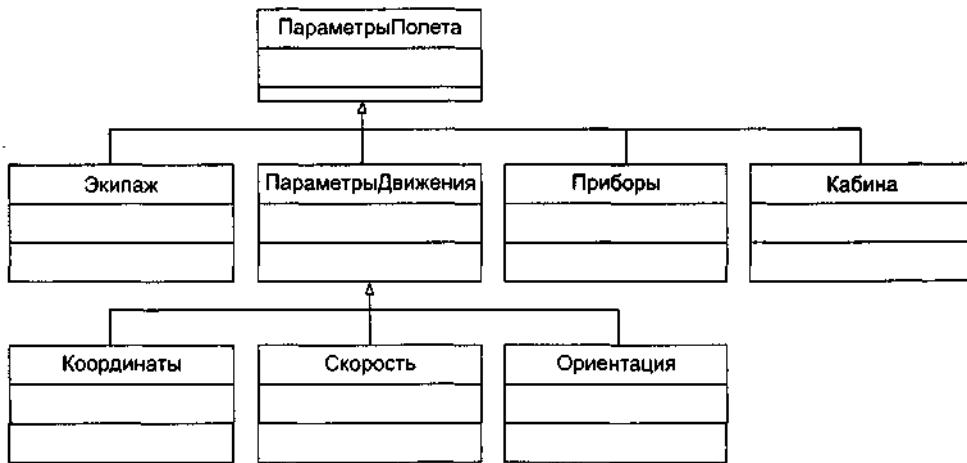


Рис. 9.12. Иерархия простого наследования

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.

Рассмотрим различные реализации процедуры Записывать. Для класса ПараметрыПолета реализация имеет вид

```

procedure Записывать (the: in out ПараметрыПолета) is
begin
    -- записывать имя параметра
    -- записывать отметку времени
end Записывать;
  
```

В классе Кабина предусмотрена другая реализация процедуры:

```

procedure Записывать (the: in out Кабина) is
begin
    Записывать (ПараметрыПолета (the)); -- вызов метода
        -- суперкласса
    -- записывать значение давления
    -- записывать процентное содержание кислорода
    -- записывать значение температуры
end Записывать;
  
```

Предположим, что мы имеем по экземпляру каждого из этих двух классов:

Вполете: ПараметрыПолета:= Инициировать;

Вкабине: Кабина:= Инициировать (768. 21. 20);

Предположим также, что имеется свободная процедура:

```

procedure СохранятьНовДанные (d: in out
    ПараметрыПолета'class; t: БортовоеВремя) is
begin
    if ТекущВремя(d) >= t then
        Записывать (d); -- диспетчирование с помощью тега
    end if;
end СохранятьНовДанные;
  
```

Что случится при выполнении следующих операторов?

- СохранятьНовДанные (Вполете, БортовоеВремя (60));
- СохранятьНовДанные (Вкабине, БортовоеВремя (120));

Каждый из операторов вызывает операцию Записывать нужного класса. В первом случае диспетчирование приведет к операции Записывать из класса ПараметрыПолета. Во втором случае будет выполняться операция из класса Кабина. Как видим, в свободной процедуре переменная d может

обозначать объекты разных классов, значит, здесь записан вызов полиморфной операции.

Агрегация

Отношения агрегации между классами аналогичны отношениям агрегации между объектами.

Повторим пример с описанием класса КонтроллерУгла:

```
with Класс_ГрафикРазворота, Класс_РегуляторУгла;
use Класс_ГрафикРазворота, Класс_РегуляторУгла;
Package Класс_КонтроллерУгла is
    type укз_наГрафик is access all ГрафикРазворота;
    type КонтроллерУгла is tagged private:
        procedure Обрабатывать (the: in out КонтроллерУгла;
                               уг: укз_наГрафик);
        function Запланировано (the: КонтроллерУгла;
                               уг: укз_наГрафик) return Секунда;
    private
        type КонтроллерУгла is tagged record
            регулятор: РегуляторУгла;
        ...
    end Класс_КонтроллерУгла;
```

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. 9.13.

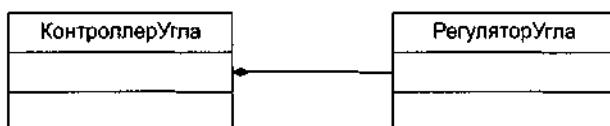


Рис. 9.13. Отношение агрегации по величине (композиция)

Возможен косвенный тип агрегации — включением по ссылке. Если мы запишем в приватной части класса КонтроллерУгла:

```
...
private
    type укз_наРегуляторУгла is access all РегуляторУгла;
    type КонтроллерУгла is tagged record
        регулятор: укз_наРегуляторУгла;
    ...
end Класс_КонтроллерУгла;
```

то регулятор как часть контроллера будет доступен косвенно.

Теперь сцепление объектов уменьшено. Экземпляры каждого класса создаются и уничтожаются независимо.

Еще два примера агрегации по ссылке и по величине (композиции) приведены на рис. 9.14. 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. 9.15, возможны и другие формы представления агрегации по величине — композиции. Композицию можно отобразить графическим вложением символов частей в символ агрегата (левая часть рис. 9.15). 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.

По ссылке

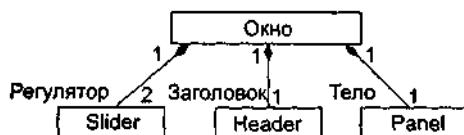
По величине
(композиция)

Рис. 9.14. Агрегация классов

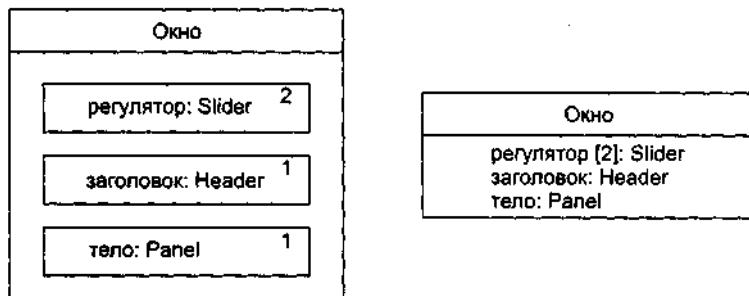


Рис. 9.15. Формы представления композиции

Эта роль соответствует той роли, которую играет часть в неявном (в этой нотации) отношении композиции между частью и целым (агрегатом).

Отметим, что, как представлено в правой части рис. 9.15, в сущности, свойства (атрибуты) класса находятся в отношении композиции между всем классом и его элементами-свойствами. Тем не менее в общем случае свойства 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, свойства 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.

Наиболее часто зависимости показывают, что один класс использует другой класс как аргумент в сигнатуре своей операции. В предыдущем примере (на языке Ada 95) класс ГрафикРазворота появляется как аргумент в методах Обрабатывать и Запланировано класса КонтроллерУгла. Поэтому, как показано на рис. 9.16, КонтроллерУгла зависит от класса ГрафикРазворота.



Рис. 9.16. Отношение зависимости

Конкретизация

Г. Буч определяет конкретизацию как процесс наполнения шаблона (родового или параметризованного класса). Целью является получение класса, от которого возможно создание

экземпляров [22].

Родовой 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 родового 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. Воспользуемся возможностями языка Ada 95, в котором впервые была реализована идея настройки-параметризации. Здесь формальные родовые параметры записываются между словом *generic* и заголовком пакета, размещающего класс.

Пример: представим родовой (параметризованный) класс Очередь:

```
generic
    type Элемент is private;
package Класс_Очередь is
    type Очередь is limited tagged private;
    ...
    procedure Добавить (В_Очередь: in out Очередь;
                         элт: Элемент );
    ...
private
    ...
end Класс_Очередь;
```

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:

```
package Класс_ОчередьЦелыхЭлементов is new Класс_Очередь
    (Элемент => Integer);
package Класс_ОчередьЛилипутов is new Класс_Очередь
    (Элемент => Лилипут);
```

В первом случае мы настраивали класс на конкретный тип Integer (фактический родовой параметр), во втором случае — на конкретный тип Лилипут.

Классы ОчередьЦелыхЭлементов и ОчередьЛилипутов можно использовать как обычные классы. Они содержат все средства родового класса, но только эти средства настроены на использование конкретного типа, заданного при конкретизации.

Графическая иллюстрация отношений конкретизации приведена на рис. 9.17. Отметим, что отношение конкретизации отображается с помощью подписанной стрелки отношения зависимости. Это логично, поскольку конкретизированный класс зависит от родового класса (класса-шаблона).



Рис. 9.17. Отношения конкретизации родового класса

Контрольные вопросы

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.

ГЛАВА 10. 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 Джекобсона (I. Jacobson), язык Коада-Йордона (Coad-Yourdon), язык Шлеера-Меллора (Shlaer-Mellor) и т. д [41], [64], [69]. 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 принят Unified Modeling Language (UML), создававшийся в 1994-1997 годах (основные разработчики — три «amigos» Г. Буч, Дж. Рамбо, И. Джекобсон). В настоящее время разработана версия UML 1.4, которая описывается в данном учебнике [53]. Данная глава посвящена определению базовых понятий языка UML.

Унифицированный язык моделирования

UML — стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем [23], [53], [67]. UML может использоваться для визуализации, спецификации, конструирования 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 95, Object Pascal) и даже в таблицы для реляционной БД.

Словарь UML образуют три разновидности строительных блоков: предметы, отношения, диаграммы. Предметы — это абстракции, которые являются основными элементами в модели, отношения

связывают эти предметы, диаграммы группируют коллекции предметов.

Предметы в UML

В UML имеются четыре разновидности предметов:

- структурные предметы;
- предметы поведения;
- группирующие предметы;
- поясняющие предметы.

Эти предметы являются базовыми объектно-ориентированными строительными блоками. Они используются для написания моделей.

Структурные предметы являются существительными в UML-моделях. Они представляют статические части модели — понятийные или физические элементы. Перечислим восемь разновидностей структурных предметов.

1. *Класс* — описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Класс реализует один или несколько интерфейсов. Как показано на рис. 10.1, графически класс отображается в виде прямоугольника, обычно включающего секции с именем, свойствами (атрибутами) и операциями.

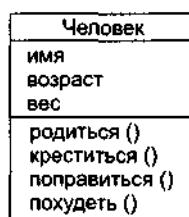


Рис. 10.1. Классы

2. *Интерфейс* — набор операций, которые определяют услуги класса или компонента. Интерфейс описывает поведение элемента, видимое извне. Интерфейс может представлять полные услуги класса или компонента или часть таких услуг. Интерфейс определяет набор спецификаций операций (их сигнатуры), а не набор реализаций операций. Графически интерфейс изображается в виде кружка с именем, как показано на рис. 10.2. Имя интерфейса обычно начинается с буквы «I». Интерфейс редко показывают самостоятельно. Обычно его присоединяют к классу или компоненту, который реализует интерфейс.
3. *Кооперация* (сотрудничество) определяет взаимодействие и является совокупностью ролей и других элементов, которые работают вместе для обеспечения коллективного поведения более сложного, чем простая сумма всех элементов. Таким образом, кооперации имеют как структурное, так и поведенческое измерения. Конкретный класс может участвовать в нескольких кооперациях. Эти кооперации представляют реализацию паттернов (образцов), которые формируют систему. Как показано на рис. 10.3, графически кооперация изображается как пунктирный эллипс, в который вписывается ее имя.



Рис. 10.2. Интерфейсы

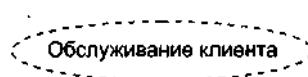


Рис. 10.3. Кооперации

4. *Актер* — набор согласованных ролей, которые могут играть пользователи при взаимодействии с системой (ее элементами Use Case). Каждая роль требует от системы определенного поведения. Как показано на рис. 10.4, актер изображается как проволочный человечек с именем.



Заказчик

Рис. 10.4. Актеры

5. Элемент *Use Case* (Прецедент) — описание последовательности действий (или нескольких последовательностей), выполняемых системой в интересах отдельного актера и производящих видимый для актера результат. В модели элемент *Use Case* применяется для структурирования предметов поведения. Элемент *Use Case* реализуется кооперацией. Как показано на рис. 10.5, элемент *Use Case* изображается как эллипс, в который вписывается его имя.



Рис. 10.5. Элементы Use Case

6. *Активный класс* — класс, чьи объекты имеют один или несколько процессов (или потоков) и поэтому могут инициировать управляющую деятельность. Активный класс похож на обычный класс за исключением того, что его объекты действуют одновременно с объектами других классов. Как показано на рис. 10.6, активный класс изображается как утолщенный прямоугольник, обычно включающий имя, свойства (атрибуты) и операции.

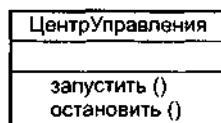


Рис. 10.6. Активные классы

7. *Компонент* — физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов. В систему включаются как компоненты, являющиеся результатами процесса разработки (файлы исходного кода), так и различные разновидности используемых компонентов (COM+-компоненты, Java Beans). Обычно компонент — это физическая упаковка различных логических элементов (классов, интерфейсов и сотрудничеств). Как показано на рис. 10.7, компонент изображается как прямоугольник с вкладками, обычно включающий имя.

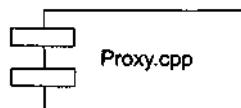


Рис. 10.7. Компоненты

8. *Узел* — физический элемент, который существует в период работы системы и представляет ресурс, обычно имеющий память и возможности обработки. В узле размещается набор компонентов, который может перемещаться от узла к узлу. Как показано на рис. 10.8, узел изображается как куб с именем.



Рис. 10.8. Узлы

Предметы поведения — динамические части UML-моделей. Они являются глаголами моделей, представлением поведения во времени и пространстве. Существует две основные разновидности предметов поведения.

1. *Взаимодействие* — поведение, заключающее в себе набор сообщений, которыми обменивается набор объектов в конкретном контексте для достижения определенной цели. Взаимодействие может определять динамику как совокупности объектов, так и отдельной операции. Элементами взаимодействия являются сообщения, последовательность действий (поведение, вызываемое

сообщением) и связи (соединения между объектами). Как показано на рис. 10.9, сообщение изображается в виде направленной линии с именем ее операции.



Рис. 10.9. Сообщения

2. *Конечный автомат* — поведение, которое определяет последовательность состояний объекта или взаимодействия, выполняемые в ходе его существования в ответ на события (и с учетом обязанностей по этим событиям). С помощью конечного автомата может определяться поведение индивидуального класса или кооперации классов. Элементами конечного автомата являются состояния, переходы (от состояния к состоянию), события (предметы, вызывающие переходы) и действия (реакции на переход). Как показано на рис. 10.10, состояние изображается как закругленный прямоугольник, обычно включающий его имя и его подсостояния (если они есть).

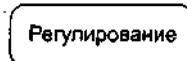


Рис. 10.10. Состояния

Эти два элемента — взаимодействия и конечные автоматы — являются базисными предметами поведения, которые могут включаться в UML-модели. Семантически эти элементы ассоциируются с различными структурными элементами (прежде всего с классами, сотрудничествами и объектами).

Группирующие предметы — организационные части UML-моделей. Это ящики, по которым может быть разложена модель. Предусмотрена одна разновидность группирующего предмета — пакет.

Пакет — общий механизм для распределения элементов по группам. В пакет могут помещаться структурные предметы, предметы поведения и даже другие группировки предметов. В отличие от компонента (который существует в период выполнения), пакет — чисто концептуальное понятие. Это означает, что пакет существует только в период разработки. Как показано на рис. 10.11, пакет изображается как папка с закладкой, на которой обозначено его имя и, иногда, его содержание.



Рис. 10.11. Пакеты

Поясняющие предметы — разъясняющие части UML-моделей. Они являются замечаниями, которые можно применить для описания, объяснения и комментирования любого элемента модели. Предусмотрена одна разновидность поясняющего предмета — примечание.

Примечание — символ для отображения ограничений и замечаний, присоединяемых к элементу или совокупности элементов. Как показано на рис. 10.12, примечание изображается в виде прямоугольника с загнутым углом, в который вписывается текстовый или графический комментарий.

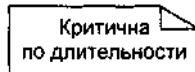


Рис. 10.12. Примечания

Отношения в UML

В UML имеются четыре разновидности отношений:

- 1) зависимость;
- 2) ассоциация;
- 3) обобщение;
- 4) реализация.

Эти отношения являются базовыми строительными блоками отношений. Они используются при написании моделей.

1. *Зависимость* — семантическое отношение между двумя предметами, в котором изменение в одном предмете (независимом предмете) может влиять на семантику другого предмета (зависимого предмета). Как показано на рис. 10.13, зависимость изображается в виде пунктирной линии, возможно направленной на независимый предмет и иногда имеющей метку.

Рис. 10.13. Зависимости

2. *Ассоциация* — структурное отношение, которое описывает набор связей, являющихся соединением между объектами. Агрегация — это специальная разновидность ассоциации, представляющая структурное отношение между целым и его частями. Как показано на рис. 10.14, ассоциация изображается в виде сплошной линии, возможно направленной, иногда имеющей метку и часто включающей другие «украшения», такие как мощность и имена ролей.

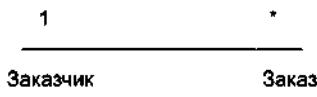


Рис. 10.14. Ассоциации

3. *Обобщение* — отношение специализации/обобщения, в котором объекты специализированного элемента (потомка, ребенка) могут заменять объекты обобщенного элемента (предка, родителя). Иначе говоря, потомок разделяет структуру и поведение родителя. Как показано на рис. 10.15, обобщение изображается в виде сплошной стрелки с полым наконечником, указывающим на родителя.



Рис. 10.15. Обобщения

4. *Реализация* — семантическое отношение между классификаторами, где один классификатор определяет контракт, который другой классификатор обязуется выполнять (к классификаторам относят классы, интерфейсы, компоненты, элементы Use Case, кооперации). Отношения реализации применяют в двух случаях: между интерфейсами и классами (или компонентами), реализующими их; между элементами Use Case и кооперациями, которые реализуют их. Как показано на рис. 10.16, реализация изображается как нечто среднее между обобщением и зависимостью.



Рис. 10.16. Реализации

Диаграммы в UML

Диаграмма — графическое представление множества элементов, наиболее часто изображается как связный граф из вершин (предметов) и дуг (отношений). Диаграммы рисуются для визуализации системы с разных точек зрения, затем они отображаются в систему. Обычно диаграмма дает неполное представление элементов, которые составляют систему. Хотя один и тот же элемент может появляться во всех диаграммах, на практике он появляется только в некоторых диаграммах. Теоретически диаграмма может содержать любую комбинацию предметов и отношений, на практике ограничиваются малым количеством комбинаций, которые соответствуют пяти представлениям архитектуры ПС. По этой причине UML включает девять видов диаграмм:

- 1) диаграммы классов;
- 2) диаграммы объектов;
- 3) диаграммы Use Case (диаграммы прецедентов);
- 4) диаграммы последовательности;
- 5) диаграммы сотрудничества (кооперации);
- 6) диаграммы схем состояний;
- 7) диаграммы деятельности;
- 8) компонентные диаграммы;
- 9) диаграммы размещения (развертывания).

Диаграмма классов показывает набор классов, интерфейсов, сотрудничеств и их отношений. При моделировании объектно-ориентированных систем диаграммы классов используются наиболее часто. Диаграммы классов обеспечивают статическое проектное представление системы. Диаграммы классов, включающие активные классы, обеспечивают статическое представление процессов системы.

Диаграмма объектов показывает набор объектов и их отношения. Диаграмма объектов представляет статический «моментальный снимок» с экземпляров предметов, которые находятся в диаграммах классов. Как и диаграммы классов, эти диаграммы обеспечивают статическое проектное представление или статическое представление процессов системы (но с точки зрения реальных или фототипичных случаев).

Диаграмма Use Case (диаграмма прецедентов) показывает набор элементов Use Case, актеров и их отношений. С помощью диаграмм Use Case для системы создается статическое представление Use Case. Эти диаграммы особенно важны при организации и моделировании поведения системы, задании требований заказчика к системе.

Диаграммы последовательности и диаграммы сотрудничества — это разновидности диаграмм взаимодействия.

Диаграмма взаимодействия показывает взаимодействие, включающее набор объектов и их отношений, а также пересылаемые между объектами сообщения. Диаграммы взаимодействия обеспечивают динамическое представление системы.

Диаграмма последовательности — это диаграмма взаимодействия, которая выделяет упорядочение сообщений по времени.

Диаграмма сотрудничества (диаграмма кооперации) — это диаграмма взаимодействия, которая выделяет структурную организацию объектов, посылающих и принимающих сообщения. Диаграммы последовательности и диаграммы сотрудничества изоморфны, что означает, что одну диаграмму можно трансформировать в другую диаграмму.

Диаграмма схем состояний показывает конечный автомат, представляет состояния, переходы, события и действия. Диаграммы схем состояний обеспечивают динамическое представление системы. Они особенно важны при моделировании поведения интерфейса, класса или сотрудничества. Эти диаграммы выделяют такое поведение объекта, которое управляет событиями, что особенно полезно при моделировании реактивных систем.

Диаграмма деятельности — специальная разновидность диаграммы схем состояний, которая показывает поток от действия к действию внутри системы. Диаграммы деятельности обеспечивают динамическое представление системы. Они особенно важны при моделировании функциональности системы и выделяют поток управления между объектами.

Компонентная диаграмма показывает организацию набора компонентов и зависимости между компонентами. Компонентные диаграммы обеспечивают статическое представление реализации системы. Они связаны с диаграммами классов в том смысле, что в компонент обычно отображается один или несколько классов, интерфейсов или коопераций.

Диаграмма размещения (диаграмма развертывания) показывает конфигурацию обрабатывающих узлов периода выполнения, а также компоненты, живущие в них. Диаграммы размещения обеспечивают статическое представление размещения системы. Они связаны с компонентными диаграммами в том смысле, что узел обычно включает один или несколько компонентов.

Механизмы расширения в UML

UML — развитый язык, имеющий 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. 10.17 введено простое ограничение на свойство *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.

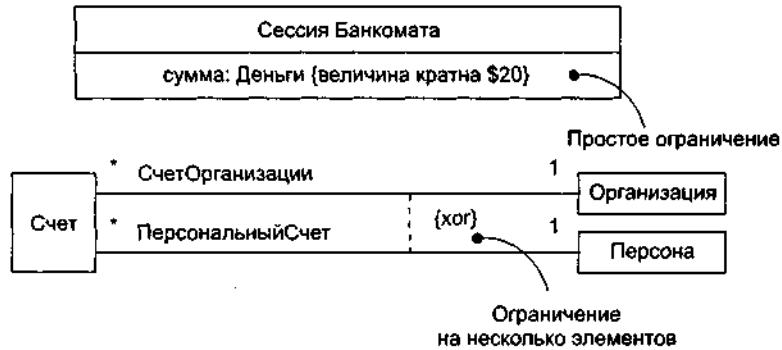


Рис. 10.17. Ограничения

Теговая величина (tagged value) расширяет характеристики строительного UML-блока, позволяя создать новую информацию в спецификации конкретного элемента. Теговую величину показывают как строку в фигурных скобках {}. Стока 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. 10.18 класс ТекстовыйПроцессор расширен путем явного указания его версии и автора.

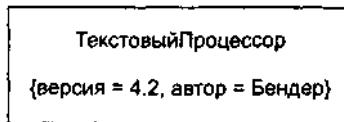


Рис. 10.18. Расширение класса

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, указанное в двойных угловых скобках (или в угловых кавычках).

Примеры элементов со стереотипами приведены на рис. 10.19. Стереотип «exception» говорит о том, что класс ПотеряЗначимости теперь рассматривается как специальный класс, которому, положим, разрешается только генерация и обработка сигналов исключений. Особые возможности метакласса получил класс ЭлементМодели. Кроме того, здесь показано применение стереотипа «call» к отношению зависимости (у него появился новый смысл).

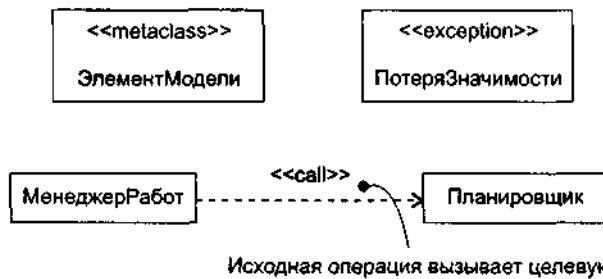


Рис. 10.19. Стереотипы

Таким образом, механизмы расширения позволяют адаптировать UML под нужды конкретных проектов и под новые программные технологии. Возможно добавление новых строительных блоков, модификация спецификаций существующих блоков и даже изменение их семантики. Конечно, очень важно обеспечить контролируемое введение расширений.

Контрольные вопросы

1. Сколько поколений языков визуального моделирования вы знаете?
2. Назовите численность языков визуального моделирования 2-го поколения.
3. Какая необходимость привела к созданию языка визуального моделирования третьего поколения?
4. Поясните назначение UML.
5. Какие строительные блоки образуют словарь UML? Охарактеризуйте их.
6. Какие разновидности предметов UML вы знаете? Их назначение?
7. Перечислите известные вам разновидности структурных предметов UML.
8. Перечислите известные вам разновидности предметов поведения UML.
9. Перечислите известные вам группирующие предметы UML.
10. Перечислите известные вам поясняющие предметы UML.
11. Какие разновидности отношений предусмотрены в UML? Охарактеризуйте каждое отношение.
12. Дайте характеристику диаграммы классов.
13. Дайте характеристику диаграммы объектов.
14. Охарактеризуйте диаграмму Use Case.
15. Охарактеризуйте диаграммы взаимодействия.
16. Дайте характеристику диаграммы последовательности.
17. Дайте характеристику диаграммы сотрудничества.
18. Охарактеризуйте диаграмму схем состояний.
19. Охарактеризуйте диаграмму деятельности.
20. Дайте характеристику компонентной диаграммы.
21. Охарактеризуйте диаграмму размещения.
22. Для чего служат механизмы расширения в UML?
23. Поясните механизм ограничений в UML.
24. Объясните механизм теговых величин в UML.
25. В чем суть механизма стереотипов UML?

ГЛАВА 11. СТАТИЧЕСКИЕ МОДЕЛИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММНЫХ СИСТЕМ

Статические модели обеспечивают представление структуры систем в терминах базовых строительных блоков и отношений между ними. «Статичность» этих моделей состоит в том, что здесь не показывается динамика изменений системы во времени. Вместе с тем следует понимать, что эти модели несут в себе не только структурные описания, но и описания операций, реализующих заданное поведение системы. Основным средством для представления статических моделей являются диаграммы классов [8], [23], [53], [67]. 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 формируют системную архитектуру.

Вершины в диаграммах классов

Итак, вершина в диаграмме классов — класс. Обозначение класса показано на рис. 11.1.



Рис. 11.1. Обозначение класса

Имя класса указывается всегда, свойства и операции — выборочно. Предусмотрено задание области действия свойства (операции). Если свойство (операция) подчеркивается, его областью действия является класс, в противном случае областью Действия является экземпляр (рис. 11.2).

Что это значит? Если областью действия свойства является класс, то все его экземпляры (объекты)

используют общее значение этого свойства, в противном случае у каждого экземпляра свое значение свойства.

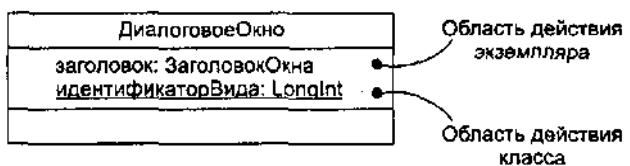


Рис. 11.2. Свойства уровней класса и экземпляра

Свойства

Общий синтаксис представления свойства имеет вид

Видимость Имя [Множественность]: Тип = НачальнЗначение {Характеристики}

Рассмотрим видимость и характеристики свойств.

В языке UML определены три уровня видимости:

public	Любой клиент класса может использовать свойство (операцию), обозначается символом +
protected	Любой наследник класса может использовать свойство (операцию), обозначается символом #
private	Свойство (операция) может использоваться только самим классом, обозначается символом -

ПРИМЕЧАНИЕ

Если видимость не указана, считают, что свойство объявлено с публичной видимостью.

Определены три характеристики свойств:

changeable	Нет ограничений на модификацию значения свойства
addOnly	Для свойств с множественностью, большей единицы; дополнительные значения могут быть добавлены, но после создания значение не может удаляться или
frozen	изменяться После инициализации объекта значение свойства не изменяется

ПРИМЕЧАНИЕ

Если характеристика не указана, считают, что свойство объявлено с характеристикой changeable.

Примеры объявления свойств:

начало	Только имя
+ начало	Видимость и имя
начало : Координаты	Имя и тип
имяфамилия [0..1] : String	Имя, множественность, тип
левыйУгол : Координаты=(0, 10)	Имя, тип, начальное значение
сумма : Integer {frozen}	Имя и характеристика

Операции

Общий синтаксис представления операции имеет вид

Видимость Имя (Список Параметров): ВозвращаемыйТип {Характеристики}

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 характеристика

В сигнатуре операции можно указать ноль 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

Допустимо применение следующих характеристик 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.
guarded	Другими словами, допустим только один поток вызовов (поток управления) 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 свойств и операций

Известно, что пиктограмма класса включает три секции (для имени, для свойств и для операций). Пустота секции не означает, что у класса отсутствуют свойства или операции, просто в данный момент они не показываются. Можно явно определить наличие у класса большего количества свойств или атрибутов. Для этого в конце показанного списка проставляются три точки. Как показано на рис. 11.3, в длинных списках свойств и операций разрешается группировка — каждая группа начинается со своего стереотипа.

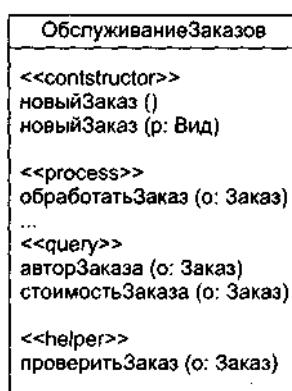


Рис. 11.3. Стереотипы для характеристик класса

Множественность

Иногда бывает необходимо ограничить количество экземпляров класса:

- задать ноль экземпляров (в этом случае класс превращается в утилиту, которая предлагает свои свойства 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. 11.4,

КонтроллерУглов — это класс-singleton, а для класса ДатчикУгла разрешены три экземпляра.

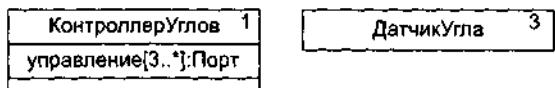


Рис. 11.4. Множественность

Множественность применима не только к классам, но и к свойствам. Множественность свойства задается выражением в квадратных скобках, записанным после его имени. Например, на рисунке заданы три и более экземпляра свойства Управление (в экземпляре класса КонтроллерУглов).

Отношения в диаграммах классов

Отношения, используемые в диаграммах классов, показаны на рис. 11.5.



Рис. 11.5. 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. 11.6, имени можно придать направление — достаточно добавить треугольник направления, который указывает направление, заданное для чтения имени.



Рис. 11.6. Имена ассоциаций

Когда класс участвует в ассоциации, он играет в этом отношении определенную роль. Как показано на рис. 11.7, роль определяет, каким представляется класс на одном конце ассоциации для класса на противоположном конце ассоциации.

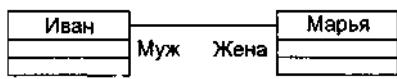


Рис. 11.7. Роли

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 ложностью роли в ассоциации, записывается в виде выражения, задающего диапазон величин или одну величину (рис. 11.8).

Запись мощности на одном конце ассоциации определяет количество объектов, соединяемых с каждым объектом на противоположном конце 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.

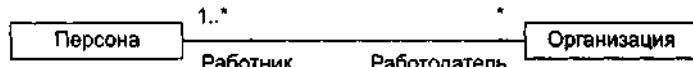


Рис. 11.8. Мощность

Достаточно часто возникает следующая проблема — как для объекта на одном конце ассоциации выделить набор объектов на противоположном конце? Например, рассмотрим взаимодействие между банком и клиентом — вкладчиком. Как показано на рис. 11.9, 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 НомерСчета 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 прямоугольнике вписывается свойство — атрибут ассоциации.

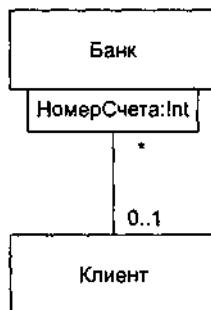


Рис. 11.9. Квалификация

Кроме того, роли в ассоциациях могут иметь пометки *видимости*. Например, на рис. 11.10 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 недоступна извне. Как показано на рисунке, из объекта Начальника можно перемещаться к экземплярам Женщины (и наоборот), но нельзя видеть экземпляры Загадки для объектов Женщины.



Рис. 11.10. Видимость

На конце ассоциации можно задать три уровня видимости, добавляя символ видимости к имени роли:

- по умолчанию для роли задается публичная видимость;
- приватная видимость указывает, что объекты на данном конце недоступны любым объектам вне ассоциации;
- защищенная видимость (protected) указывает, что объекты на данном конце недоступны любым объектам вне ассоциации, за исключением потомков того класса, который указан на противоположном конце ассоциации.

В языке UML ассоциации могут иметь свойства. Как показано на рис. 11.11, такие возможности

отображаются с помощью классов-ассоциаций. Эти классы присоединяются к линии ассоциации пунктирной линией и рассматриваются как классы со свойствами ассоциаций или как ассоциации со свойствами классов.

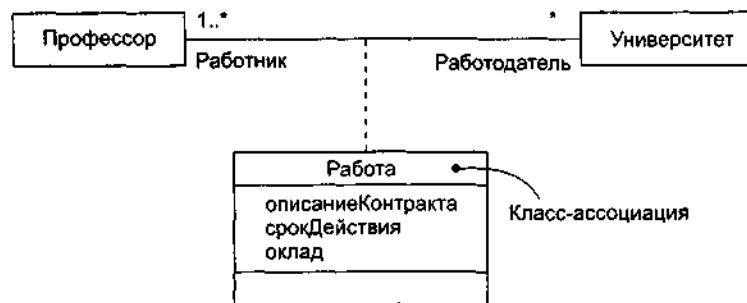


Рис. 11.11. Класс-ассоциация

Свойства класса-ассоциации характеризуют не один, а пару объектов, в данном случае — пару экземпляров, Профессор и Университет.

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.

Например, на рис. 11.12 показана зависимость класса Заказ от класса Книга, так как Книга используется в операциях проверкаДоступности, добавить и удалить класса Заказ.

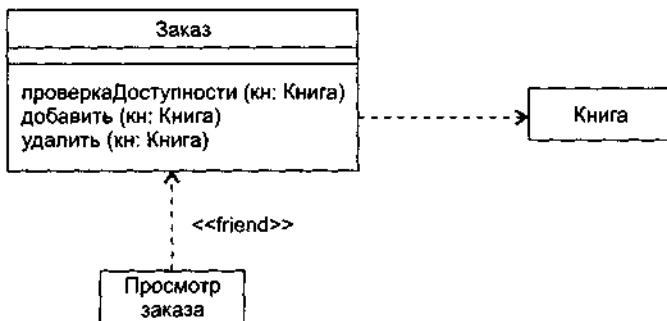


Рис. 11.12. Отношения зависимости

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 17 разновидностей зависимостей, 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. 11.13, подкласс Летающий шкаф является наследником суперклассов Летающий предмет и Хранилище вещей. Этому подклассу достаются в наследство все свойства 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 свойства от его родителя, хотя обычно не все свойства применимы к потомку. Очевидно, что Яблочный_Пирог является Пирогом, но не является Яблоком, так как пироги не растут на деревьях.



Рис. 11.13. Множественное наследование

Еще более сложные проблемы возникают при наследовании от двух классов, имеющих общего родителя. Говорят, что в результате образуется ромбовидная решетка наследования (рис. 11.14).

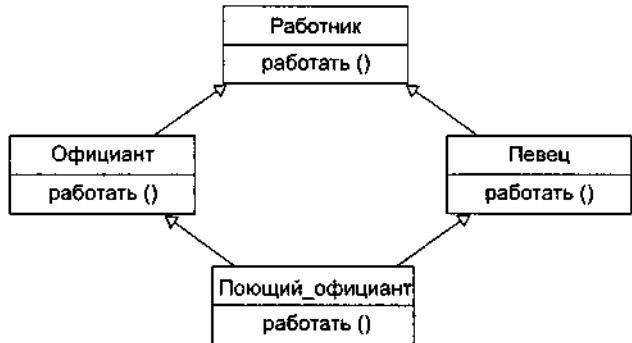


Рис. 11.14. Ромбовидная решетка наследования

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 со свойствами, доставшимися в наследство от родителей и общего прародителя? Хотим ли мы иметь несколько копий свойства 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. 11.15 показано, что класс Каталог должен реализовать интерфейс Обработчик каталога, то есть Обработчик каталога рассматривается как источник, а Каталог — как приемник.

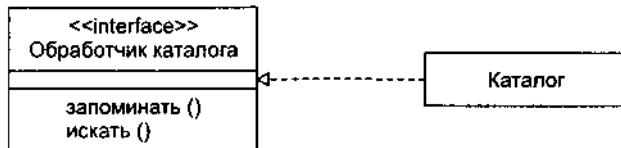


Рис. 11.15. Реализация интерфейса

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.).

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. 11.16 показаны абстрактные классы *Млекопитающие*, *Собаки*, *Кошки*.

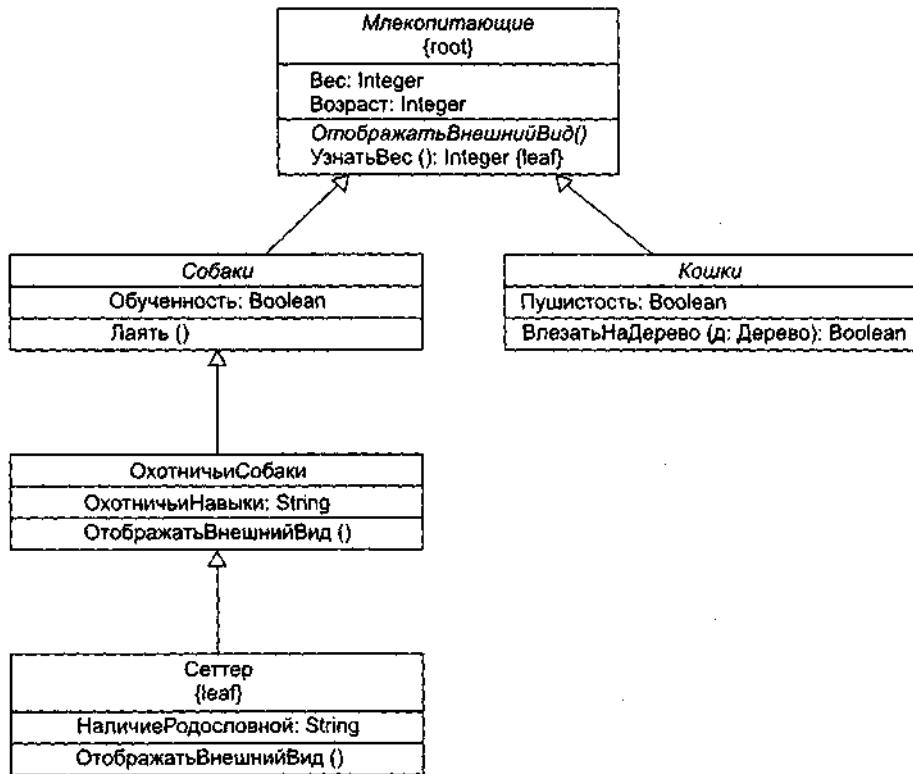


Рис. 11.16. Абстрактность и полиморфизм

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 свойства 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. 11.17 показана диаграмма классов системы управления полетом летательного аппарата.

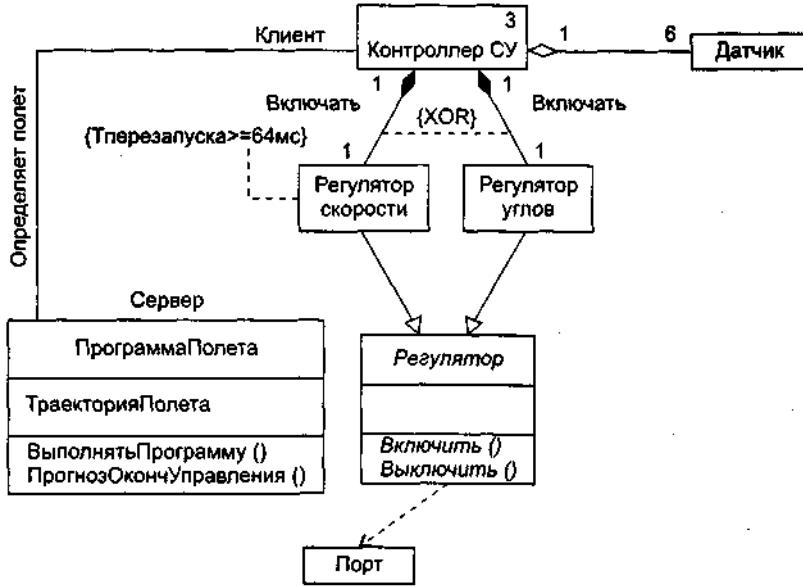


Рис. 11.17. Диаграмма классов системы управления полетом

Здесь представлен класс ПрограммаПолета, который имеет свойство 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. 11.18 приведена диаграмма классов для информационной системы театра. Эту систему образует 6 классов.

Классы-агрегаты Театр и Труппа имеют операции добавления и удаления своих частей, которые включаются в агрегаты по ссылке. Частьми 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 существуют 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 в зрителях, и в актерах есть человеческое начало.

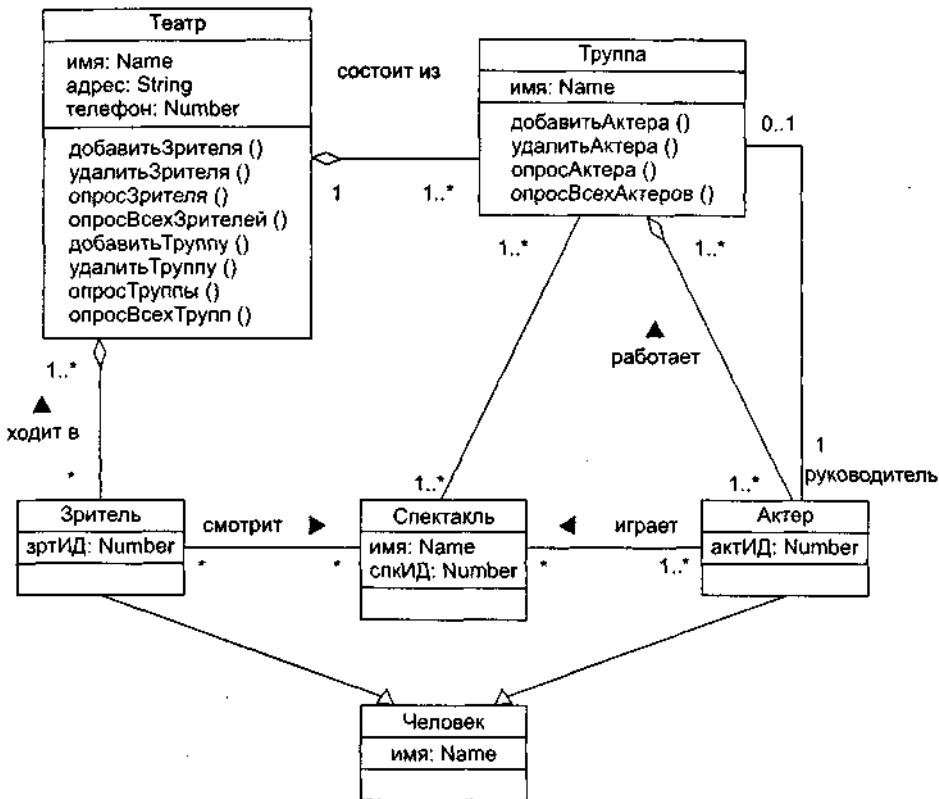


Рис. 11.18. Диаграмма классов информационной системы театра

Контрольные вопросы

- Поясните назначение статических моделей объектно-ориентированных программных систем.
- Что является основным средством для представления статических моделей?
- Как используются статические модели?
- Какие секции входят в графическое обозначение класса?
- Какие секции класса можно не показывать?
- Какие имеются разновидности области действия свойства (операции)?
- Поясните общий синтаксис представления свойства.
- Какие уровни видимости вы знаете? Их смысл?
- Какие характеристики свойств вам известны?
- Поясните общий синтаксис представления операции.
- Какой вид имеет форма представления параметра операции?
- Какие характеристики операций вам известны?
- Что означают три точки в списке свойств (операций)?
- Как организуется группировка свойств (операций)?
- Как ограничить количество экземпляров класса?
- Перечислите известные вам «украшения» отношения ассоциации.
- Может ли статическая модель программной системы не иметь отношений ассоциации?
- Какой смысл имеет квалификатор? К чему он относится?
- Какие отношения могут иметь пометки видимости и что эти пометки обозначают?
- Какой смысл имеет класс-ассоциация?
- Чем отличается агрегация от композиции? Разновидностями какого отношения (в UML) они являются?
- Что обозначает в UML простая зависимость?
- Какой смысл имеет отношение обобщения?
- Какие недостатки у множественного наследования?
- Перечислите недостатки ромбовидной решетки наследования.
- В чем смысл отношения реализации?
- Что обозначает мощность «многие-ко-многим» и в каких отношениях она применяется?
- Что такое абстрактный класс (операция) и как он (она) отображается?

29. Как запретить полиморфизм операции?
30. Как обозначить корневой класс?

ГЛАВА 12. ДИНАМИЧЕСКИЕ МОДЕЛИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММНЫХ СИСТЕМ

Динамические модели обеспечивают представление поведения систем. «Динамизм» этих моделей состоит в том, что в них отражается изменение состояний в процессе работы системы (в зависимости от времени). Средства языка UML для создания динамических моделей многочисленны и разнообразны [8], [23], [41], [53], [67]. Эти средства ориентированы не только на собственно программные системы, но и на отображение требований заказчика к поведению таких систем.

Моделирование поведения программной системы

Для моделирования поведения системы используют:

- автоматы;
- взаимодействия.

Автомат (State machine) описывает поведение в терминах последовательности состояний, через которые проходит объект в течение своей жизни. Взаимодействие (Interaction) описывает поведение в терминах обмена сообщениями между объектами.

Таким образом, автомат задает поведение системы как цельной, единой сущности; моделирует жизненный цикл единого объекта. В силу этого автоматный подход удобно применять для формализации динамики отдельного трудного для понимания блока системы.

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 фиксации полной динамики системы.

Автоматы отображают с помощью:

- диаграмм схем состояний;
- диаграмм деятельности.

Взаимодействия отображают с помощью:

- диаграмм сотрудничества (кооперации);
- диаграмм последовательности.

Диаграммы схем состояний

Диаграмма схем состояний — одна из пяти диаграмм UML, моделирующих динамику систем. Диаграмма схем состояний отображает конечный автомат, выделяя поток управления, следующий от состояния к состоянию. Конечный автомат — поведение, которое определяет последовательность состояний в ходе существования объекта. Эта последовательность рассматривается как ответ на события и включает реакции на эти события.

Диаграмма схем состояний 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 показано на рис. 12.1, состояние изображается как закругленный прямоугольник, обычно включающий его имя и подсостояния (если они есть).

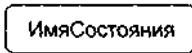


Рис. 12.1. Обозначение состояния

Переходы между состояниями отображаются помеченными стрелками (рис. 12.2).

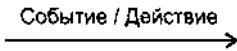


Рис. 12.2. Переходы между состояниями

На рис. 12.2 обозначено: Событие — происшествие, вызывающее изменение состояния, Действие — набор операций, запускаемых событием.

Иначе говоря, события вызывают переходы, а действия являются реакциями на переходы.

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); Ш.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. 12.3.



Рис. 12.3. Переход в начальное состояние

Соответственно, обозначение перехода в конечное состояние имеет вид, представленный на рис. 12.4.

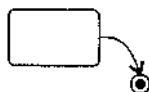


Рис. 12.4. Переход в конечное состояние

В качестве примера на рис. 12.5 показана диаграмма схем состояний для системы охранной сигнализации.

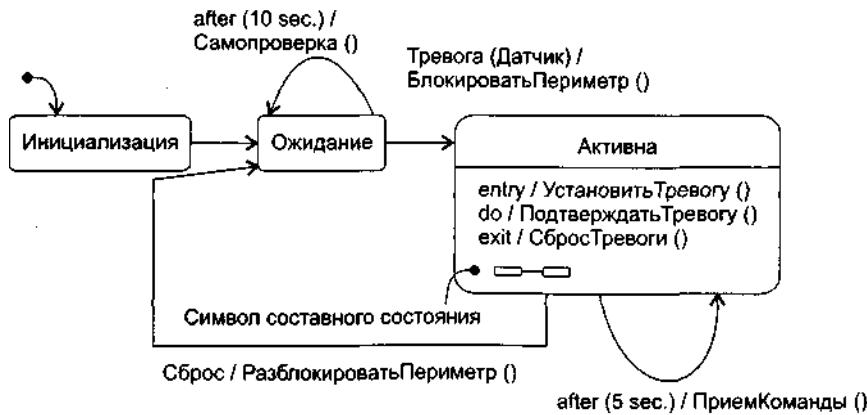


Рис. 12.5. Диаграмма схем состояний системы охранной сигнализации

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.

Например, как показано на рис. 12.6, при входе в состояние Активна выполняется операция УстановитьТревогу() из класса Контроллер, а при выходе из состояния — операция СбросТревоги().

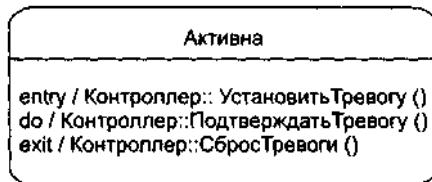


Рис. 12.6. Входные и выходные действия и деятельность в состоянии Активна

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().

Условные переходы

Между состояниями возможны различные типы переходов. Обычно переход инициируется событием. Допускаются переходы и без событий. Наконец, разрешены условные или охраняемые переходы.

Правила пометки стрелок условных переходов иллюстрирует рис. 12.7.

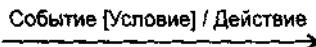


Рис. 12.7. Обозначение условного перехода

Порядок выполнения условного перехода:

- 1) происходит событие;
- 2) вычисляется 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. 12.8. Он происходит по событию ПитаниеПодано, но только в том случае, если достигнут боевой режим лазера.

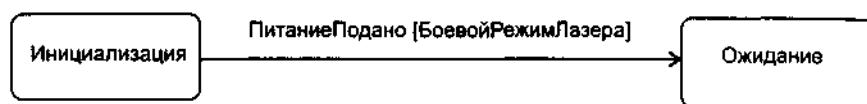


Рис. 12.8. Условный переход между состояниями

Вложенные 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. 12.9 показано составное состояние, содержащее в себе два подсостояния.



Рис. 12.9. Обозначение подсостояний

На рис. 12.10 приведена внутренняя структура составного состояния Активна.

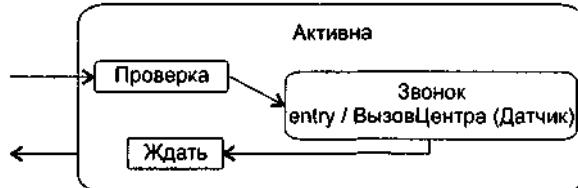


Рис. 12.10. Переходы в состоянии Активна

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 последовательных подсостояний.

Возможно наличие параллельных подсостояний — они выполняются параллельно внутри составного состояния. Графически изображения параллельных подсостояний отделяются друг от друга пунктирными линиями.

Иногда при возврате в составное 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. 12.11, подобная семантика переходов отображается значком истории — буквой Н внутри кружка.

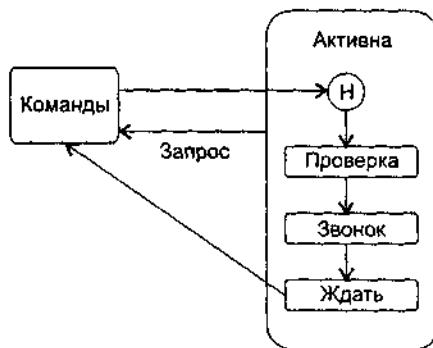


Рис. 12.11. Историческое состояние

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).

Как показано на рис. 12.12, для обозначения составного состояния, имеющего внутри себя скрытые (не показанные на диаграмме) подсостояния, используется символ «очки».

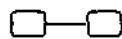


Рис. 12.12. Символ состояния со скрытыми подсостояниями

Диаграммы деятельности

Диаграмма деятельности представляет особую форму конечного автомата, в которой показываются процесс вычислений и потоки работ. В ней выделяются не обычные состояния объекта, а состояния

выполняемых вычислений — состояния действий. При этом полагается, что процесс вычислений не прерывается внешними событиями. Словом, диаграммы деятельности очень похожи на блок-схемы алгоритмов.

Основной вершиной в диаграмме деятельности является состояние действия (рис. 12.13), которое изображается как прямоугольник с закругленными боковыми сторонами.

[Создать Каталог](#)

Рис. 12.13. Состояние действия

Состояние действия 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 состояние под-деятельности. Изображение состояния под-деятельности содержит пиктограмму в правом нижнем углу (рис. 12.14).

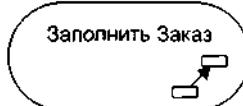


Рис. 12.14. Состояние под-деятельности

Фактически в данную вершину вписывается имя другой диаграммы, имеющей внутреннюю структуру.

Переходы между вершинами — состояниями действий — изображаются в виде стрелок. Переходы выполняются по окончании действий.

Кроме того, в диаграммах деятельности используются вспомогательные вершины:

- решение (ромбик с одной входящей и несколькими исходящими стрелками);
- объединение (ромбик с несколькими входящими и одной исходящей стрелкой);
- линейка синхронизации — разделение (жирная горизонтальная линия с одной входящей и несколькими исходящими стрелками);
- линейка синхронизации — слияние (жирная горизонтальная линия с несколькими входящими и одной исходящей стрелкой);
- начальное состояние (черный кружок);
- конечное состояние (незакрашенный кружок, в котором размещен черный кружок меньшего размера).

Вершина «решение» позволяет отобразить разветвление вычислительного процесса, исходящие из него стрелки помечаются сторожевыми условиями ветвления.

Вершина «объединение» отмечает точку слияния альтернативных потоков действий.

Линейки синхронизации позволяют показать параллельные потоки действий, отмечая точки их синхронизации при запуске (момент разделения) и при завершении (момент слияния).

Пример диаграммы деятельности приведен на рис. 12.15. Эта диаграмма описывает деятельность покупателя в Интернет-магазине. Здесь представлены две точки ветвления — для выбора способа поиска товара и для принятия решения о покупке. Присутствуют три линейки синхронизации: верхняя отражает разделение на два параллельных процесса, средняя отражает и разделение, и слияние процессов, а нижняя — только слияние процессов.

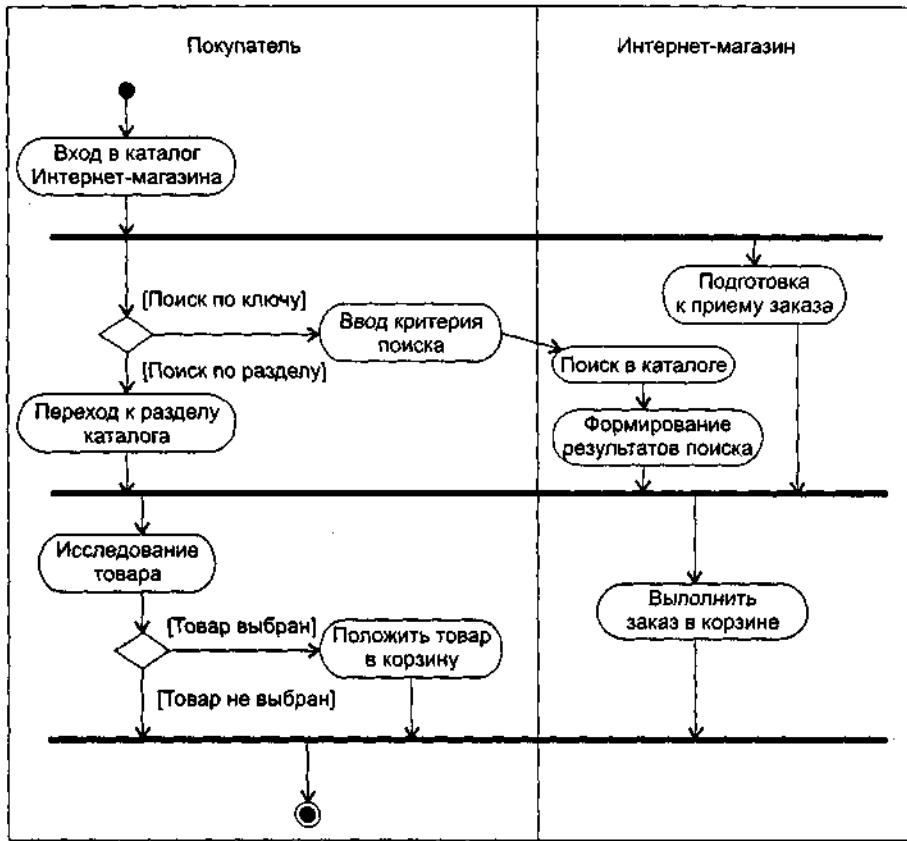


Рис. 12.15. Диаграмма деятельности покупателя в Интернет-магазине

Дополнительно на этой диаграмме показаны две плавательные дорожки — дорожка покупателя и дорожка магазина, которые разделены вертикальной линией. Каждая дорожка имеет имя и фиксирует область деятельности конкретного лица, обозначая зону его ответственности.

Диаграммы взаимодействия

Диаграммы взаимодействия предназначены для моделирования динамических аспектов системы. Диаграмма взаимодействия показывает взаимодействие, включающее набор объектов и их отношений, а также пересылаемые между объектами сообщения. Существуют две разновидности диаграммы взаимодействия — диаграмма последовательности и диаграмма сотрудничества. Диаграмма последовательности — это диаграмма взаимодействия, которая выделяет упорядочение сообщений по времени. Диаграмма сотрудничества — это диаграмма взаимодействия, которая выделяет структурную организацию объектов, посылающих и принимающих сообщения. Элементами диаграмм взаимодействия являются участники взаимодействия — объекты, связи, сообщения.

Диаграммы сотрудничества

Диаграммы сотрудничества отображают взаимодействие объектов в процессе функционирования системы. Такие диаграммы моделируют сценарии поведения системы. В русской литературе диаграммы сотрудничества часто называют диаграммами кооперации.

Обозначение объекта показано на рис. 12.16.



Рис. 12.16. Обозначение объекта

Имя объекта подчеркивается и указывается всегда, свойства указываются выборочно. Синтаксис представления имени имеет вид

ИмяОбъекта : ИмяКласса

Примеры записи имени:

<u>Адам</u> : Человек	Имя объекта и класса
<u>Пользователь</u>	Только имя класса (анонимный объект)
<u>мойКомпьютер</u>	Только имя объекта (подразумевается, что имя класса известно)
<u>агент</u> :	Объект — сирота (подразумевается, что имя класса неизвестно)

Синтаксис представления свойства имеет вид

Имя : Тип = Значение

Примеры записи свойства:

номер:Телефон = "7350-420"	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. Путь может быть снабжен характеристикой видимости. Характеристика видимости проставляется как стандартный стереотип над дальшим концом связи. В языке предусмотрены следующие стандартные стереотипы видимости:

«global»	Объект-поставщик находится в глобальной области определения
«local»	Объект-поставщик находится в локальной области определения объекта-клиента
«parameter»	Объект-поставщик является параметром операции объекта-клиента
«self»	Один и тот же объект является и клиентом, и поставщиком

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	В объекте запускается операция
Возврат	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:

ВозврВеличина := ИмяСообщения (Аргументы),

где ВозврВеличина задает величину, возвращаемую как результат обработки сообщения.

Примеры записи сообщений:

Координаты := Текущее Положение(самолетT1)	Вызов операции, возврат значения
оповещение()	Посылка сигнала
УстановитьМаршрут(x)	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 или вложенный поток (поток синхронных сообщений). Как показано на рис. 12.17, процедурный поток рисуется стрелками с заполненными

наконечниками.



Рис. 12.17. 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. 12.18, 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 ПодтверждениеВызыва определено как второе сообщение в последовательности.

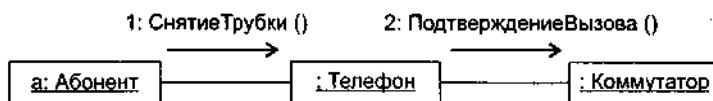


Рис. 12.18. Поток асинхронных сообщений

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

$1*[1 := 1 .. 4] : \text{РисоватьСторонуПрямоугольника}(i)$

Для моделирования ветвления после номера сообщения добавляется выражение условия, например: $[x > 0]$. Сообщение альтернативной ветви помечается таким же номером, но с другим условием: $[x \leq 0]$. Пример итерационного и разветвляющегося потока сообщений приведен на рис. 12.19.

Здесь первое сообщение повторяется 4 раза, а в качестве второго выбирается одно из двух сообщений (в зависимости от значения переменной x). В итоге экземпляр рисователя нарисует на экране прямоугольное окно, а экземпляр собеседника выведет в него соответствующее донесение.

Таким образом, для формирования диаграммы сотрудничества выполняются следующие действия:

- 1) отображаются объекты, которые участвуют во взаимодействии;
- 2) рисуются связи, соединяющие эти объекты;
- 3) связи помечаются сообщениями, которые посылают и получают выделенные объекты.

$1^*[i] = 1..4]$: РисоватьСторонуПрямоугольника(i)

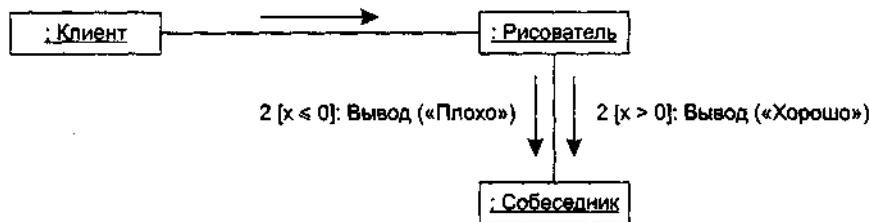


Рис. 12.19. Итерация и ветвление

В итоге формируется ясное визуальное представление потока управления (в контексте структурной организации сотрудничающих объектов).

В качестве примера на рис. 12.20 приведена диаграмма сотрудничества системы управления полетом летательного аппарата.

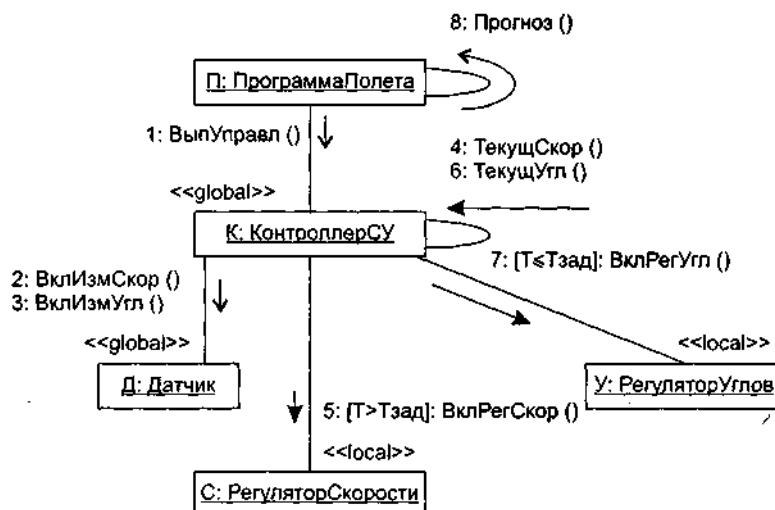


Рис. 12.20. Диаграмма сотрудничества системы управления полетом

На данной диаграмме представлены пять объектов, явно показаны характеристики видимости всех связей системы. Поток управления в системе включает восемь сообщений: четыре асинхронных и четыре синхронных сообщений. Экземпляр Контроллера СУ ждет приема и обработки сообщений:

- ВклРегСкор();
- ВрРегУгл();
- 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 поле-¹ та 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 сотрудничества (структурные характеристики объектов и связей).

Графически диаграмма последовательности — разновидность таблицы, которая показывает объекты, размещенные вдоль оси X , и сообщения, упорядоченные по времени вдоль оси Y .

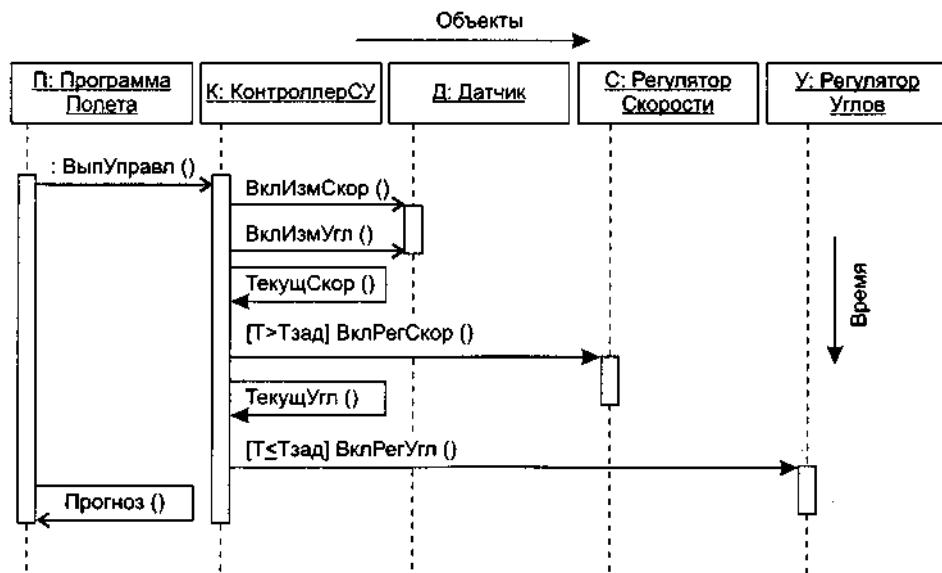


Рис. 12.21. Диаграмма последовательности системы управления полетом

Как показано на рис. 12.21, объекты, участвующие во взаимодействии, помещаются на вершине диаграммы, вдоль оси X. Обычно слева размещается объект, инициирующий взаимодействие, а справа — объекты по возрастанию подчиненности. Сообщения, посылаемые и принимаемые объектами, помещаются вдоль оси Y в порядке возрастания времени от вершины к основанию диаграммы. Используются те же синтаксис и обозначения синхронизации, что и в диаграммах сотрудничества. Таким образом, обеспечивается простое визуальное представление потока управления во времени.

От диаграмм сотрудничества диаграммы последовательности отличают две важные характеристики.

Первая характеристика — линия жизни объекта.

Линия жизни объекта — это вертикальная пунктирная линия, которая обозначает период существования объекта. Большинство объектов существуют на протяжении всего взаимодействия, их линии жизни тянутся от вершины до основания диаграммы. Впрочем, объекты могут создаваться в ходе взаимодействия. Их линии жизни начинаются с момента приема сообщения «create». Кроме того, объекты могут уничтожаться в ходе взаимодействия. Их линии жизни заканчиваются с момента приема сообщения «destroy». Как представлено на рис. 12.22, уничтожение линии жизни отмечается пометкой X в конце линии:

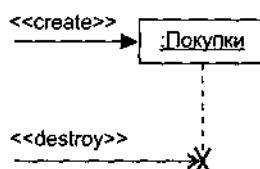


Рис. 12.22. Создание и уничтожение объекта

Вторая характеристика — фокус управления.

Фокус управления — это высокий тонкий прямоугольник, отображающий период времени, в течение которого объект выполняет действие (свою или подчиненную процедуру). Вершина прямоугольника отмечает начало действия, а основание — его завершение. Момент завершения может маркироваться сообщением возврата, которое показывается пунктирной стрелкой. Можно показать вложение фокуса управления (например, рекурсивный вызов собственной операции). Для этого второй фокус управления рисуется немного правее первого (рис. 12.23).

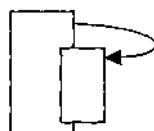


Рис. 12.23. Вложение фокусов управления

Замечания.

1. Для отображения «условности» линия жизни может быть разделена на несколько параллельных

линий жизни. Каждая отдельная линия соответствует условному ветвлению во взаимодействии. Далее в некоторой точке линии жизни могут быть снова слиты (рис. 12.24).

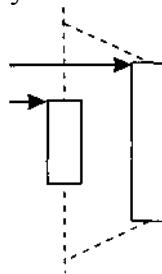


Рис. 12.24. Параллельные линии жизни

2. Ветвление показывается множеством стрелок, идущих из одной точки. Каждая стрелка отмечается сторожевым условием (рис. 12.25).

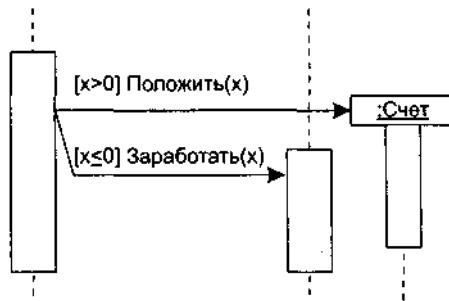


Рис. 12.25. Ветвление

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 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. Кроме того, диаграммы Use Case могут содержать пакеты, используемые для группировки элементов модели в крупные фрагменты.

Актеры и элементы Use Case

Вершинами в диаграмме Use Case являются актеры и элементы Use Case. Их обозначения показаны на рис. 12.26.

Актеры представляют внешний мир, нуждающийся в работе системы. Элементы Use Case представляют действия, выполняемые системой в интересах актеров.



Актер Элемент Use Case

Рис. 12.26. Обозначения актера и элемента Use Case

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. 12.27, в зависимости от роли Сидоров взаимодействует с разными элементами Use Case.



Рис. 12.27. 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. 12.28). Как и любая другая ассоциация, она может быть помечена именем, мощностью.

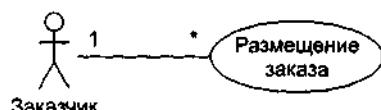


Рис. 12.28. Отношение ассоциации

Между актерами допустимо отношение обобщения (рис. 12.29), означающее, что экземпляр потомка может взаимодействовать с такими же разновидностями экземпляров элементов Use Case, что и экземпляр родителя.

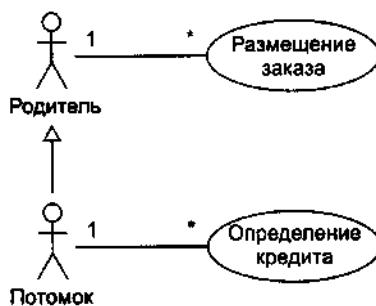


Рис. 12.29. Отношение обобщения между актерами

Между элементами Use Case определены отношение обобщения и две разновидности отношения зависимости — включения и расширения.

Отношение обобщения (рис. 12.30) 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.

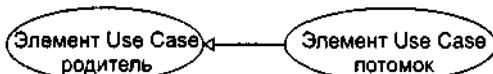


Рис. 12.30. Отношение обобщения между элементами Use Case

Отношение включения (рис. 12.31) 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 конкретизация 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).

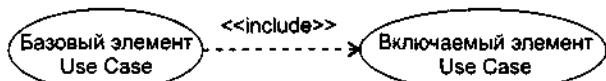


Рис. 12.31. Отношение включения между элементами Use Case

Отношение расширения (рис. 12.32) 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, вставку которых в определенную точку управляется актером.

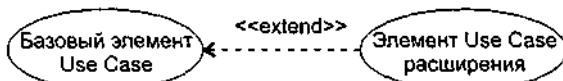


Рис. 12.32. Отношение расширения между элементами Use Case

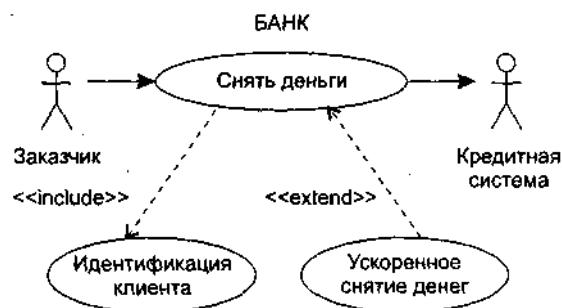


Рис. 12.33. Простейшая диаграмма Use Case для банка

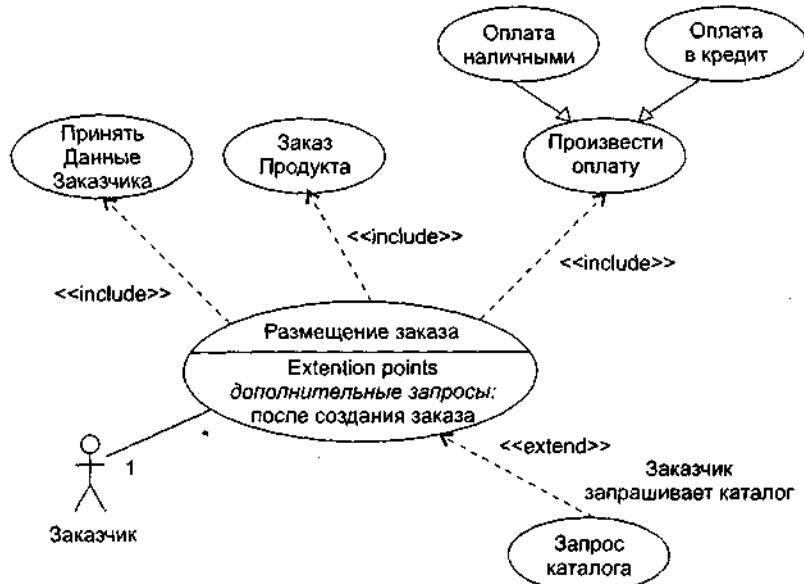


Рис. 12.34. Диаграмма Use Case для обслуживания заказчика

Пример простейшей диаграммы Use Case, в которой использованы отношения включения и расширения, приведен на рис. 12.33.

Как показано на рис. 12.34, внутри элемента Use Case может быть дополнительная секция с заголовком Extension 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.

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. 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. Говорят, что сценарий — это экземпляр элемента Use Case.

Спецификация элементов Use Case

Спецификация элемента Use Case — основной источник информации для выполнения анализа 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, правлен 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, цену. 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: введены неправильные 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*.

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, в которой использованы отношения включения и расширения, приведен на рис. 12.35.

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.

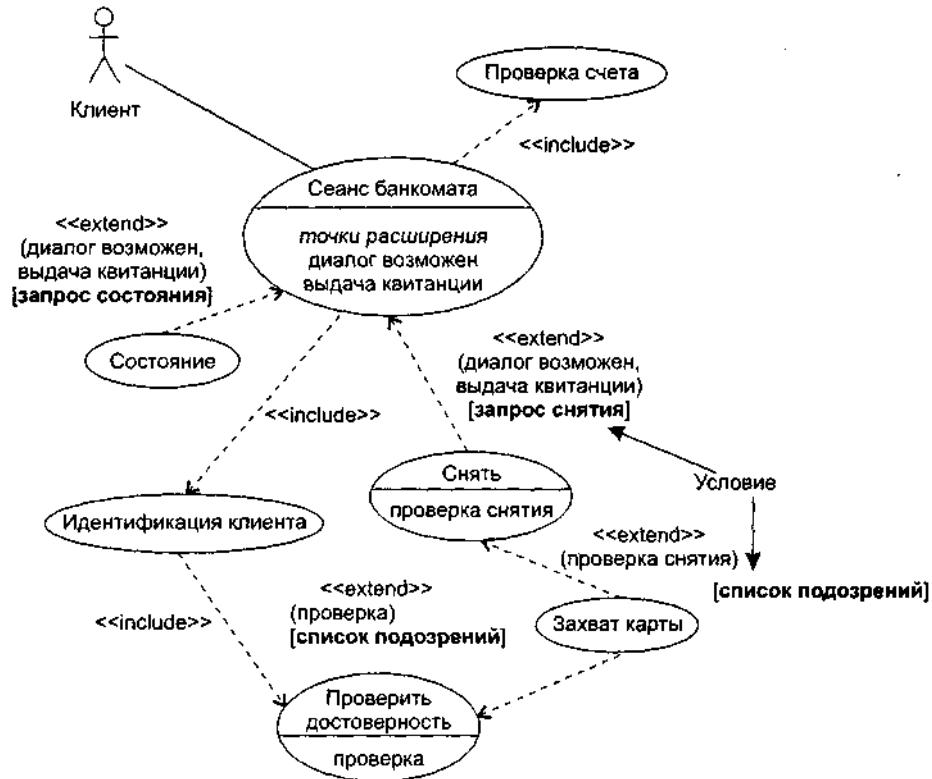


Рис. 12.35. Использование включения и расширения

Для расширяемого (базового) элемента Use Case эти условия являются внешними, то есть формируемыми вне его. Иными словами, элементу Use Case Сеанс банкомата ничего не известно 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.

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 //вторая точка расширения
конец сеанса

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.

Построение модели требований

Напомним, что основное назначение диаграмм Use Case — определение 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. 12.36.



Рис. 12.36. 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 — первый 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 для системы принимает вид, показанный на рис. 12.37.

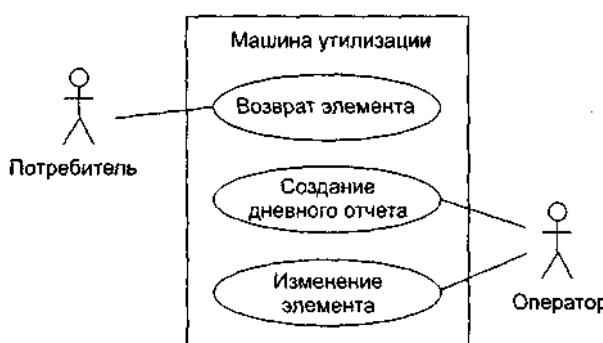


Рис. 12.37. Диаграмма Use Case для машины утилизации

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 следующий 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. 12.38).

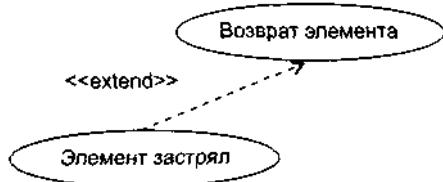


Рис 12.38. Расширение элемента Use Case возврат элемента

Описание элемента Use Case Элемент застрял может иметь следующий вид:

Если элемент застрял, для вызова Оператора вырабатывается сигнал тревоги. После удаления застрявшего элемента Оператор сбрасывает сигнал тревоги. В результате Потребитель 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. 12.39.

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.



Рис. 12.39. Применение отношения расширения

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 вариантов 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. 12.40, можно выделить абстрактный элемент Use Case Печать. Этот элемент Use Case будет специализироваться на выполнении распечаток.

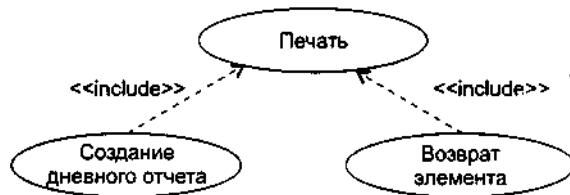


Рис. 12.40. Применение отношения включения

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. 12.41, наследниками этого актера являются Потребитель и Оператор.

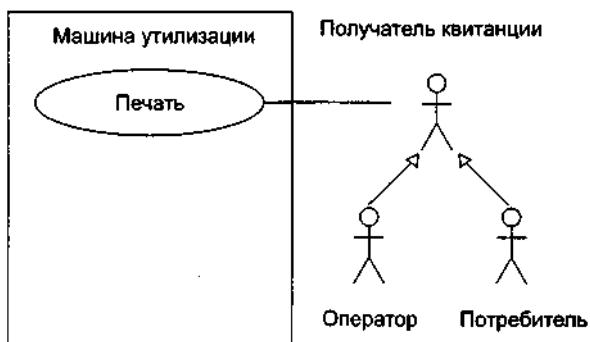


Рис. 12.41. Выделение абстрактного актера

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: устраниить повторения, ликвидировать избыточность.
3. Кроме того, это отношение часто используют для ограничения сложности большого элемента Use Case.
4. Отношение «расширяет» применяется, когда описывается вариация, дополняющая нормальное поведение.

Кооперации и паттерны

Кооперации (сотрудничества) являются средством представления комплексных решений в разработке ПО на высшем, архитектурном уровне. С одной стороны, ^ кооперации обеспечивают компактность целевой спецификации программного продукта, с другой стороны — несут в себе реализации потоков управления и данных, а также структур данных.

В терминологии фирмы Rational (вдохновителя и организатора побед языка UML) кооперации называют реализациями элементов Use Case, да и обозначения их весьма схожи (рис. 12.42).

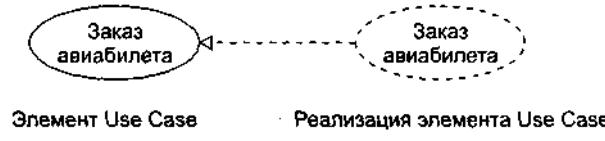


Рис. 12.42. Элемент Use Case и его реализация

Обратите внимание на то, 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. 12.43.

Соответственно, структура кооперации для заказа авиабилета может иметь вид, представленный на рис. 12.44.

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.

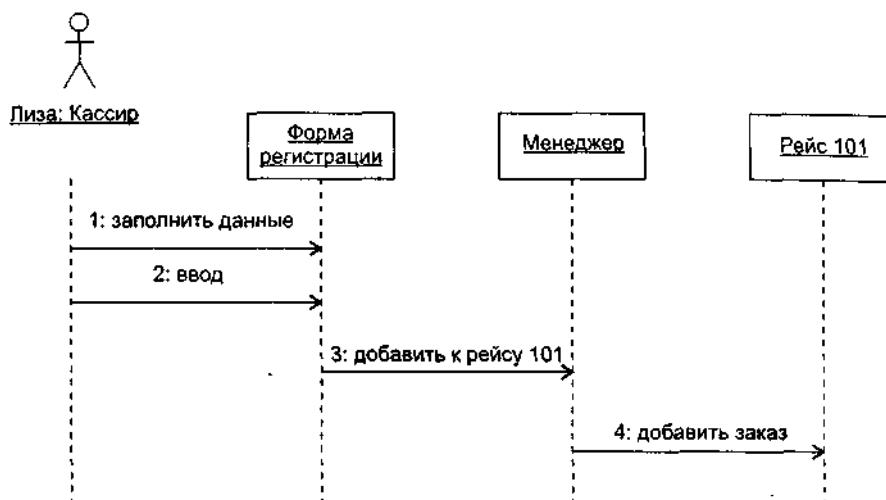


Рис. 12.43. Динамическая составляющая кооперации Заказ авиабилета



Рис. 12.44. Статическая составляющая кооперации Заказ авиабилета

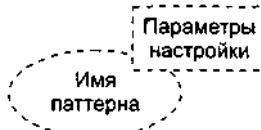


Рис. 12.45. Обозначение паттерна

Параметризованные, то есть настраиваемые кооперации называют паттернами (образцами). Паттерн является решением типичной проблемы в определенном контексте. Обозначение паттерна имеет вид, представленный на рис. 12.45.

На место параметров настройки паттерна подставляются различные фактические параметры, в результате создаются разные кооперации.

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 [26], [31], [68].

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, разделенные на три категории [31]. Как показано в табл. 12.1, по мнению «Команды четырех», описание паттерна должно состоять из четырех основных частей.

Таблица 12.1. 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

Обсудим применение нескольких паттернов из каталога «Команды четырех».

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 благотвенно 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 рассыпать сообщения другим объектам, не делая о них никаких предположений. За счет этого образуется низкое сцепление между объектами.

Решение. Принцип решения иллюстрирует рис. 12.46. 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*, синхронизуя с ним свое отображение состояния.

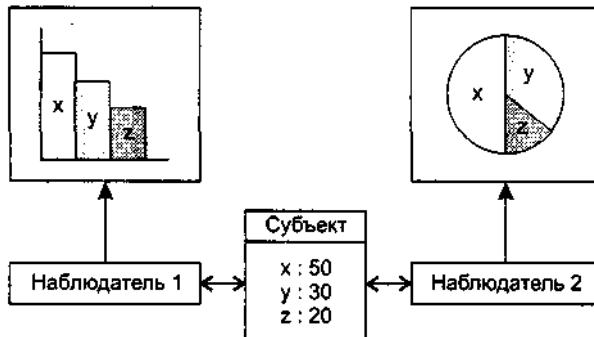


Рис. 12.46. Различные графические отображения состояния субъекта

Такое взаимодействие между элементами соответствует схеме *издатель-подписчик*. Издатель рассыпает 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. 12.47. 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).

Динамическая составляющая паттерна Наблюдатель показана на рис. 12.48. На рисунке представлено поведение паттерна при взаимодействии субъекта с двумя наблюдателями.

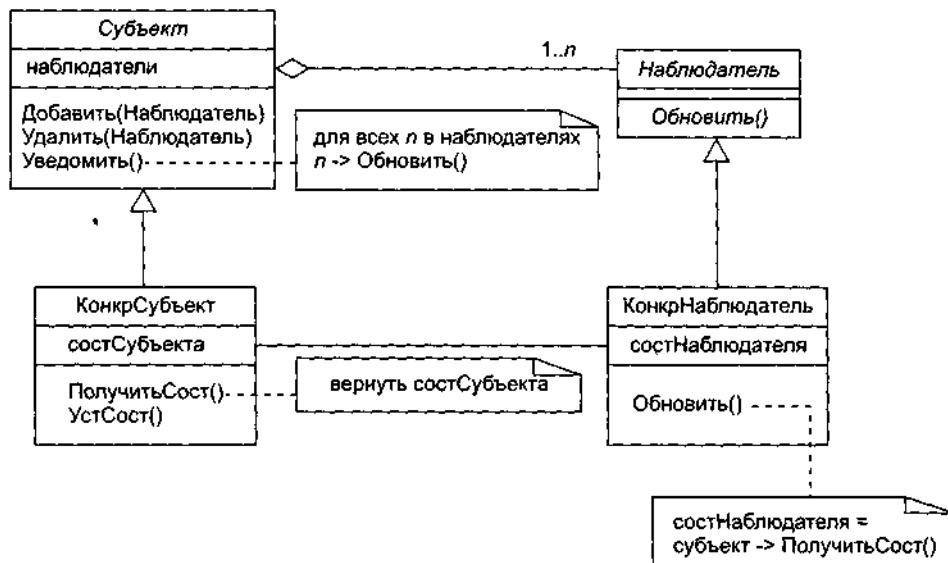


Рис. 12.47. Структурная составляющая паттерна Наблюдатель

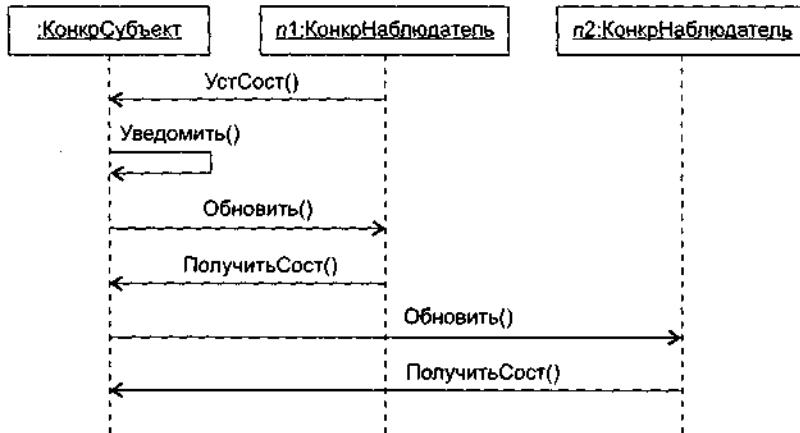


Рис. 12.48. Динамическая составляющая паттерна Наблюдатель

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).

Обозначение паттерна Наблюдатель приведено на рис. 12.49, где показано, что у него два параметра настройки — субъект и наблюдатель.

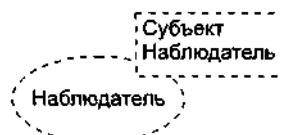


Рис. 12.49. Обозначение паттерна Наблюдатель

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. 12.50.



Рис. 12.50. 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 иерархии 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. 12.51.

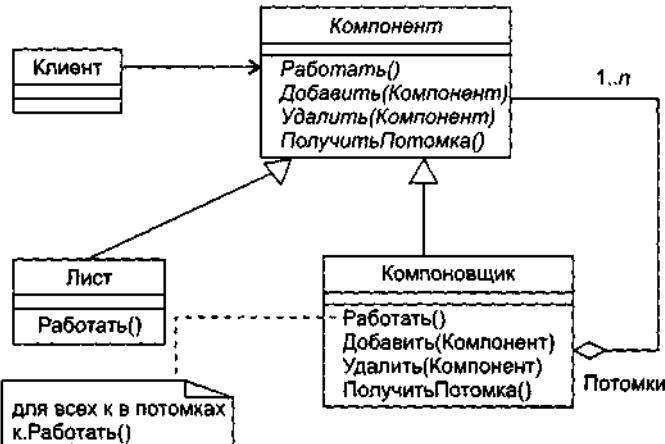


Рис. 12.51. Структурная составляющая паттерна Компоновщик

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. 12.52, где показано, что у него три параметра настройки — компонент, компоновщик и лист.

Настройку паттерна на графическое приложение иллюстрирует рис. 12.53.



Рис. 12.52. Обозначение паттерна Компоновщик

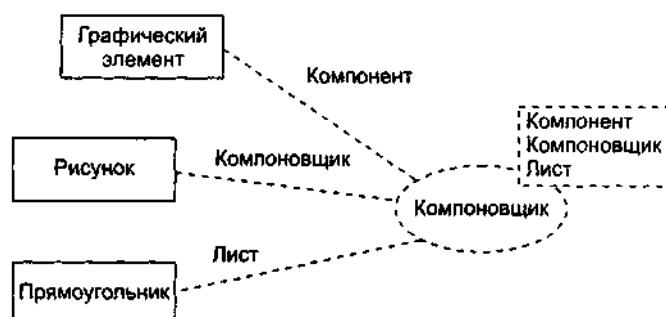


Рис. 12.53. Настройка паттерна Компоновщик

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 посыпать 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 параметризуются 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 посыпается в ходе исполнения конкретной операции Выполнять().

Структурная составляющая паттерна Команда показана на рис. 12.54. 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.

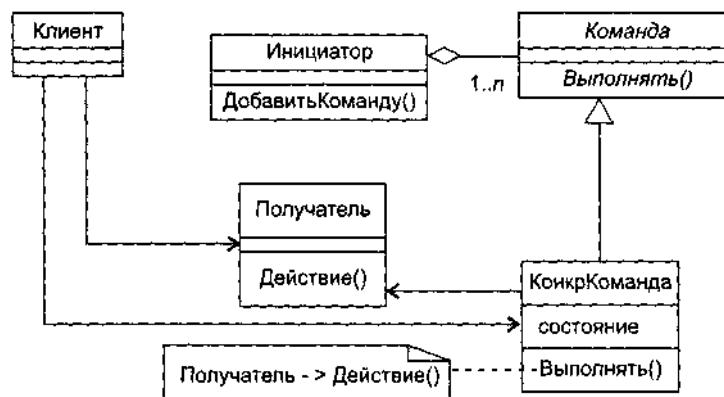


Рис. 12.54. Структурная составляющая паттерна Команда

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.

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. 12.55, где показано, что у него четыре параметра настройки — клиент, команда, инициатор и получатель.

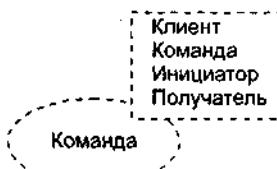


Рис. 12.55. Обозначение паттерна Команда

Настройку паттерна на приложение с графическим меню иллюстрирует рис. 12.56.



Рис. 12.56. Настройка паттерна Команда

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.

Бизнес-модели

Достаточно часто перед тем, как решиться на заказ ПО, организация проводит бизнес-моделирование. Цели бизнес-моделирования:

- отобразить структуру и процессы деятельности организации;
- обеспечить ясное, комплексное и, главное, одинаковое понимание нужд организации как сотрудниками, так и будущими разработчиками ПО;
- сформировать реальные требования к программному обеспечению деятельности организации.

Для достижения этих целей разрабатываются две модели: Q бизнес-модель Use Case; а бизнес-объектная модель.

Бизнес-модель Use Case задает внешнее представление бизнес-процессов организации (с точки зрения внешней среды — клиентов и партнеров).

Как показано на рис. 12.57, бизнес-модель Use Case строится с помощью бизнес-актеров и бизнес-элементов Use Case — простого расширения средств, используемых в обычных диаграммах Use Case.

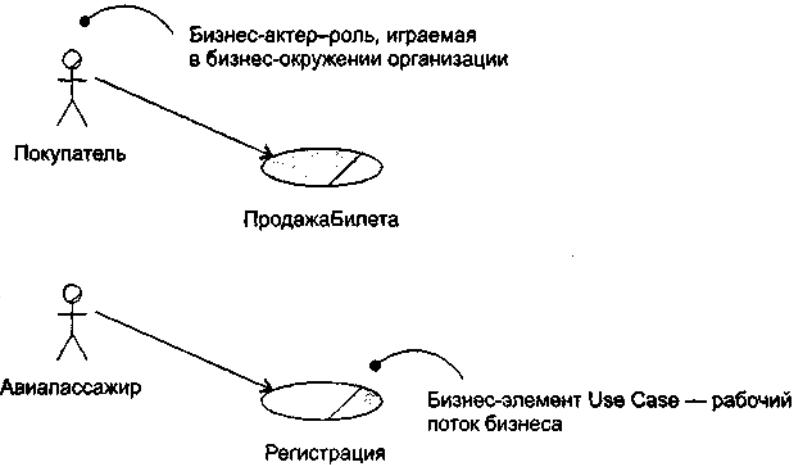


Рис. 12.57. Фрагмент бизнес-модели Use Case для аэропорта

Бизнес-актеры определяют внешние сущности и людей, с которыми взаимодействует бизнес. Бизнес-актер представляет собой человека, но информационная система, взаимодействующая с бизнесом, также может играть роль такого актера.

Бизнес-элементы Use Case изображают различные рабочие потоки бизнеса. Последовательности действий в бизнес-элементах Use Case обычно описываются диаграммами деятельности.

Бизнес-объектная модель отражает внутреннее представление бизнес-процессов организации (с точки зрения ее сотрудников).

Как показано на рис. 12.58, бизнес-объектная модель строится с помощью бизнес-работников и бизнес-сущностей — классов со специальными стереотипами. Эти классы имеют специальные графические обозначения.

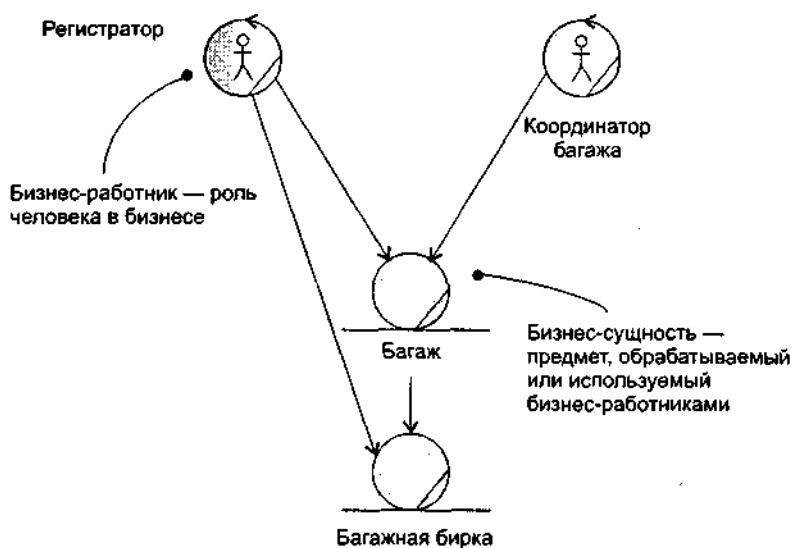


Рис. 12.58. Фрагмент бизнес-объектной модели аэропорта

Бизнес-работник — абстракция человека, действующего в бизнесе. Бизнес-сущности являются «предметами», обрабатываемыми или используемыми бизнес-работниками по мере выполнения бизнес-элемента Use Case. Например, бизнес-сущность представляет собой документ или существенную часть продукта. Фактически бизнес-объектная модель отображается с помощью диаграмм классов.

Контрольные вопросы

1. Поясните два подхода к моделированию поведения системы. Объясните достоинства и недостатки каждого из этих подходов.
2. Охарактеризуйте вершины и дуги диаграммы схем состояний. В чем состоит назначение этой диаграммы?
3. Как отображаются действия в состояниях диаграммы схем состояний?
4. Как показываются условные переходы между состояниями?

5. Как задаются вложенные состояния в диаграммах схем состояний?
6. Поясните понятие исторического подсостояния.
7. Охарактеризуйте средства и возможности диаграммы деятельности.
8. Когда не следует применять диаграмму деятельности?
9. Какие средства диаграммы деятельности позволяют отобразить параллельные действия?
10. Зачем в диаграмму деятельности введены плавательные дорожки?
11. Как представляется имя объекта в диаграмме сотрудничества?
12. Поясните синтаксис представления свойства в диаграмме сотрудничества.
13. Какие стереотипы видимости используются в диаграмме сотрудничества? Поясните их смысл.
14. В какой форме записываются сообщения в языке UML? Поясните смысл сообщения.
15. В каком отношении находятся сообщения и действия? Перечислите разновидности действий.
16. Чем отличается процедурный поток от асинхронного потока сообщений?
17. Как указывается повторение сообщений?
18. Как показать ветвление сообщений?
19. Что общего в диаграмме последовательности и диаграмме сотрудничества? Чем они отличаются друг от друга?
20. Как отображается порядок передачи сообщений в диаграмме последовательности?
21. Когда удобнее применять диаграммы последовательности?
22. Из каких элементов состоит диаграмма Use Case?
23. Какие отношения разрешены между элементами диаграммы Use Case?
24. Для чего применяют диаграммы Use Case?
25. Чем отличаются друг от друга отношения включения и расширения с точки зрения управления?
26. Каково назначение спецификации элемента Use Case и как она оформляется?
27. Что такое сценарий элемента Use Case?
28. Как документируется отношение включения?
29. Как документируется отношение расширения?
30. Каков порядок построения модели требований?
31. Каково назначение кооперации? Какие составляющие ее образуют?
32. Могут ли разные кооперации использовать одинаковые классы? Обоснуйте ответ.
33. Что такое паттерн?
34. Чем паттерн отличается от кооперации? Чем они схожи?
35. Как описывается паттерн?
36. Что нужно сделать для применения паттерна?
37. Каковы цели бизнес-моделирования?
38. Из каких частей состоит бизнес-модель? На что похожи эти части? В чем их своеобразие?

ГЛАВА 13. МОДЕЛИ РЕАЛИЗАЦИИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММНЫХ СИСТЕМ

Статические и динамические модели описывают логическую организацию системы, отражают логический мир программного приложения. Модели реализации обеспечивают представление системы в физическом мире, рассматривая вопросы упаковки логических элементов в компоненты и размещения компонентов в аппаратных узлах [8], [23], [53], [67].

Компонентные диаграммы

Компонентная диаграмма — первая из двух разновидностей диаграмм реализации, моделирующих физические аспекты объектно-ориентированных систем. Компонентная диаграмма показывает организацию набора компонентов и зависимости между компонентами.

Элементами компонентных диаграмм являются компоненты и интерфейсы, а также отношения зависимости и реализации. Как и другие диаграммы, компонентные диаграммы могут включать примечания и ограничения. Кроме того, компонентные диаграммы могут содержать пакеты или подсистемы, используемые для группировки элементов модели в крупные фрагменты.

Компоненты

По своей сути компонент является физическим фрагментом реализации системы, который заключает в себе программный код (исходный, двоичный, исполняемый), сценарные описания или наборы команд операционной систем (имеются в виде командные файлы). Язык UML дает следующее определение.

Компонент — физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов.

Интерфейс — очень важная часть понятия «компонент», его мы обсудим в следующем подразделе. Графически компонент изображается как прямоугольник с вкладками, обычно включающий имя (рис. 13.1).

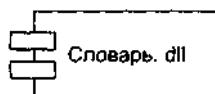


Рис. 13.1. Обозначение компонента

Компонент — базисный строительный блок физического представления ПО, поэтому интересно сравнить его с базисным строительным блоком логического представления ПО — классом.

Сходные характеристики компонента и класса:

- наличие имени;
- реализация набора интерфейсов;
- участие в отношениях зависимости;
- возможность быть вложенным;
- наличие экземпляров (экземпляры компонентов можно использовать только в диаграммах размещения).

Вы скажете — много общего. И тем не менее между компонентами и классами есть существенная разница, ее характеризует табл. 13.1.

Таблица 13.1. Различия компонентов и классов

№	Описание
1	Классы — логические абстракции, компоненты — физические предметы, которые живут в мире битов. В частности, компоненты могут «живь» в физических узлах, а классы лишены такой возможности
2	Компоненты являются физическими упаковками, контейнерами, инкапсулирующими в себе различные логические элементы. Они — элементы абстракций другого уровня
3	Классы имеют свойства и операции. Компоненты имеют только операции, которые доступны через их интерфейсы

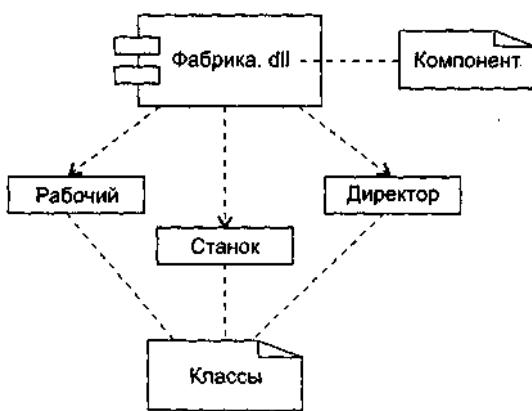


Рис. 13.2. Классы в компоненте

О чём говорят эти различия? Во-первых, класс не может «дышать» воздухом физического мира реализации. Ему нужен скафандр. Таким скафандром является компонент.

Во-вторых, им не жить друг без друга — пустые скафандры никому не нужны. Причём в скафандре-компоненте может находиться несколько классов и коопераций. Итак, в скафандре — физической реализации — располагается набор логики. Как показано на рис. 13.2, с помощью отношения

зависимости можно явно отобразить отношение между компонентом и классами, которые он реализует. Правда, чаще всего такие отношения не отображаются. Их удобно представлять в компонентной спецификации.

В-третьих, класс — душа нараспашку (он может даже показать свои свойства). Компонент всегда застегнут на все пуговицы (правда, из него торчат интерфейсные разъемы операций).

Теперь уместно перейти к обсуждению интерфейсов.

Интерфейсы

Интерфейс — список операций, которые определяют услуги класса или компонента. 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 свойства 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 предлагаемые услуги, не более того.

Очень важна взаимосвязь между компонентом и интерфейсом. Возможны два способа отображения взаимосвязи между компонентом и его интерфейсами. В первом, свернутом способе, как показано на рис. 13.3, интерфейс изображается в форме пиктограммы. Компонент Образ.java, который реализует интерфейс, соединяется со значком интерфейса (кружком) НаблюдательОбраза простой линией. Компонент РыцарьПечальногоОбраза.java, который использует интерфейс, связан с ним отношением зависимости.

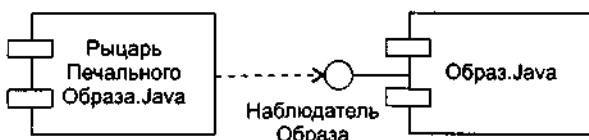


Рис. 13.3. Представление интерфейса в форме пиктограммы

Второй способ представления интерфейса иллюстрирует рис. 13.4. Здесь используется развернутая форма изображения интерфейса, в которой могут показываться его операции. Компонент, который реализует интерфейс, подключается к нему отношением реализации. Компонент, который получает доступ к услугам другого компонента через интерфейс, по-прежнему подключается к интерфейсу отношением зависимости.

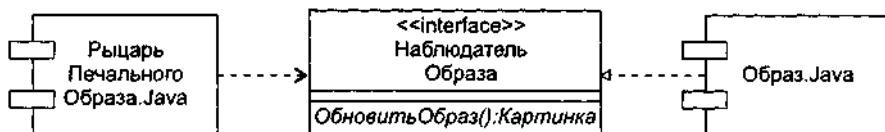


Рис. 13.4. Развернутая форма представления интерфейса

По способу связи компонента с интерфейсом различают:

- ❑ экспортруемый интерфейс — тот, который компонент реализует и предлагает как услугу клиентам;
- ❑ импортируемый интерфейс — тот, который компонент использует как услугу другого компонента.

У одного компонента может быть несколько экспортруемых и несколько импортируемых интерфейсов.

Тот факт, что между двумя компонентами всегда находится интерфейс, устраниет их прямую зависимость. Компонент, использующий 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.

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-мальчиша 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 конструкторов 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. 13.2.

Таблица 13.2. Особенности компонента

Компонент физичен. Он живет в мире битов, а не логических понятий 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, предназначеннной 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.

Разновидности компонентов

Мир современных компонентов достаточно широк и разнообразен. В языке UML для обозначения новых разновидностей компонентов используют механизм стереотипов. Стандартные стереотипы, предусмотренные в UML для компонентов, представлены в табл. 13.3.

Таблица 13.3. Разновидности компонентов

Стереотип	Описание
«executable»	Компонент, который может выполняться в физическом узле (имеет расширение .exe)
«library»	Статическая или динамическая объектная библиотека (имеет расширение .dll)
«file»	Компонент, который представляет файл, содержащий исходный код или данные (имеет расширение .ini)
«table»	Компонент, который представляет таблицу базы данных (имеет расширение .tbl)
«document»	Компонент, который представляет документ (имеет расширение .hip)

В языке UML не определены пиктограммы для перечисленных стереотипов, применяемые на практике пиктограммы компонентов показаны на рис. 13.5-13.9.

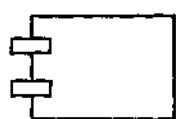


Рис. 13.5. Пиктограмма исполняемого элемента библиотеки

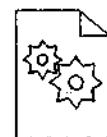


Рис. 13.6. Пиктограмма объектной библиотеки



Рис. 13.7. Пиктограмма документа



Рис. 13.8. Пиктограмма таблицы с исходным кодом или данными базы данных



Рис. 13.9. Пиктограмма документа

Использование компонентных диаграмм

Компонентные диаграммы используются для моделирования статического представления реализации системы. Это представление поддерживает управление конфигурацией системы, составляемой из компонентов. Подразумевается, что для получения работающей системы существуют различные способы сборки компонентов.

Компонентные диаграммы показывают отношения:

- периода компиляции (среди текстовых компонентов);
- периода сборки, линковки (среди объектных двоичных компонентов);
- периода выполнения (среди машинных компонентов).

Рассмотрим типовые варианты применения компонентных диаграмм.

Моделирование программного текста системы

При разработке сложных систем программный текст (исходный код) разбросан по многим файлам исходного кода. При использовании Java исходный код сохраняется в .java-файлах, при использовании C++ — в заголовочных файлах (.h-файлах) и телах (.cpp-файлах), при использовании Ada 95 — в спецификациях (.ads-файлах) и реализациях (.adb-файлах).

Между файлами существуют многочисленные зависимости компиляции. Если к этому добавить, что по мере разработки рождаются новые версии файлов, то становится очевидной необходимость управления конфигурацией системы, визуализации компиляционных зависимостей.

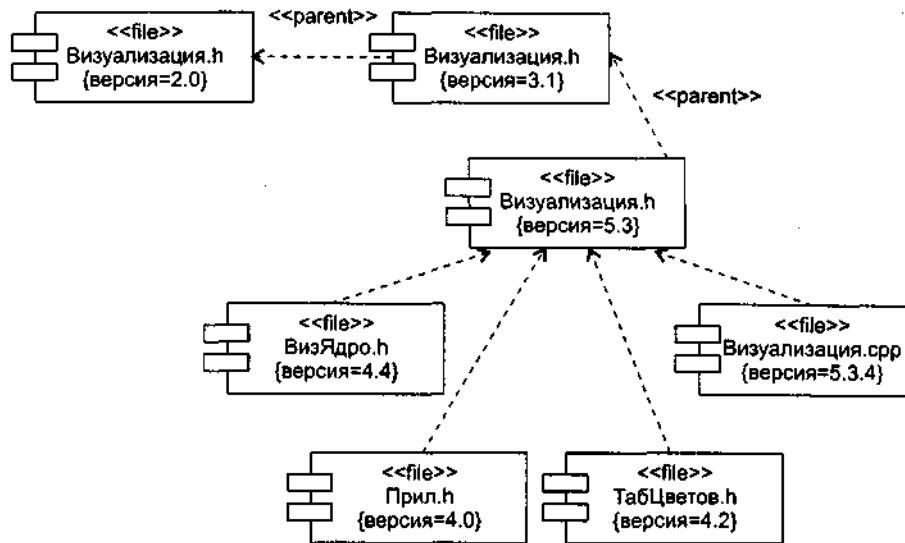


Рис. 13.10. Моделирование исходного кода



Рис. 13.11. Моделирование исходного кода с использованием пиктограмм

В качестве примера на рис. 13.10 приведена компонентная диаграмма, где изображены файлы исходного кода, используемые для построения библиотеки Визуализация.dll. Имеются четыре заголовочных файла (Визуализация.h, ВизЯдро.h, Прил.h, ТабЦветов.h), которые представляют исходный код для спецификации определенных классов. Файл реализации здесь один (Визуализация.cpp), он является реализацией одного из заголовков. Отметим, что для каждого файла явно указана его версия, причем для файла Визуализация.h показаны три версии и история их появления. На рис. 13.11 повторяется та же диаграмма, но здесь для обозначения компонентов использованы специальные пиктограммы.

Моделирование реализации системы

Реализация системы может включать большое количество разнообразных компонентов:

- исполняемых элементов;
- динамических библиотек;
- файлов данных;
- справочных документов;
- файлов инициализации;
- файлов регистрации;
- сценариев;
- файлов установки.

Моделирование этих компонентов, отношений между ними — важная часть управления конфигурацией системы.

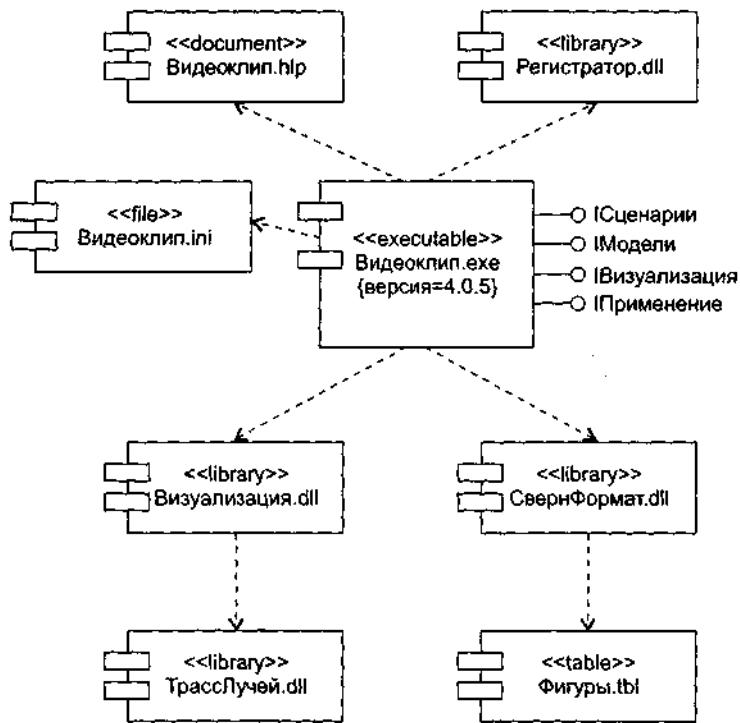


Рис. 13.12. Моделирование реализации системы

Например, на рис. 13.12 показана часть реализации системы, группируемая вокруг исполняемого элемента Видеоклип.exe. Здесь изображены четыре библиотеки (Регистратор.dll, СвернФормат.dll, Визуализация.dll, ТрассЛучей.dll), один документ (Видеоклип.hlp), один простой файл (Видеоклип.ini), а также таблица базы данных (Фигуры.tbl). В диаграмме указаны отношения зависимости, существующие между компонентами.

Для исполняемого компонента Видеоклип.exe указан номер версии (с помощью плавкой величины), представлены его экспортируемые интерфейсы (ICценарии, IVизуализация, IMодели, IPрименение). Эти интерфейсы образуют API компонента «интерфейс прикладного программирования».

На рис. 13.13 повторяется та же диаграмма, моделирующая реализацию, но здесь для обозначения компонентов использованы специальные пиктограммы.

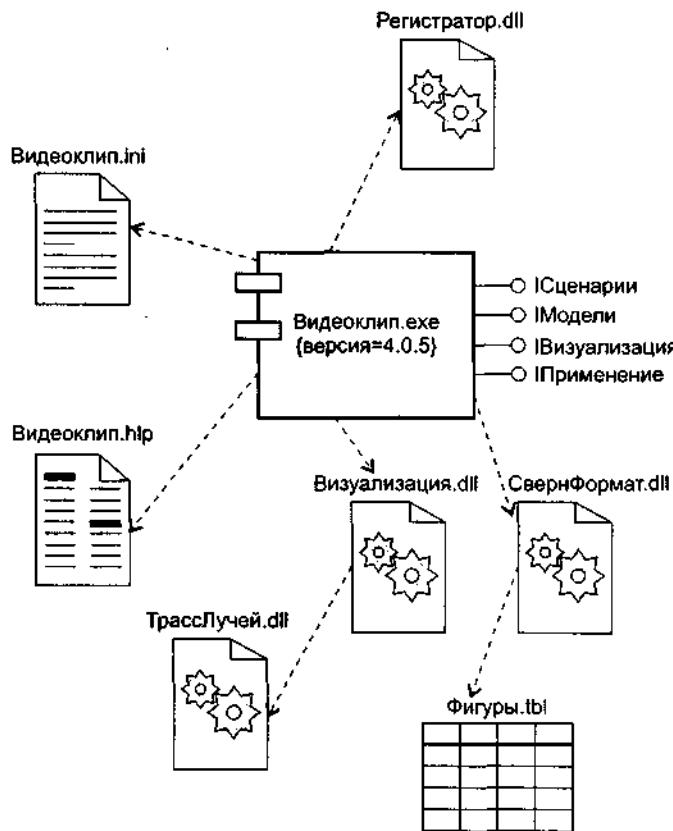


Рис. 13.13. Моделирование реализации с использованием пиктограмм

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 [5].

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.

Как показано на рис. 13.14, 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). Отметим, что здесь мы не будем применять графику языка UML, а воспользуемся принятыми в COM обозначениями.

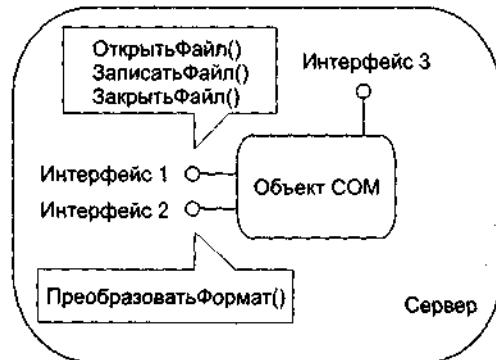


Рис. 13.14. Организация объекта СОМ

Для вызова операции интерфейса клиент объекта СОМ должен получить указатель на его интерфейс. Клиенту требуется отдельный указатель для каждого интерфейса, операции которого он намерен вызывать. Например, как показано на рис. 13.15, клиенту нашего объекта СОМ нужен один указатель интерфейса для вызова операций интерфейса РаботаСФайлами, а другой — для вызова операций интерфейса ПреобразованиеФорматов.

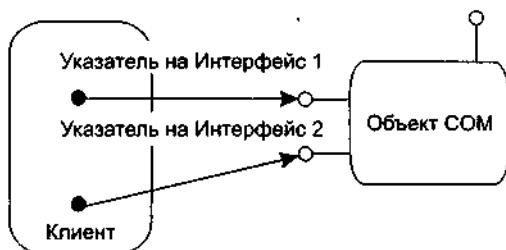


Рис. 13.15. Доступ клиента к интерфейсам объекта СОМ

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).

Благодаря СОМ клиентам нет нужды учитывать эти отличия — доступ ко всему осуществляется единообразно. Другими словами, в СОМ для доступа к услугам, предоставляемым любыми типами ПО, используется одна общая модель.

Организация интерфейса СОМ

Каждый интерфейс объекта СОМ — 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 СОМ 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 СОМ-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 "unknown.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 — РаботаСФайлами, за ним — двоеточие и имя другого интерфейса — `IUnknown`. Такая запись означает, что интерфейс РаботаСФайлами 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. 13.16.

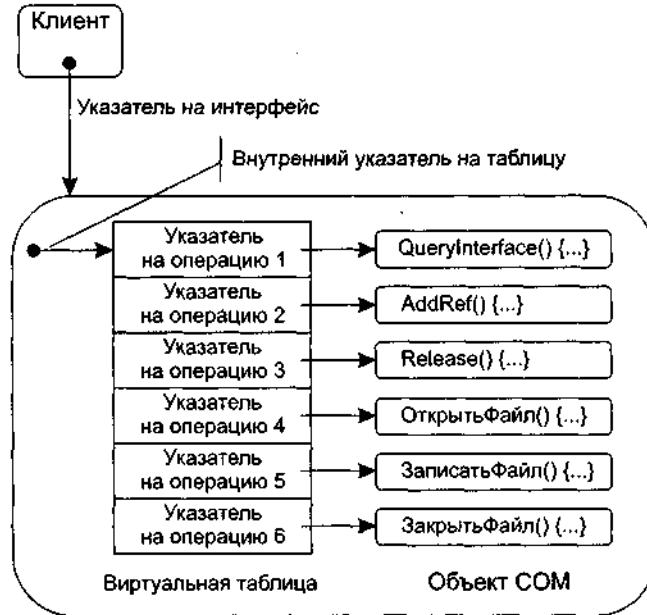


Рис. 13.16. Внутренняя структура интерфейса IWorkWithFiles

Внешний указатель на интерфейс (указатель клиента) ссылается на внутренний указатель объекта СОМ. 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 `IWorkWithFiles` указывают 4-, 5- и 6-й элементы виртуальной таблицы. Такая ситуация типична для любого СОМ-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.

Unknown — базовый интерфейс СОМ

Интерфейс `IUnknown` обеспечивает минимальное «снаряжение» каждого объекта СОМ. Он содержит три операции и предоставляет любому объекту СОМ 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 СОМ. Порядок получения других указателей на интерфейсы (для вызова их операций) поясняет рис. 13.17, где расписаны три шага работы. В качестве параметра операции `QueryInterface` задается идентификатор требуемого интерфейса (IID). Если требуемый интерфейс отсутствует, операция возвращает значение `NULL`.

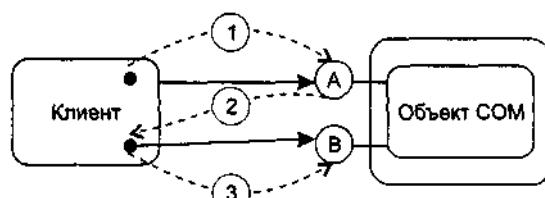


Рис. 13.17. Получение указателя на интерфейс с помощью `QueryInterface`: 1 — с помощью указателя на интерфейс А клиент запрашивает указатель на интерфейс В, вызывая `QueryInterface` (IID_B); 2 — объект возвращает указатель на интерфейс В; 3 — теперь клиент может вызывать операции из интерфейса В

Имеет смысл отметить и второе важное достоинство операции `QueryInterface`. В сочетании с

требованием неизменности СОМ-интерфейсов она позволяет «кубить двух зайцев»:

- развивать компоненты;
- обеспечивать стабильность клиентов, использующих компоненты.

Поясним это утверждение. По законам СОМ-этики новый СОМ-объект должен нести в себе и старый СОМ-интерфейс, а операция `QueryInterface` всегда обеспечит доступ к нему.

Ну а теперь обсудим правила жизни, а точнее смерти СОМ-объекта. В многоликой СОМ-среде бремя ответственности за решение вопроса о финализации должно лежать как на клиенте, так и на СОМ-объекте. Можно сказать, что фирма Microsoft (создатель этой модели) разработала самурайский кодекс поведения СОМ-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) СОМ-объекта.

Правила финализации СОМ-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 хакарири объекта-самурая:

- при получении от другого клиента указателя на интерфейс СОМ-объекта он должен вызывать в этом объекте операцию `AddRef`;
- в конце работы с объектом он обязан вызвать его операцию `Release`.

Серверы СОМ-объектов

Каждый СОМ-объект существует внутри конкретного сервера. Этот сервер содержит программный код реализации операций, а также данные активного СОМ-объекта. Один сервер может обеспечивать несколько объектов и даже несколько СОМ-классов. Как показано на рис. 13.18, 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 СОМ-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.

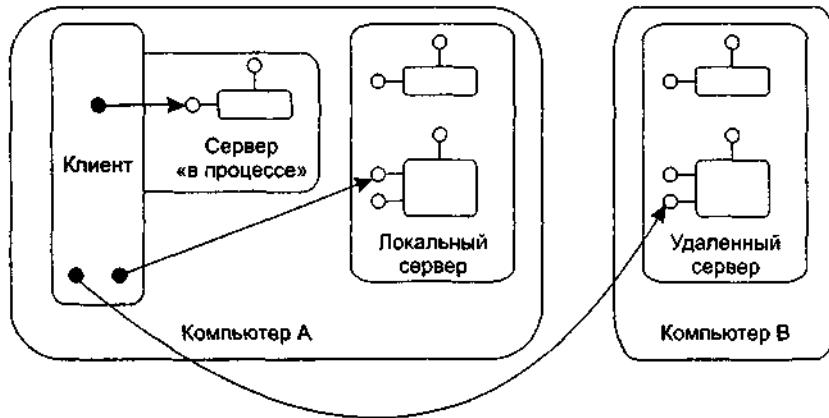


Рис. 13.18. Различные серверы СОМ-объектов

Преимущества СОМ

В качестве кратких выводов отметим основные преимущества СОМ.

1. СОМ обеспечивает удобный способ фиксации услуг, предоставляемых разными фрагментами

- ПО.
2. Общий подход к созданию всех типов программных услуг в СОМ упрощает проблемы разработки.
 3. СОМ безразличен язык программирования, на котором пишутся СОМ-объекты и клиенты.
 4. СОМ обеспечивает эффективное управление изменением программ — замену текущей версии компонента на новую версию с дополнительными возможностями.

Работа с СОМ-объектами

При работе с СОМ-объектами приходится их создавать, повторно использовать, размещать в других процессах, описывать в библиотеке операционной системы. Рассмотрим каждый из этих вопросов.

Создание СОМ-объектов

Создание СОМ-объекта базируется на использовании функций библиотеки СОМ. Библиотека СОМ:

- содержит функции, предлагающие базовые услуги объектам и их клиентам;
- предоставляет клиентам возможность запуска серверов СОМ-объектов.

Доступ к услугам библиотеки СОМ выполняется с помощью вызовов обычных функций. Чаще всего имена функций библиотеки СОМ начинаются с префикса «Co». Например, в библиотеке имеется функция CoCreateInstance.

Для создания СОМ-объекта клиент вызывает функцию библиотеки СОМ 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 СОМ-объектов.

Закончив поиск, библиотека СОМ запускает сервер класса. В результате создается неинициализированный СОМ-объект, то есть объект, данные которого не определены. Описанный процесс иллюстрирует рис. 13.19.

Как правило, после получения указателя на созданный СОМ-объект клиент предлагает объекту самоинициализироваться, то есть загрузить себя конкретными значениями данных. Этую процедуру обеспечивают стандартные СОМ-интерфейсы IPersistFile, IPersistStorage и IPersistStream.

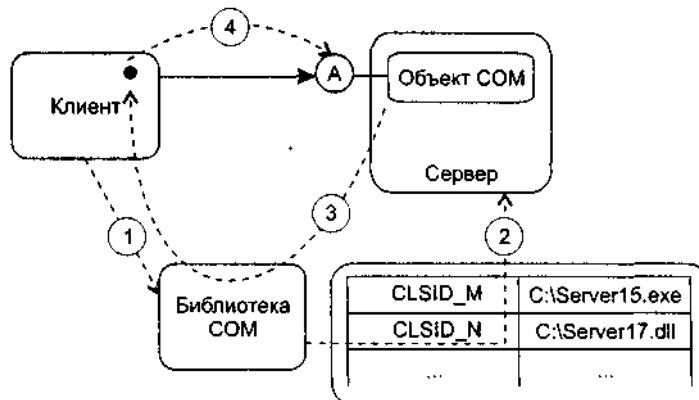


Рис. 13.19. Создание одиночного СОМ-объекта: 1 — клиент вызывает CoCreateInstance (CLSID M, IID A); 2 — библиотека СОМ находит сервер и запускает его; 3 — библиотека СОМ возвращает указатель на интерфейс A; 4 — теперь клиент может вызывать операции СОМ-объекта

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 СОМ-объектов одного и того же класса. Для этого клиент использует фабрику класса (class factory) — СОМ-объект, способный генерировать объекты одного конкретного класса.

Фабрика класса поддерживает интерфейс IClassfactory, включающий две операции. Операция CreateInstance создает СОМ-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. 13.20.

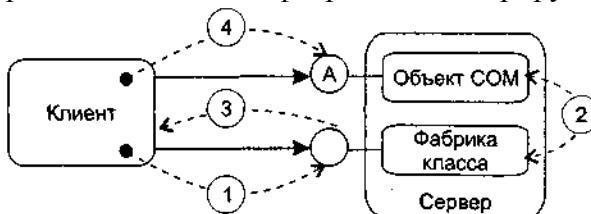


Рис. 13.20. Создание СОМ-объекта с помощью фабрики класса: 1 — клиент вызывает IClassFactory :: CreateInstance (IID A); 2 — фабрика класса создает СОМ-объект и получает указатель на его интерфейс; 3 — фабрика класса возвращает указатель на интерфейс A СОМ-объекта; 4 — теперь клиент может вызывать операции СОМ-объекта

Клиент вызывает операцию IClassFactory::CreateInstance фабрики, в качестве параметра которой задает идентификатор необходимого интерфейса объекта (IID). В ответ фабрика класса создает СОМ-объект и возвращает указатель на заданный интерфейс. Теперь клиент применяет возвращенный указатель для вызова операций СОМ-объекта.

Очень часто возникает следующая ситуация — существующий СОМ-класс заменили другим, поддерживающим как старые, так и дополнительные интерфейсы и имеющим другой CLSID. Появляется задача — обеспечить использование нового СОМ-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 СОМ-объектов

Известно, что основным средством повторного использования существующего кода является наследование реализации (новый класс наследует реализацию операций существующего класса). СОМ не поддерживает это средство. Причина — в типовой СОМ-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. СОМ 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. 13.21, когда клиент вызывает операцию внешнего объекта, эта операция, в свою очередь, вызывает операцию внутреннего объекта.

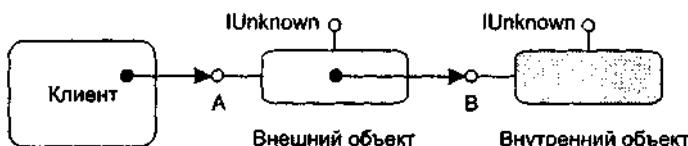


Рис. 13.21. Повторное использование СОМ-объекта с помощью включения

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. 13.22, 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.

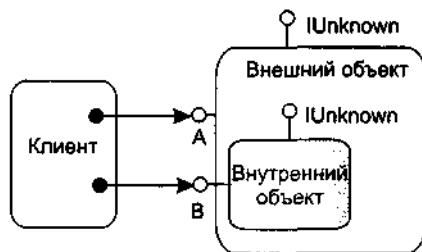


Рис. 13.22. Повторное использование СОМ-объекта с помощью агрегирования

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 СОМ-объект только в одном случае — когда СОМ-объект размещен в сервере «в процессе». В случае локального или удаленного сервера, как показано на рис. 13.23, он ссылается на посредника.

Посредник — СОМ-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.

Посредник принимает параметры, переданные клиентом, и упаковывает их для дальнейшей пересылки. Эта процедура называется маршалингом. Затем посредник (с помощью средства коммуникации) посыпает запрос в процесс, который на самом деле реализует СОМ-объект.

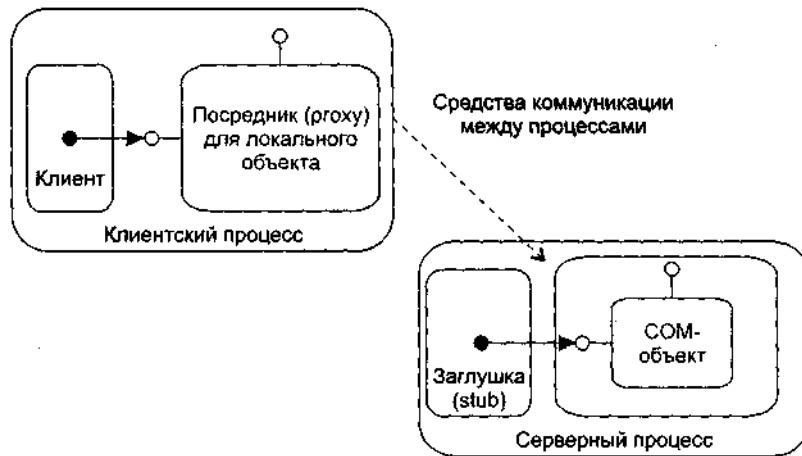


Рис. 13.23. Организация маршалинга и демаршалинга

По прибытии в процесс локального сервера запрос передается заглушке. Заглушка распаковывает

параметры запроса и вызывает операцию СОМ-объекта. Эта процедура называется демаршалингом. После завершения СОМ-операции результаты возвращаются в обратном направлении.

Код посредника и заглушки автоматически генерируется компилятором MIDL (Microsoft IDL) по IDL-описанию интерфейса.

IDL-описание библиотеки типа

Помимо информации об интерфейсах, IDL-описание может содержать информацию о библиотеке типа.

Библиотека типа определяет важные для клиента характеристики СОМ-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 3 частей. Первые две части описывают интерфейсы IPoработаСФайлами 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(E7CDODOO-1827-11CF-9946-444553540000) ]
```

```
interface IPoработаСФайлами: IUnknown
```

```
{ import "unknown.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(B2ECFAAO-1827-11CF-9946-444553540000) ]
```

```
coclass СоФайлы
```

```
{ interface IPoработаСФайлами;
```

```
    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 — СОМ-класс (`coclass`), на основе которого создается СОМ-объект.

В начале описания СОМ-класса приводится его уникальное имя (это и есть идентификатор класса — `CLSID`), затем символьное имя — СоФайлы. В теле класса перечислены имена поддерживаемых интерфейсов — РаботаСФайлами и IПреобразованиеФорматов.

Как показано на рис. 13.24, доступ к библиотеке типа выполняется по стандартному интерфейсу `ITypeLib`, а доступ к отдельным элементам библиотеки — по интерфейсу `ITypeInfo`.

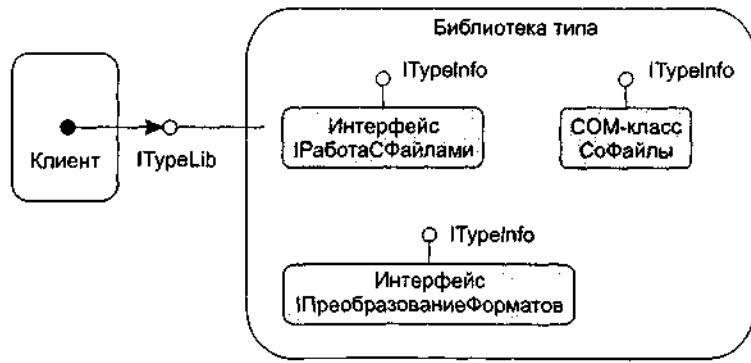


Рис. 13.24. Доступ к библиотеке типа

Диаграммы размещения

Диаграмма размещения (развертывания) — вторая из двух разновидностей диаграмм реализации UML, моделирующих физические аспекты объектно-ориентированных систем. Диаграмма размещения показывает конфигурацию обрабатывающих узлов в период работы системы, а также компоненты, «живущие» в них.

Элементами диаграмм размещения являются узлы, а также отношения зависимости и ассоциации. Как и другие диаграммы, диаграммы размещения могут включать примечания и ограничения. Кроме того, диаграммы размещения могут включать компоненты, каждый из которых должен жить в некотором узле, а также содержать пакеты или подсистемы, используемые для группировки элементов модели в крупные фрагменты. При необходимости визуализации конкретного варианта аппаратной топологии в диаграммы размещения могут помещаться объекты.

Узлы

Узел — физический элемент, который существует в период работы системы и представляет компьютерный ресурс, имеющий память, а возможно, и способность обработки. Графически узел изображается как куб с именем (рис. 13.25).



Рис. 13.25. Обозначение узла

Как и класс, узел может иметь дополнительную секцию, отображающую размещаемые в нем элементы (рис. 13.26).



Рис. 13.26. Размещение компонентов в узле

Сравним узлы с компонентами. Конечно, у них есть сходные характеристики:

- наличие имени;
- возможность быть вложенным;
- наличие экземпляров.

Последняя характеристика говорит о том, что на предыдущем рисунке изображен тип Контроллера. Изображение конкретного экземпляра, принадлежащего этому типу, представлено на рис. 13.27.

Теперь обсудим отличия узлов от компонентов. Во-первых, они принадлежат к разным уровням иерархии в физической реализации системы. Физически система состоит из узлов, а узлы — из компонентов. Во-вторых, у каждого из них свое назначение. Компонент предназначен для физической упаковки и материализации набора логических элементов (классов и коопераций). Узел же является тем местом, где физически размещаются компоненты, то есть играет роль «квартиры» для компонентов.

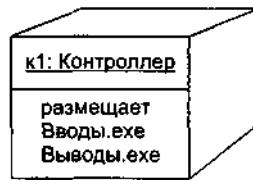


Рис. 13.27. Экземпляр узла

Отношение между узлом и компонентами, которые он размещает, можно отобразить явно. Отношение зависимости между узлом Контроллер и компонентами Вводы.exe, Выходы.exe иллюстрирует рис. 13.28. Правда, чаще всего такие отношения не отображаются. Их удобно представлять в отдельной спецификации узла.

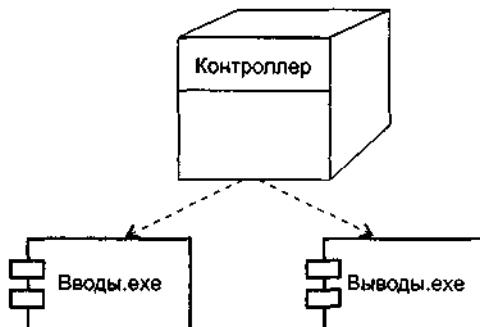


Рис. 13.28. Зависимость узла от компонентов

Группировку набора объектов или компонентов, размещаемых в узле, обычно называют распространяемым модулем.

Для узла, как и для класса, можно задать свойства и операции. Например, можно определить свойства БыстродействиеПроцессора, ЕмкостьПамяти, а также операции Запустить, Выключить.

Использование диаграмм размещения

Диаграммы размещения используются для моделирования статического представления того, как размещается система. Это представление поддерживает распространение, поставку и инсталляцию частей, образующих физическую систему.

Графически диаграмма размещения — это граф из узлов (или экземпляров узлов), соединенных ассоциациями, которые показывают существующие коммуникации. Экземпляры узлов могут содержать экземпляры компонентов, живущих или запускаемых в узлах. Экземпляры компонентов могут содержать объекты. Как показано на рис. 13.29, компоненты соединяются друг с другом пунктирными стрелками зависимостей (прямо или через интерфейсы).

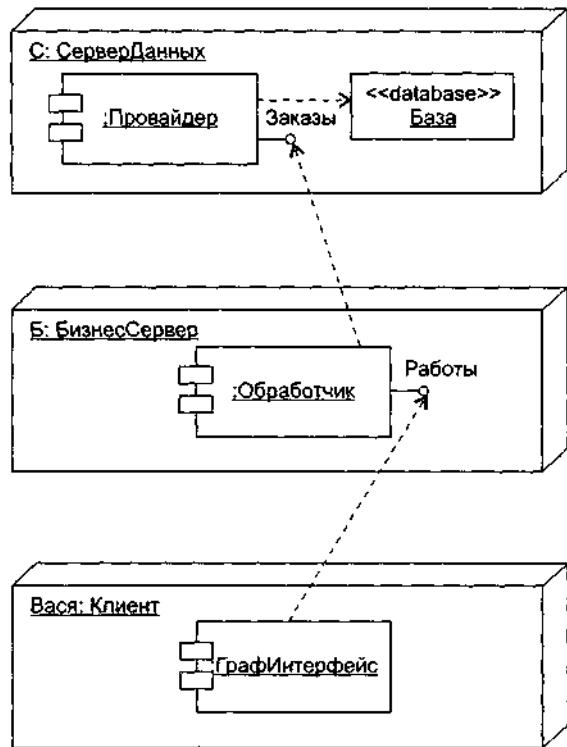


Рис. 13.29. Моделирование размещения компонентов

На этой диаграмме изображена типовая трехуровневая система:

- уровень базы данных реализован экземпляром С узла СерверДанных;
- уровень бизнес-логики представлен экземпляром Б узла БизнесСервер;
- уровень графического интерфейса пользователья образован экземпляром Вася узла Клиент.

В узле сервера данных показано размещение анонимного экземпляра компонента Провайдер и объекта База со стереотипом <<database>>. Узел бизнес-сервера содержит анонимный экземпляр компонента Обработчик, а узел клиента — анонимный экземпляр компонента ГрафИнтерфейс. Кроме того, здесь явно отображены интерфейсы компонентов Провайдер и Обработчик, имеющие, соответственно, имена Заказы и Работы.

Как представлено на рис. 13.30, перемещение компонентов от узла к узлу (или объектов от компонента к компоненту) отмечается стереотипом <<becomes>> на отношении зависимости. В этом случае считают, что компонент (объект) резидентен в узле (компоненте) только в пределах некоторого кванта времени. На рисунке видим, что возможность миграции предоставлена объектам X и Y.

Иногда полезно определить физическое распределение компонентов по процессорам и другим устройствам системы. Есть три способа моделирования распределения:

- графически распределение не показывать, а документировать его в текстовых спецификациях узлов;
- соединять каждый узел с размещаемыми компонентами отношениями зависимости;
- в дополнительной секции узла указывать список размещаемых компонентов.

Диаграмма размещения, иллюстрирующая третий способ моделирования, показана на рис. 13.31.

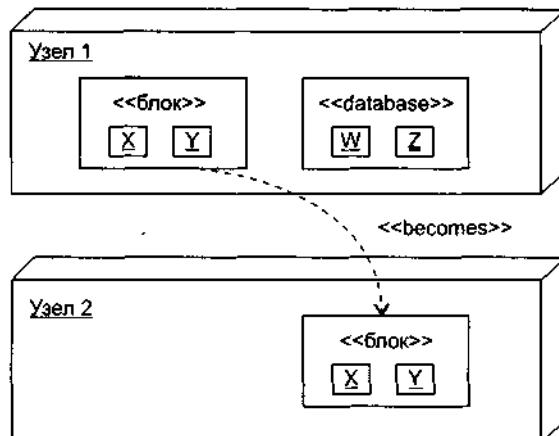


Рис. 13.30. Моделирование перемещения компонентов и объектов

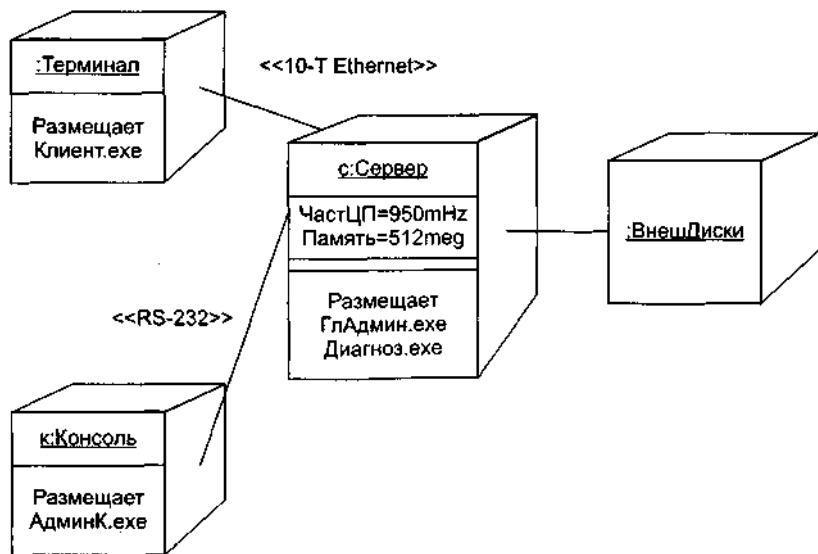


Рис. 13.31. Распределение компонентов в системе

На рисунке показаны два анонимных экземпляра узлов (:ВнешДиски, :Терминал) и два экземпляра узлов с именем (с для Сервера и к для Консоли). Каждый процессор нарисован с дополнительной секцией, в которой показаны размещенные компоненты. В экземпляре Сервера, кроме того, отображены его свойства (ЧастЦП, Память) и их значения.

С помощью стереотипов заданы характеристики физических соединений между процессорами: одно из них определено как Ethernet-соединение, другое — как последовательное RS-232-соединение.

Контрольные вопросы

1. В чем основное назначение моделей реализации?
2. Какие вершины и дуги образуют компонентную диаграмму?
3. Что такое компонент? Чем он отличается от класса?
4. Что такое интерфейс?
5. Какие формы представления интерфейса вы знаете?
6. Чем полезен интерфейс?
7. Какие разновидности компонентов вы знаете?
8. Для чего используют компонентные диаграммы?
9. Каково назначение СОМ? Какие преимущества дает использование СОМ?
10. Чем СОМ-объект отличается от обычного объекта?
11. Что должен иметь клиент для использования операции СОМ-объекта?
12. Как идентифицируется СОМ-интерфейс?
13. Как описывается СОМ-интерфейс?
14. Как реализуется СОМ-интерфейс?
15. Чего нельзя делать с СОМ-интерфейсом? Обоснуйте ответ.
16. Объясните назначение и применение операции QueryInterface.
17. Объясните назначение и применение операций AddRef и Release.
18. Что такое сервер СОМ-объекта и какие типы серверов вы знаете?
19. В чем назначение библиотеки СОМ?
20. Как создается одиночный СОМ-объект?
21. Как создаются несколько СОМ-объектов одного и того же класса?
22. Как обеспечить использование нового СОМ-класса старыми клиентами?
23. В чем состоят особенности повторного использования СОМ-объектов?
24. Какие требования предъявляет агрегация к внутреннему СОМ-объекту?
25. Что такое маршалинг и демаршалинг?
26. Поясните назначение посредника и заглушки.
27. Зачем нужна библиотека типа и как она описывается?

28. Какие вершины и ребра образуют диаграмму размещения?
29. Чем отличается узел от компонента?
30. Где можно использовать и где нельзя использовать экземпляры компонентов?
31. Как применяют диаграммы размещения?

ГЛАВА 14. МЕТРИКИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММНЫХ СИСТЕМ

При конструировании объектно-ориентированных программных систем значительная часть затрат приходится на создание визуальных моделей. Очень важно корректно и всесторонне оценить качество этих моделей, сопоставив качеству числовую оценку. 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 конструирования;
- 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: локализацию, инкапсуляцию, информационную 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 — 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 свойствами (а для агрегатов — и свойствами других классов), операциями и состояниями.

Для метрик учет инкапсуляции приводит к смещению фокуса измерений с одного модуля на группу свойств 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 (циклическая 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 — 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 свойства, 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 удачное 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 4 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 свойства 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 проблемной 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 свойства 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 свойства 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 [55]. 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 «склеиваются» 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. 14.1 представлен программный текст метода SumAndProduct. Все лексемы, входящие в секцию переменной SumN, выделены рамками. Сама секция для SumN записывается как следующая последовательность лексем:

$N_1 \cdot \text{SumN}_1 \cdot I_1 \cdot \text{SumN}_2 \cdot O_1 \cdot I_2 \cdot I_2 \cdot N_2 \cdot \text{SumN}_3 \text{ SumN}_4 \cdot I_3.$

```
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. 14.1. Секция данных для переменной SumN

Заметим, что индекс в « I_2 » указывает на второе вхождение лексемы « I » в текст метода.

Аналогичным образом определяется секция для переменной ProdN:

$$N_1 \cdot ProdN_1 \cdot I_1 \cdot ProdN_2 \cdot I_1 \cdot I_2 \cdot I_2 \cdot N_2 \cdot ProdN_3 \cdot ProdN_4 \cdot I_4$$

Для определения отношений между секциями данных можно показать профиль секций данных в методе. Для нашего примера профиль секций данных приведен в табл. 14.1.

Таблица 14.1. Профиль секций данных для метода SumAndProduct

SumN	ProdN	Оператор
		procedure SumAndProduct
1	1	(Niinteger;
1	1	varSumN, ProdNiinteger)
		var
1	1	l:integer;
		begin
2		SumN:=0
	2	ProdN:=1
3	3	for l:=1 to N do begin
3		SumN:=SumN+l
	3	ProdN:=ProdN*l
		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

$$SA(SumAndProduct) = \{N_1 \cdot SumN_1 \cdot I_1 \cdot SumN_2 \cdot I_1 \cdot I_2 \cdot I_2 \cdot N_2 \cdot SumN_3 \cdot SumN_4 \cdot I_3, \\ N_1 \cdot ProdN_1 \cdot I_1 \cdot ProdN_2 \cdot I_1 \cdot I_2 \cdot I_2 \cdot N_2 \cdot ProdN_3 \cdot ProdN_4 \cdot I_4\}.$$

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.

Склленными лексемами называют те лексемы данных, которые являются элементами более чем одной секции данных.

Сильно скленными лексемами называют те скленные 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 скленных лексем в классе *C*, она вычисляется по формуле:

$$SDC(C) = \frac{|SG(CSA(C))|}{|\text{лексемы}(C)|},$$

где *SG(CSA(C))* — объединение сильно скленных лексем каждого из методов класса *C*, *лексемы(C)* — множество всех лексем данных класса *C*.

Таким образом, класс без сильно скленных лексем имеет нулевую сильную связность по данным.

Слабая связность по данным (Weak Data Cohesion) — метрика, которая оценивает связность, базируясь на скленных лексемах. Склленные лексемы не требуют связывания всех секций данных, поэтому данная метрика определяет более слабый тип связности. Слабая связность по данным вычисляется по формуле:

$$WDC(C) = \frac{|G(CSA(C))|}{|\text{лексемы}(C)|},$$

где *G(CSA(C))* — объединение скленных лексем каждого из методов класса. Класс без скленных лексем не имеет слабой связности 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 скленную лексему, к произведению количества лексем данных в классе на количество секций данных. Метрика вычисляется по формуле:

$$DA(C) = \frac{\sum d \in G(\text{CSA}(C)) \text{ " } d \in \text{Секции}}{|\text{лексемы}(C)| \times |\text{CSA}(C)|}.$$

Приведем пример. Применим метрики к классу, профиль секций которого показан в табл. 14.2.

Таблица 14.2. Профиль секций данных для класса Stack

array	top	size	Класс Stack
2	2	2	class Stack { int *array, top, size; public:
2	2	2	Stack (int s) { size=s; array=new int [size]; top=0;}
2	2	2	int IsEmpty () { return top==0}; int Size (){ return size}; int Vtop(){
3	3	3	return array [top-1]; } void Push (int item) { if (top==size) printf ("Empty stack. \n"); else array [top++]=item;}
1	3	3	int Pop () { if (IsEmpty ()) printf ("Full stack. \n"); else --top;};
1			}

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:

$$SDC(\text{CSA}(Stack)) = 5/19 = 0,26$$

$$WDC(\text{CSA}(Stack)) = 12/19 = 0,63$$

$$DA(\text{CSA}(Stack)) = (7*2 + 5*3)/(19*3) = 0,51$$

Метрики связности по методам

Д. Билемен и Б. Кенг предложили метрики связности класса, которые основаны на прямых и косвенных соединениях между парами методов [15]. 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. 14.2 представлены отношения между элементами класса Stack. Прямоугольниками обозначены методы класса, а овалами — экземплярные переменные. Связи показывают отношения использования между методами и переменными.

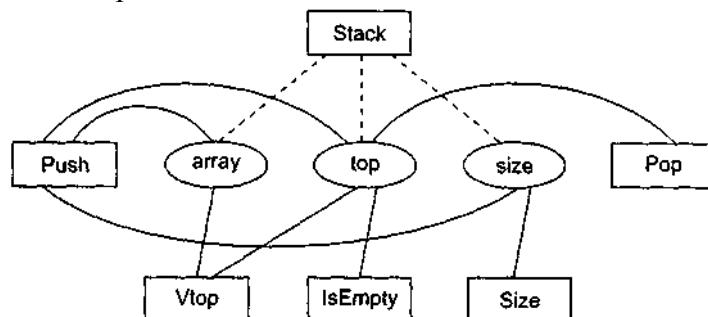


Рис. 14. 2. Отношения между элементами класса Stack

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. 14.2 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 ; 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 мульти множества (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. 14.2, 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 $AC(Q)$;
- $NIC(C)$ — 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:

$$LCC(C) \geq TCC(C).$$

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 [77].

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 свойства 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 гарантируют 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 [38].

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. 14.3, где n — количество методов класса CC, ω — количество методов CC, потенциально затрагиваемых изменением.

Таблица 14.3. Вклад отношений между клиентом и сервером в зависимость изменения

Тип отношения	ω
---------------	----------

SC не используется классом CC	0
SC — класс экземплярной переменной в классе CC	n
Локальные переменные типа SC используются внутри /-методов класса CC	j
SC является суперклассом CC	n
SC является типом параметра для /-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 ($0 < k < 1$). Вклад доступа к интерфейсу в зависимость изменения можно учесть умножением на $(1 - k)$.

Метрика для вычисления степени CDBC имеет вид:

$$A = \sum_{\substack{\text{Доступ} \\ \text{к реализации}}} \alpha_i + (1-k) \times \sum_{\substack{\text{Доступ} \\ \text{к интерфейсу}}} \alpha_i ;$$

$$\text{CDBC(CC, SC)} = \min(n, A).$$

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 $M_i (1 \leq i \leq n)$ 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:

- $L_i (1 \leq i \leq n)$ — 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 ;
- $T_i (1 \leq i \leq n)$ — 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 [24]. 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 Чидамбара-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, \dots, 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++ [45]. 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 от листа до корня дерева наследования классов. Для показанной на рис. 14.3 иерархии классов метрика DIT равна 3.

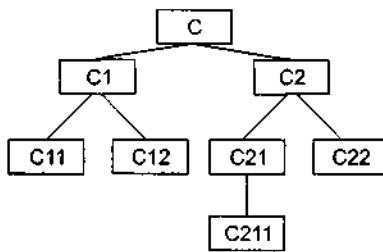


Рис. 14.3. Дерево наследования классов

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 проекта, 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. 14.3 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 [22].

Метрика 4: Сцепление между классами объектов СВО (Coupling between object classes)

СВО — 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 следующий вид: СВО равно количеству сцеплений класса; сцепление образует вызов метода или свойства в другом классе.

Данная метрика характеризует статическую составляющую внешних связей классов.

С ростом СВО многократность использования класса, вероятно, уменьшается. Очевидно, что чем больше независимость класса, тем легче его повторно использовать в другом приложении.

Высокое значение СВО усложняет модификацию и тестирование, которое следует за выполнением модификации. Понятно, что, чем больше количество сцеплений, тем выше чувствительность всего проекта к изменениям в отдельных его частях. Минимизация межобъектных сцеплений улучшает модульность и способствует инкапсуляции проекта.

СВО 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 усложняются 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 свойствам (экземплярным переменным). Метрика *LCOM* показывает, насколько методы не связаны друг с другом через свойства (переменные). Если все методы обращаются к одинаковым свойствам, 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

$$\begin{aligned} \text{НЕ СВЯЗАНЫ} &= \text{card } \{I_{ij} \mid I_i \cap I_j = \emptyset\}, \\ \text{СВЯЗАНЫ} &= \text{card } \{I_{ij} \mid I_i \cap I_j \neq \emptyset\}. \end{aligned}$$

Тогда формула для вычисления недостатка связности в методах примет вид

$$LCOM = \begin{cases} \text{НЕ СВЯЗАНЫ} - \text{СВЯЗАНЫ}, & \text{если } (\text{НЕ СВЯЗАНЫ} > \text{СВЯЗАНЫ}); \\ 0 & \text{в противном случае.} \end{cases}$$

Можно определить метрику по-другому: LCOM — это количество пар методов, не связанных по свойствам класса, минус количество пар методов, имеющих такую связь.

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

$$\begin{aligned} \text{НЕ СВЯЗАНЫ} &= \text{card } (I_{13}, I_{14}, I_{23}, I_{24}, I_{34}) = 5; \text{СВЯЗАНЫ} = \text{card } (I_{12}) = 1. \\ LCOM &= 5 - 1 = 4. \end{aligned}$$

Пример 2: В классе используются методы: $M1, M2, M3$. Для каждого метода задан свой набор экземплярных переменных:

$$\begin{aligned} I_1 &= \{a, b\}; I_2 = \{a, c\}; I_3 = \{x, y\}, \\ \text{НЕ СВЯЗАНЫ} &= \text{card } (I_{13}, I_{23}) = 2; \text{СВЯЗАНЫ} = \text{card } (I_{12}) = 1, \\ LCOM &= 2 - 1 = 1. \end{aligned}$$

Связность методов внутри класса должна быть высокой, так как это соответствует инкапсуляции. Если LCOM имеет высокое значение, то методы слабо связаны друг с другом через свойства. 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)

Setjnethod (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 определить метрику WMC2:

$$WMC2 = \sum_{i=1}^n (\text{Количество параметров } i\text{-го метода}).$$

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):

$$\text{ANAM} = WMC2/WMC.$$

Полезность метрики ANAM объяснить еще легче. Она ориентирована на принятые в ОО-проектировании решения — применять простые операции с малым количеством аргументов, а несложные операции — с многочисленными аргументами.

Еще одно предложение — ввести метрику, симметричную метрике LCOM. В то время как формула метрики LCOM имеет вид:

$$LCOM = \max(0, \text{НЕ СВЯЗАНЫ} - \text{СВЯЗАНЫ}),$$

симметричная ей метрика *Нормализованная NLCOM* вычисляется по формуле:

$$NLCOM = \text{СВЯЗАНЫ}/(\text{НЕ СВЯЗАНЫ} + \text{СВЯЗАНЫ}).$$

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* (Behavioural 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. 14.4, представлен в табл. 14.4.

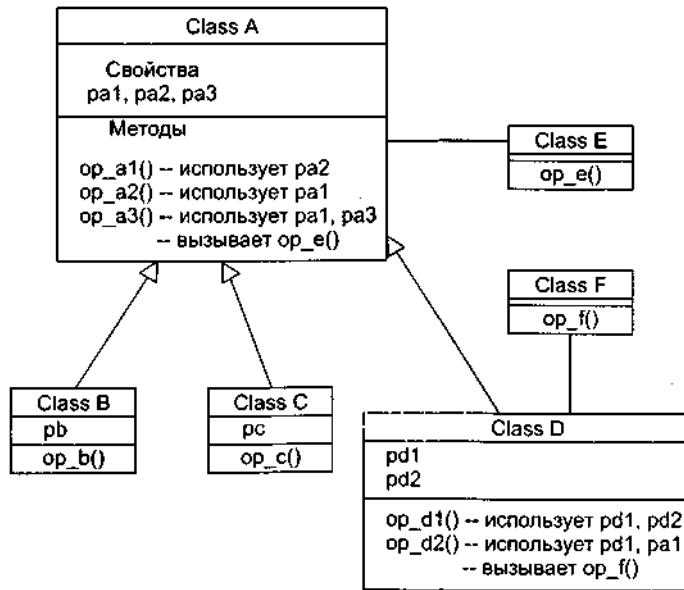


Рис. 14.4. Структура классов для расчета метрик Чидамбера-Кемерера

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 СВО 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 четырех методов (три объявлены в этом классе, а четвертый метод op_e() вызывается из op_a3()).

Таблица 14.4. Пример расчета метрик Чидамбера-Кемерера

Имя класса	WMC	DIT	NOC	СВО	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

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 свойств, третья пара имеет общее свойство (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 СВО равна 2, так как здесь используются свойство pa1 и метод op_f() из других классов. Метрика LCOM в этом классе равна 0, поскольку методы op_d1() и op_d2() связаны по свойству pd1, а отрицательное значение запрещено.

Метрики Лоренца и Кидда

Коллекция метрик Лоренца и Кидда — результат практического, промышленного подхода к оценке ОО-проектов [45].

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 свойств 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 свойств (вместе с приватными и наследуемыми экземплярными свойствами), 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 свойствам придают больший удельный вес. Причина — приватные операции и свойства обеспечивают специализацию и более локализованы в проекте.

Могут вычисляться средние количества свойств 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 NOO велико, 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 свойств. 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:

$$SI = (NOO \times \text{уровень}) / M_{общ},$$

где *уровень* — номер уровня в иерархии, на котором находится подкласс, $M_{общ}$ — общее количество методов класса.

Пример расчета индексов специализации приведен на рис. 14.5.

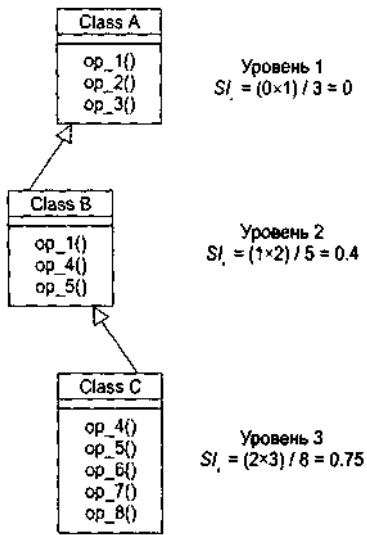


Рис. 14_05

Рис. 14.5. Расчет индексов специализации классов

Чем выше значение SI_i , 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 ОС (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 ОС суммированием оценок с весовыми коэффициентами, приведенными в табл. 14.5.

Таблица 14.5. Весовые коэффициенты для метрики ОС

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 ОС.

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, свойств 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 (NumberofSUBsystem)

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 человека-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 [6]. 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 свойства (AHF);
- 3) фактор наследования метода (MIF);
- 4) фактор наследования свойства (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 ТС — количество классов в системе.

Если видимость m -го метода i -го класса из j -го класса вычислять по выражению:

$$is_visible(M_{mi}, C_j) = \begin{cases} 1, & \text{if } \begin{cases} j \neq 1 \\ C_j \text{ может вызвать } M_{mi} \end{cases}, \\ 0, & \text{else} \end{cases}$$

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 свойства AHF (Attribute Hiding Factor)

Введем обозначения:

- $A_v(C_i)$ — количество видимых свойств в классе C_i (интерфейс класса);

- $A_h(C_i)$ — количество скрытых свойств в классе C_i (реализация класса);
- $A_d(C_i) = A_v(C_i) + A_h(C_i)$ — общее количество свойств, определенных в классе C_i (унаследованные свойства 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 ТС — количество классов в системе.

Если видимость m -го свойства i -го класса из j -го класса вычислять по выражению:

$$is_visible(A_{mi}, C_j) = \begin{cases} 1, & \text{if } \begin{cases} j \neq 1 \\ C_j \text{ может вызвать } A_{mi} \end{cases} \\ 0, & \text{else} \end{cases}$$

а процентное количество классов, которые видят m -е свойство i -го класса, определять по соотношению:

$$V(A_{mi}) = \frac{\sum_{i=1}^{TC} is_visible(A_{mi}, C_j)}{TC - 1},$$

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 свойств во всех классах. Закрытость свойства — процентное количество классов, из которых данное свойство невидимо. Знаменатель AHF — общее количество свойств, определенных в рассматриваемой системе.

В идеальном случае все свойства 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_i .

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 уменьшаются 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 свойства AIF (Attribute Inheritance Factor)

Введем обозначения:

- $A_i(C_i)$ — количество унаследованных и не переопределенных свойств в классе C_i ;
- $A_0(C_i)$ — количество унаследованных и переопределенных свойств в классе C_i ;
- $A_n(C_i)$ — количество новых (не унаследованных и переопределенных) свойств в классе C_i ;
- $A_d(C_i) = A_n(C_i) + A_0(C_i)$ — количество свойств, определенных в классе C_i ;
- $A_a(C_i) = A_d(C_i) + A_i(C_i)$ — общее количество свойств, 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) свойств во всех классах рассматриваемой системы. Знаменатель AIF — это общее количество доступных свойств (локально определенных и унаследованных) для всех классов.

Метрика 5: Фактор полиморфизма POF (Polymorphism Factor)

Введем обозначения:

- $M_0(C_i)$ — 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 ;
- $M_d(C_i) = M_n(C_i) + M_0(C_i)$ — количество методов, определенных в классе C_i .

Тогда формула метрики POF примет вид:

$$POF = \frac{\sum_{i=1}^{TC} M_0(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}.$$

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 свойство или метод класса-поставщика.

$$is_client(C_c, C_s) = \begin{cases} 1, & \text{if } C_c \Rightarrow C_s \cap C_c \neq C_s, \\ 0, & \text{else,} \end{cases}$$

Если наличие отношения «клиент-поставщик» определять по выражению:

$$COF = \frac{\sum_{i=1}^{TC} \sum_{j=1}^{TC} is_client(C_i, C_j)}{TC^2 - TC}.$$

то формула для вычисления метрики COF примет вид:

Знаменатель COF соответствует максимально возможному количеству сцеплений в системе с ТС-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 усложняет сопровождение ПО.

Метрики для объектно-ориентированного тестирования

Рассмотрим проектные метрики, которые, по мнению Р. Байндера (Binder), прямо влияют на тестируемость ОО-систем [17]. 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 свойства наследуются от других классов и поэтому видимы для этих классов. Защищенные свойства являются специализацией и приватны для определенного подкласса. Эта метрика показывает процент публичных свойств 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 свойствам 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 О-О-систем FIN фиксирует множественное наследование. Значение $FIN > 1$ указывает, что класс наследует свои свойства и операции от нескольких корневых классов. Следует избегать $FIN > 1$ везде, где это возможно.

Метрика 6: Количество детей НОС

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, СВО, 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

1. Какие факторы объектно-ориентированных систем влияют на метрики для их 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. Сравните наборы Чидамбера-Кемерера и Лоренца-Кидда. Чем они похожи? В чем различие?
15. На какие цели ориентирован набор метрик Фернандо Абреу?
16. Охарактеризуйте состав набора метрик Фернандо Абреу.
17. Сравните наборы Чидамбера-Кемерера и Фернандо Абреу. Чем они похожи? В чем различие?
18. Сравните наборы Лоренца-Кидда и Фернандо Абреу. Чем они похожи? В чем различие?
19. Дайте характеристику метрик для объектно-ориентированного тестирования.

ГЛАВА 15. УНИФИЦИРОВАННЫЙ ПРОЦЕСС РАЗРАБОТКИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПС

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 конструирования программных приложений. Далее описывается содержание XP-процесса экстремальной разработки, являющегося носителем адаптивной технологии, применяемой в условиях частого изменения требований заказчика.

Эволюционно-инкрементная организация жизненного цикла разработки

Рассматриваемый подход является развитием спиральной модели Боэма [8], [40], [44], [57]. 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 сбор 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. 15.1.

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.

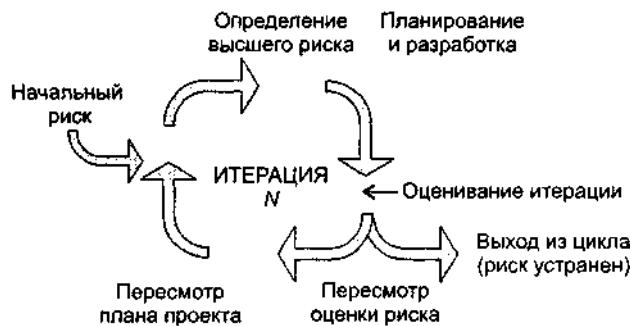


Рис. 15.1. Типовая итерация эволюционно-инкрементного жизненного цикла

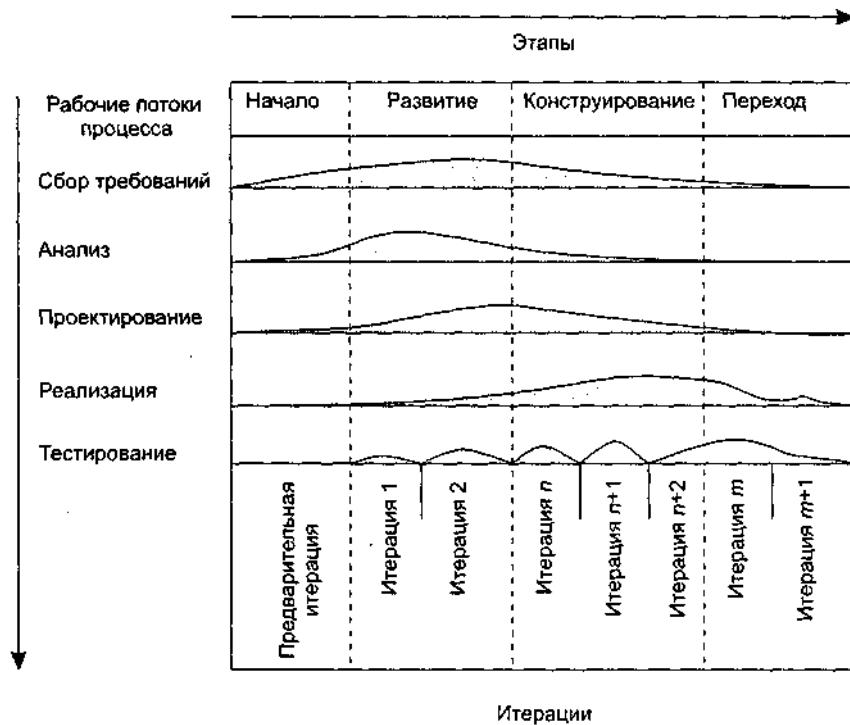


Рис. 15.2. Два измерения унифицированного процесса разработки

Как показано на рис. 15.2, 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 сбора требований до тестирования. От итерации к итерации меняется лишь удельный вес каждого рабочего потока — он зависит от этапа. На этапе Начало основное внимание уделяется сбору требований, на этапе Развитие — анализу и проектированию, на этапе Конструирование — реализации, на этапе Переход — тестированию. Каждый этап и итерация уменьшают 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:

- Сбор 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 сбора 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;
- проектная модель. Определяет словарь проблемы и ее решение;
- модель размещения. 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 сконструирована система;
- набор реализации. Описывает сборку разработанных программных компонентов;
- набор размещения. 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 проектную модель, 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 размещения группирует всю информацию об упаковке, отправке, установке и запуске системы.

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 [20].

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, реализации, формирования интерфейса, тестирования и сопровождения;
- неточность 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 расписания 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. 15.1, сводятся в таблицу.

Таблица 15.1. 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
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. 15.1 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 приходится 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. 15.3 показана кривая останова, составленная из эталонных точек.

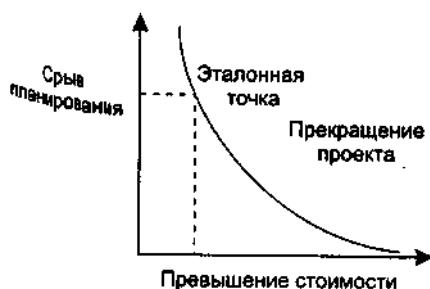


Рис. 15.3. Кривая останова проекта

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 — 2-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.

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 проектируемой системы (ее предназначение, границы, интерфейсы с внешней средой, критерий признания — приемки);
- определить элементы Use Case, критические для 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); а 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, устраниить 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. 15.4, обычно используется «4+1»-представление [44].



Рис. 15.4. «4+1»-представление архитектуры

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.

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 размещения 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 свойства (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 соответствуют элементы 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 свойств) 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 итерации.

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 бета-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 14.

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, переопределляемых 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_S вычисляются для всей системы.

На последней итерации дополнительно вычисляется набор метрик MOOD, предложенный Абреу:

- MHF — фактор закрытости метода;
- AHF — фактор закрытости свойства;
- MIF — фактор наследования метода;
- AIF — фактор наследования свойства;
- POF — фактор полиморфизма;
- COF — фактор сцепления.

Пример объектно-ориентированной разработки

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) — среда, управляемая событиями. Действия в среде инициируются функциями обратного вызова, которые вызываются в ответ на событие — пользовательский ввод. Ядром WUI является цикл обработки событий, который организуется менеджером ввода.

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. 15.5.

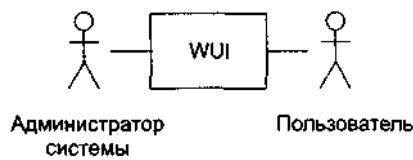


Рис. 15.5. Внешнее окружение WUI

Идентификация элементов Use Case

В WUI могут быть выделены два элемента Use Case:

- ❑ управление окнами;
- ❑ использование окон.

Диаграмма Use Case для среды WUI представлена на рис. 15.6.



Рис. 15.6. Диаграмма Use Case для среды 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 сохраняется 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.

На следующем шаге сценарии элементов Use Case преобразуются в диаграммы последовательности — за счет этого достигается формализация описаний, требуемая для построения диаграмм классов. Для построения диаграмм последовательности проводится грамматический разбор каждого сценария элемента Use Case: значащие существительные превращаются в объекты, а значащие глаголы — в сообщения, пересылаемые между объектами.

Диаграммы последовательности

Диаграммы изображены на рис. 15.7-15.11.

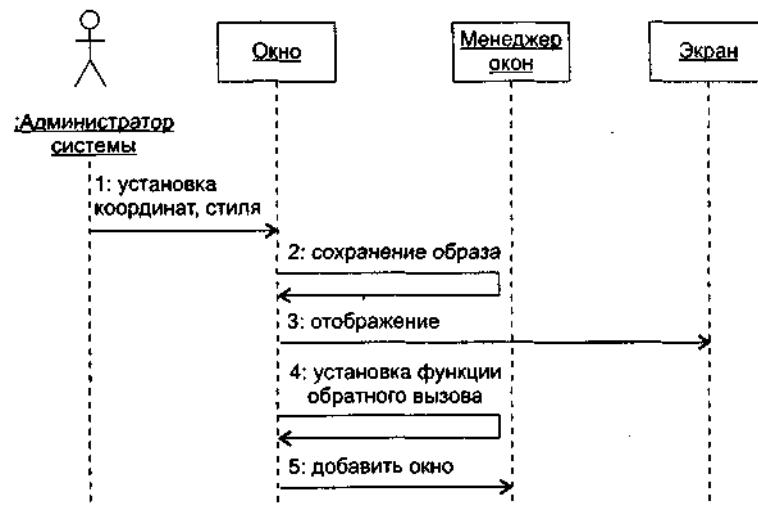
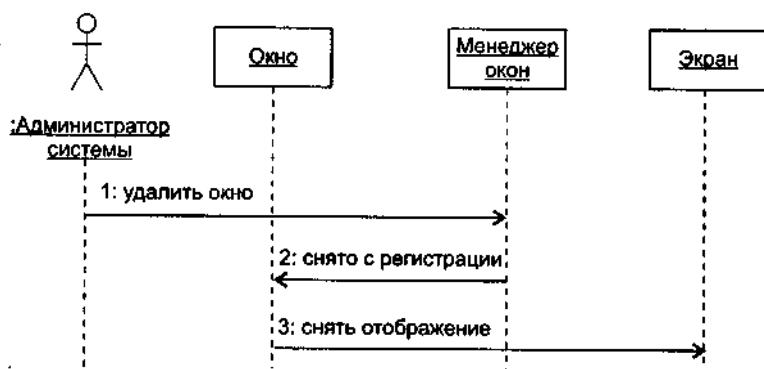


Рис. 15.7. Диаграмма последовательности Создание окна



Рис. 15.8. Диаграмма последовательности Изменение стиля рамки



15.9. Диаграмма последовательности Уничтожение окна



Рис. 15.10. Диаграмма последовательности Использование простого окна

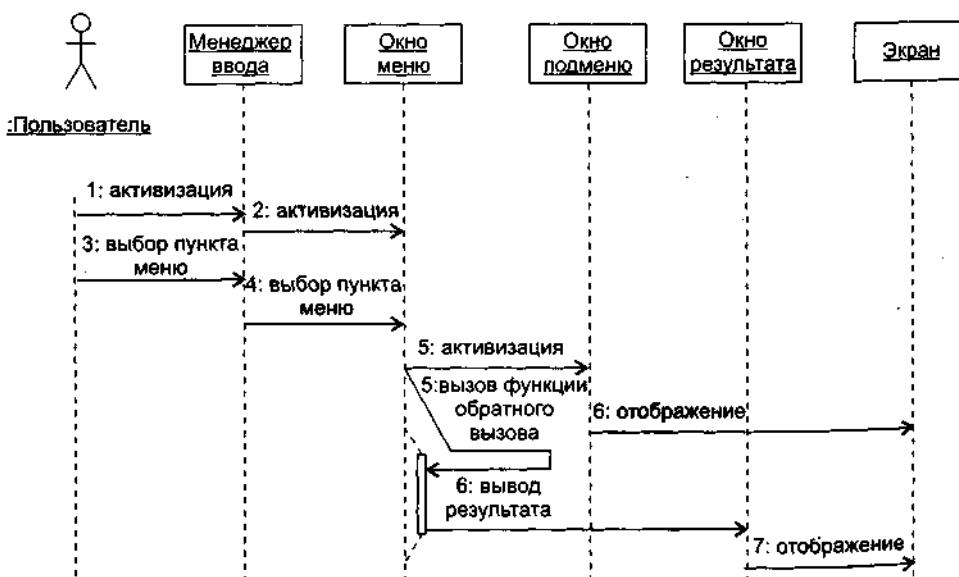


Рис. 15.11. Диаграмма последовательности Использование окна меню

Создание классов

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 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 си операцию add_to_list().

На третьем этапе определяются отношения ассоциации между классами — они обеспечивают пересылки сообщений между соответствующими объектами.

В нашем примере анализ диаграмм последовательности позволяет выделить следующие классы:

- Window — класс, объектами которого являются простые окна;
- Menu — класс, объектами которого являются окна меню. Этот класс является потомком класса Window;
- 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.

Для оптимизации ресурсов создается абстрактный суперкласс Root_Window. Он определяет минимальные обязанности, которые должен реализовать любой тип окна (а (посылка символа в окно, перевод окна в активное/пассивное состояние, перерисовка окна, возврат информации об окне). Все остальные классы окон являются его потомками.

Для реализации функций, определенных в сценариях, в классы добавляются свойства и операции. По результатам формирования свойств и операций классов обновляется содержание диаграмм последовательности.

Начальное представление иерархии классов WUI показано на рис. 15.12. Результаты начальной оценки качества проекта сведены в табл. 15.2.

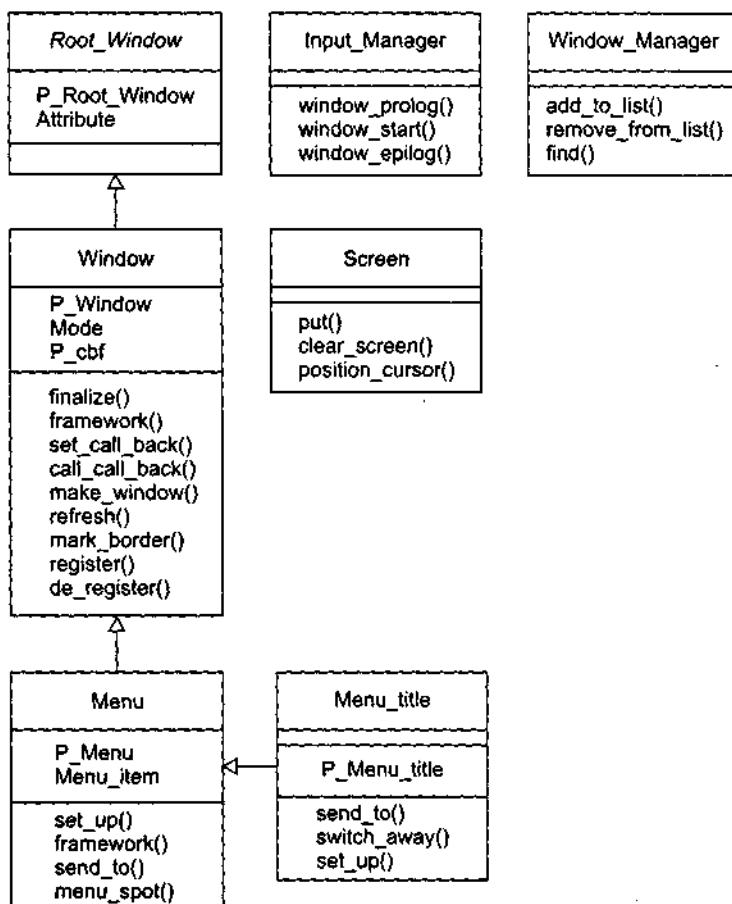


Рис. 15.12. Начальная диаграмма классов WUI

Таблица 15.2. 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							

Отметим, что для упрощения рисунка на этой диаграмме не показаны существующие между классами отношения ассоциации. В реальной диаграмме они обязательно отображаются — без них экземпляры классов не смогут взаимодействовать друг с другом.

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:

4. Использование простого окна.
5. 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:

- dear_screen — очистка экрана;
- position_cursor — позиционирование курсора;
- put — вывод на экран дисплея строк, символов и чисел.

Результаты оценки качества первой итерации представлены в табл. 15.3.

Таблица 15.3. 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_AVG	4	4,2	2,2	0	4,6	3
NP_AVG	0	1,3	1	0	2,4	0,9
Метрики, вычисляемые для системы						
DIT	1					
NC	5					
MOM	35					
LOC _Σ	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. 15.4.

Таблица 15.4. 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 _{we}	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							
MOM	48							
LOC _Z	223							

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 возрастаания 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.

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. 15.13.

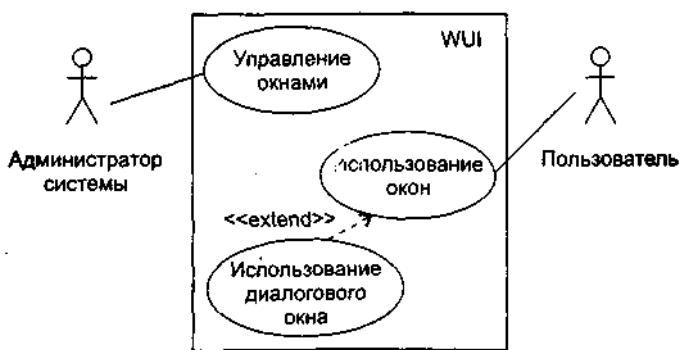


Рис. 15.13. Модифицированная диаграмма Use Case для WUI

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 обратного 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. 15.14.

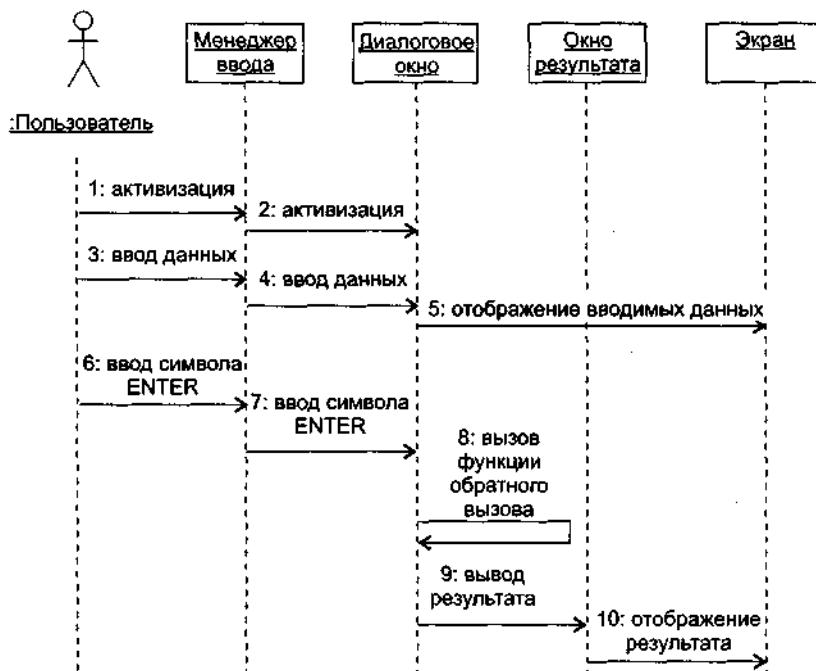


Рис. 15.14. Диаграмма последовательности Использование диалогового окна

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, посыпаемый 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. 15.15. Результаты оценки качества проекта (в конце третьей итерации) сведены в табл. 15.5. Динамика изменения значений для метрик класса показана в табл. 15.6.

Таблица 15.5. Оценки качества WUI после третьей итерации

Метрика	Input_Manager	Window_Manager	Screen	Root_Window	Window	Menu	Menu-title	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 15.6. 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
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

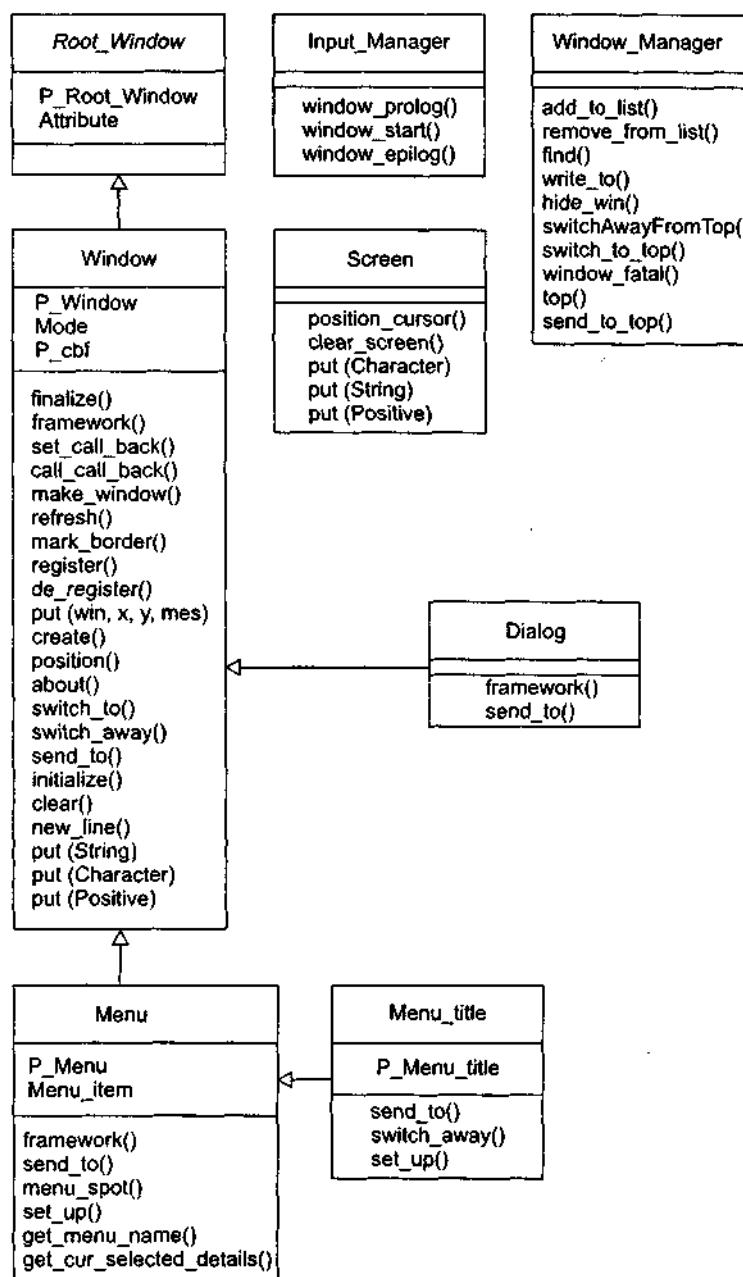


Рис. 15.15. Конечная диаграмма классов WUI

Сравним средние значения метрик второй и третьей итераций:

1. Общая сложность WUI возросла (увеличились значения LOC_Σ, 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).
3. Возросла средняя связность класса (уменьшилось среднее значение метрики LCOM).
4. Уменьшилось среднее значение сцепления класса (сохранилось среднее значение СВО и уменьшилось среднее значение RFC).

Вывод: качество проекта стало выше.

На последней итерации рассчитаны значения интегральных метрик Абреу, они представлены в табл. 15.7. Эти данные также характеризуют качество проекта и подтверждают наши выводы.

Таблица 15.7. 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

Метрические данные проекта помещают в метрический базис фирмы-разработчика, тем самым обеспечивается возможность их использования в последующих разработках.

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 [10], [11], [12], [75]. 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. 15.16.

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 используются для проверки правильности, реализации пользовательских историй.

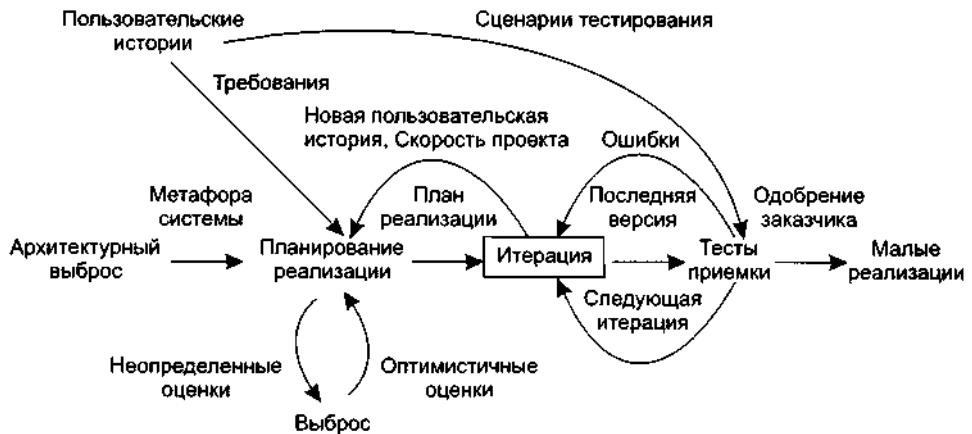


Рис. 15.16. Структура XP-реализации

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 получают 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. 15.17.

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 приспособливаться 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.

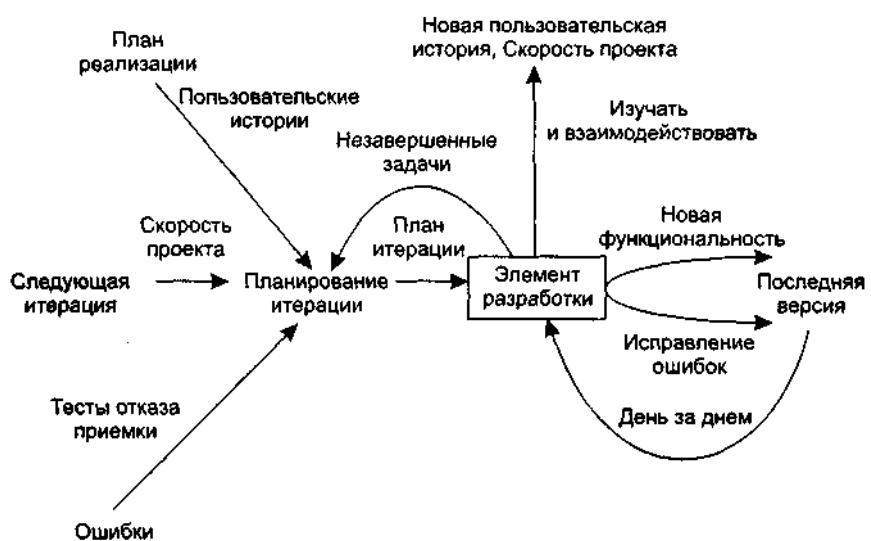


Рис. 15.17. Структура XP-итерации

Для решения каждой из задач требуется от одного до трех дней. Задачи, для которых нужно менее одного дня, группируются вместе, а большие задачи разбиваются 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 каждого 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. 15.18.

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.

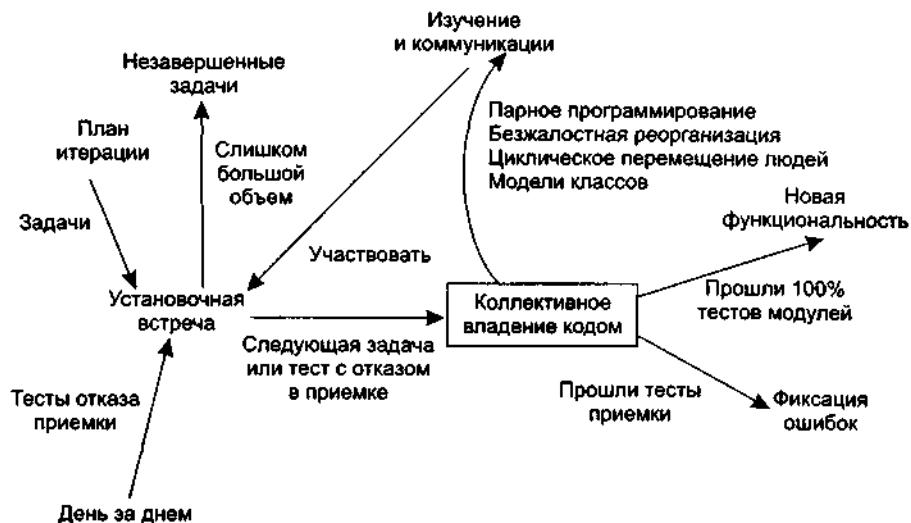


Рис. 15.18. Структура элемента XP-разработки

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. 15.19.

Коллективное владение кодом позволяет каждому разработчику выдвигать новые идеи в любой части проекта, изменять любую строку программы, добавлять функциональность, фиксировать ошибку и проводить реорганизацию. 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.

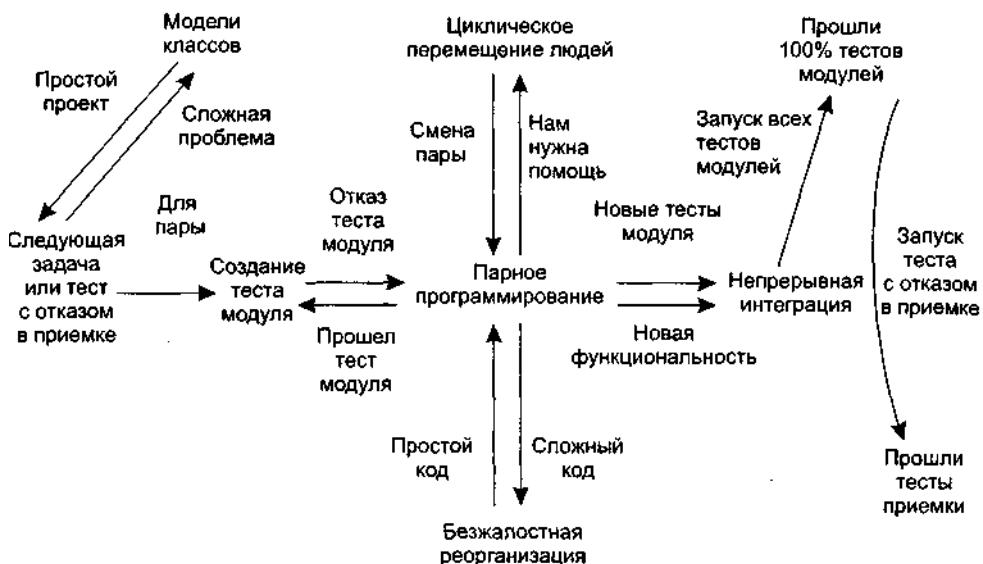


Рис. 15.19. Организация коллективного владения кодом

На основе результатов тестирования разработчики включают в очередную итерацию работу над ошибками. Вообще, следует помнить, что тестирование — один из краеугольных камней ХР.

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 снижению эффективности систем. В ХР считают, что код нужно постоянно обновлять — удалять лишние части, убирать ненужную функциональность. Этот процесс называют реорганизацией кода (refactoring). Поощряется безжалостная реорганизация, сохраняющая простоту проектных решений. Реорганизация поддерживает прозрачность и целостность кода, обеспечивает его легкое понимание, исправление и расширение. На реорганизацию 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 корректировки.

По возможности ХР-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 ХР — постоянное участие заказчика в проведении разработки. По сути, заказчик является одним из разработчиков.

Все этапы ХР 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 конструирования лежит утверждение об экспоненциальной кривой стоимости изменения. Согласно этой кривой, по мере развития проекта стоимость внесения изменений экспоненциально возрастает (рис. 15.20) — 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. 15.21) [25], [30], [36], [37], [62]. Такое сглаживание, с одной стороны, возникает при применении методологии XP, а с другой стороны, 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.

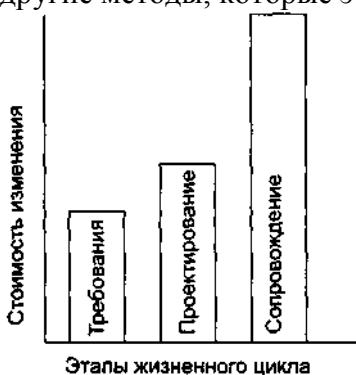


Рис. 15.20. Экспонента стоимости изменения в прогнозирующем процессе

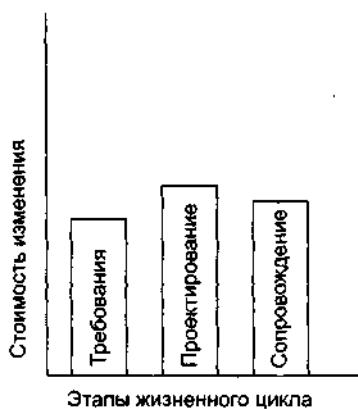


Рис. 15.21. Сглаживание стоимости изменения в адаптивном процессе

К числу основных методов, осуществляющих сглаживание, относят:

- тотальное тестирование;
- непрерывную интеграцию;
- реорганизацию (рефакторинг).

Повторим — надежность кода, которую обеспечивает сквозное тестирование, создает базис успеха, обеспечивает остальные возможности XP-разработки. Непрерывная интеграция необходима для синхронной работы всех сотрудников, так 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 реорганизации (рефакторинга) очень интересно пишет Джим Хайсмит (Jim Highsmith) [37]. Он приводит аналогию с весами (см. рис. 15.22 и 15.23). На одной чаше весов лежит

предварительное проектирование, на другой — реорганизация. В прогнозирующих процессах разработки перевешивает предварительное проектирование, поскольку скорость изменений низкая (на рисунке скорость иллюстрируется положением точки равновесия). В адаптивных, облегченных процессах перевешивает реорганизация, 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. 15.22. Балансировка проектирования и реорганизации при прогнозе

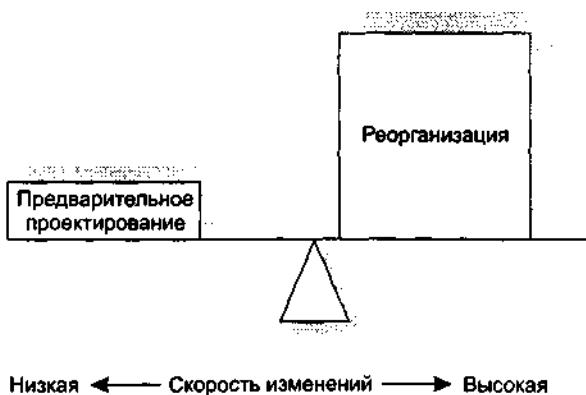


Рис. 15.23. Балансировка проектирования и реорганизации при адаптации

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 оправданно — 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.

Контрольные вопросы

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. В чем суть управления риском?
8. Какие действия определяют управление риском?
9. Какие источники проектного риска вы знаете? 10. Какие источники технического риска вы знаете? И. Какие источники коммерческого риска вы знаете?
12. В чем суть анализа риска?
13. В чем состоит ранжирование риска?
14. В чем состоит планирование управления риском?
15. Что означает разрешение и наблюдение риска? Поясните методику «Отслеживание 10 верхних элементов риска».
16. Дайте характеристику целей, действий и результатов этапа НАЧАЛО.
17. Дайте характеристику целей, действий и результатов этапа РАЗВИТИЕ.
18. Дайте характеристику целей, действий и результатов этапа КОНСТРУИРОВАНИЕ.
19. Дайте характеристику целей, действий и результатов этапа ПЕРЕХОД.
20. Какие метрики используют для оценки качества унифицированного процесса разработки?
21. Охарактеризуйте содержание XP-реализации.
22. В чем разница между пользовательскими историями и обычными требованиями к системе?
23. Что такое выброс?
24. Как создаются тесты приемки?
25. Поясните содержание XP-итерации.
26. В чем заключается планирование XP-итерации?
27. Что такое скорость проекта?
28. Поясните структуру элемента XP-разработки.
29. В чем заключается коллективное владение кодом? Охарактеризуйте содержание такого владения.
30. Как организуется взаимодействие с XP-заказчиком?
31. Прокомментируйте стоимость XP-изменения.
32. Поясните особенности XP-проектирования.

ГЛАВА 16. 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 глав 6, 7 и 8 глав, в которых описывались классические основы тестирования, разработанные в расчете (в основном) на процедурное ПО. В этой главе рассматриваются вопросы объектно-ориентированного тестирования [17], [18], [42], [50], [51]. 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 физическому представлению строятся модели компоновки и размещения системы.

На конструирование 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 [18], [51].

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 в одной части представления, 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 [9], [76]. 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 посыпает сообщения и от которых он зависит при выполнении своих обязанностей). Сотрудничества подразумевают наличие ассоциаций и зависимостей между классами. Они фиксируются в других моделях — диаграмме сотрудничества (последовательности) 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 [50].

1. Выполняется перекрестный просмотр CRC-карты и диаграммы сотрудничества (последовательности) 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. 16.1 показана CRC-карта Банкомат. Для этой карты выясняем, выполняется ли обязанность Читать карту клиента, которая требует использования сотрудника Карта клиента. Это означает, что класс Карта клиента должен иметь операцию, которая позволяет ему быть прочитанным.

Таблица 16.1. CRC-карта Банкомат

Имя класса: Банкомат	
Обязанности:	Сотрудники:
Читать карту 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, или 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 — Читать карту 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 testируют модули, затем testируют интеграцию модулей, проверяют правильность реализации требований, после чего testируют взаимодействие всех блоков компьютерной системы.

Особенности тестирования объектно-ориентированных «модулей»

При рассмотрении объектно-ориентированного ПО меняется понятие модуля. Наименьшим testируемым элементом теперь является класс (объект). Класс содержит несколько операций и свойств. Поэтому сильно изменяется содержание testирования модулей.

В данном случае нельзя testировать отдельную операцию изолированно, как это принято в стандартном подходе к testированию 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 свойств и операций. Этот контекст меняется, поэтому операцию OP() надо testировать в контексте каждого подкласса. Таким образом, изолированное testирование операции OP(), являющееся традиционным подходом в testировании 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 неосуществим.

Р. Байндер предлагает две методики интеграции объектно-ориентированных систем [16]:

- 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 [50]. Кластер сотрудничающих классов определяется исследованием CRC-модели или диаграммы сотрудничества 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 схем состояний.

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 свойств 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 `переопределен()`. К чему это приводит? Поскольку реализация операции `переопределен()` 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 `переопределен()`, 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()`.

Выходы:

- 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 [46]. Разработчик выдвигает гипотезу о предполагаемых ошибках. Для проверки его предположений разрабатываются тестовые варианты.

В качестве примера рассмотрим булево выражение

$$\text{if } (X \text{ and not } Y \text{ or } Z).$$

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 ($X = \text{false}$, $Y = \text{false}$, $Z = \text{false}$) 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 свойствам, так и к операциям. Поведение объекта определяется значениями, которые получают его свойства. Поэтому проверка значений свойств 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, чего 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 [47]. 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, основанных на сценариях, для текстового редактора.

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. Данный 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 сотрудничество, 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 [43].

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 операцию 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 операцию 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 [43]. 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 свойствам*. Основывается на свойствах, которые используются операциями. В классе Счет для определения разбиений можно использовать свойства остаток и ограничение кредита. Например, на основе свойства 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 [43]:

- 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();
ПроверитьPIN();
ПроверитьПолис();
ЗапросСнятия();

ЗапросДепозита();
ИнфоСчета();
ОткрытьСчет();
НачальнДепозит();

РазрешитьКарту();
СнятьРазрешен();
ЗакрытьСчет();

Банкомат:

КартаВставлена();
Пароль();

Положить();
Снять();

СостояниеСчета();
Завершить();

ИнтерфейсБанкомата:

ПроверитьСостояние();
СостояниеПоложить();

ВыдатьНаличные();
ПечатьСостСчета();

ЧитатьИнфоКарты();
ПолучитьКолвоНалич();

Счет:

ОграничКредит();
ТипСчета();

Остаток();
Снять();

Положить();
Закрыть();

ПодтверждениеПравильности:

ПодтвPIN();

ПодтвСчет();

Диаграмма сотрудничества объектов банковской системы представлена на рис. 16.1. На этой диаграмме отображены связи между объектами, стрелки передачи сообщений подписаны именами вызываемых операций.

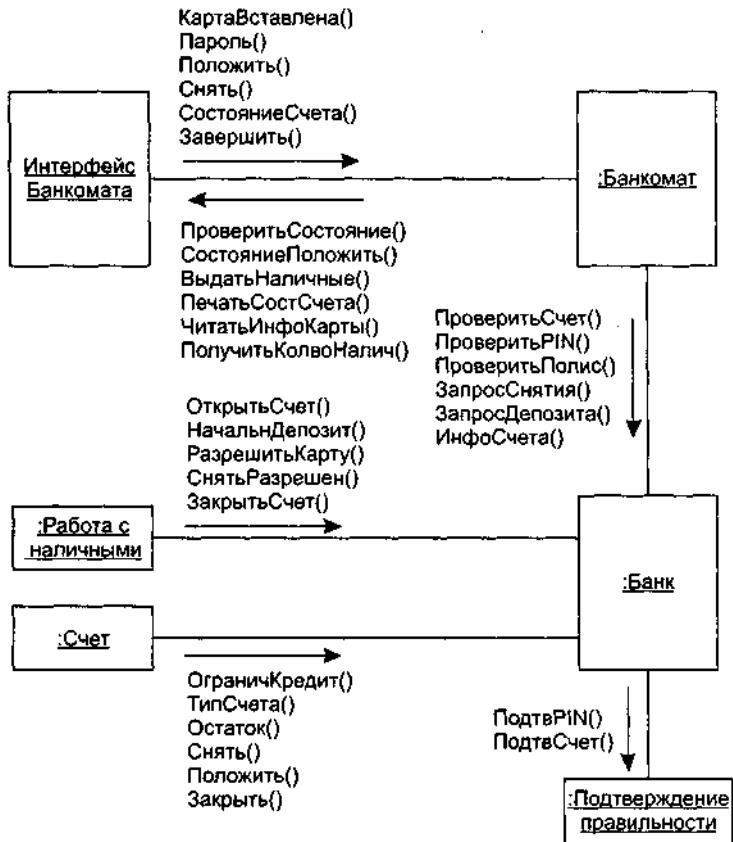


Рис. 16.1. Диаграмма сотрудничества банковской системы

Стохастическое тестирование

Стохастические тестовые варианты генерируются следующей последовательностью шагов.

1. Для создания тестов используют списки операций каждого класса-клиента. Операции будут посыпать 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 операцию следования, точка — операцию 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 ПроверитьРТМ 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}
► (ПодтвРШ_{ПодтвПрав}) ► ЗапросДепозита_{Банк} ► (Положить_{Счет}).

В этой последовательности операции классов-сотрудников Банка помещены в круглые 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. 16.1, 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 схем состояний, фиксирующие динамику поведения класса. Данный способ позволяет получить набор тестов, проверяющих поведение класса и тех классов, которые сотрудничают с ним [43]. В качестве примера на рис. 16.2 показана диаграмма схем состояний класса Счет.

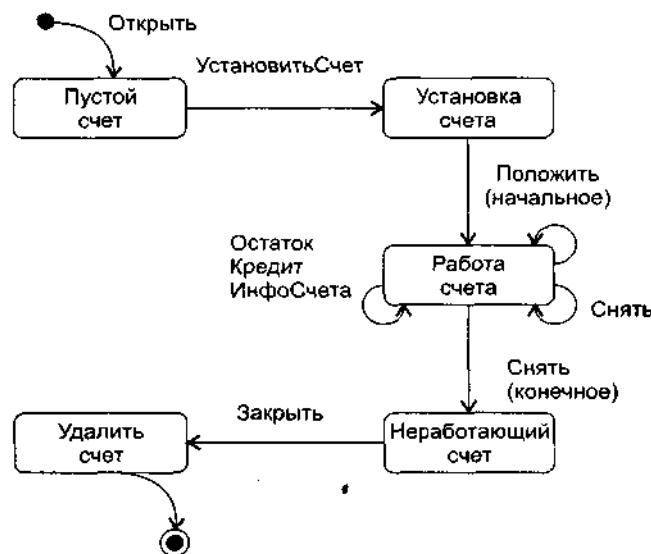


Рис. 16.2. Диаграмма схем состояний класса Счет

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 схем состояний, 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 Кarta клиента (рис. 16.3). 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. Кarta клиента переходит в состояние Предъявлется 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.

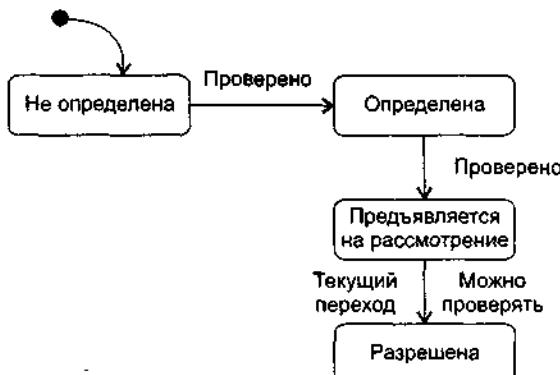


Рис. 16.3. Тестирование «преимущественно в ширину»

Подход «преимущественно в ширину» требует: нельзя проверять Разрешена перед проверкой Не определена, Определена и Предъявлется на рассмотрение. В противном случае нарушается условие этого подхода: перед тестированием текущего перехода должны быть протестированы все переходы, ведущие к нему.

Предваряющее тестирование при экстремальной разработке

Предваряющее тестирование и рефакторинг (реорганизация) — основной способ разработки при экстремальном программировании.

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 (test-first) тестирование и рефакторинг — это способ 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 выполнять 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 (с любезного разрешения автора)*.

* Robert C. Martin. RUP/XP Guidelines: Test-first Design and Refactoring. - Rational Software White Paper, 2000.

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 16.1. 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 ПосещениеКафе, содержащий нужные данные. Следовательно, надо написать тест, создающий этот объект и опрашивающий его свойства. Тесты будем записывать как тестовые функции (их имена должны начинаться с префикса тест). Введем тестовую функцию тестСоздатьПосещениеКафе (листинг 16.2).

Листинг 16.2. ТестЛакомки.java

```
import junit.framework.*;
public class ТестЛакомки extends TestCase
{
    public ТестЛакомки (String name)
    {
        super(name);
    }
    public void тестСоздатьПосещениеКафе()
    {
        ПосещениеКафе v = new ПосещениеКафе();
    }
}
```

Для компиляции этого фрагмента подключим класс ПосещениеКафе.

Листинг 16.3. 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 16.4. 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.получитьСтоймость(), 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 егоСтоймость;
    private double ероBес;
    public ПосещениеКафе(Date дата, double булочки,
        double стоимость, double вес)
    {
        егодата = дата;
        егоБулочки = булочки;
        егоСтоймость = стоимость;
        ероBес = вес;
    }
    public Date получитьДату() {return егодата;}
    public double получитьБулочки() {return егоБулочки;}
    public double получитьСтоймость() {return егоСтоймость;}
    public double получитьЦену(){return егоСтоймость/егоБулочки;}
    public double получитьВес() {return ероBес;}
}

```

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 свойство ероBес характеризует лакомку. Таким образом, объект ПосещениеКафе записывает часть состояния лакомки па

момент посещения кафе. Следовательно, нужно создать объект Лакомка и содержать объекты класса ПосещениеКафе в нем.

Листинг 16.5. ТестЛакомки.java и Лакомка.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 тестСоздатьЛакомку()
    {
        Лакомка g = new Лакомка();
        assertEquals(0, д.получитьЧислоПосещений());
    }
}
Лакомка.java
public class Лакомка
{
    public int получитьЧислоПосещений()
    {
        return 0;
    }
}
```

Листинг 16.5 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 16.6. 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 прогонаем 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 16.7. 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 объекта Лакомка.

Листинг 16.8. Лакомка.jaya

```

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 свойствами класса.

Листинг 16.9. ТестЛакомки.jaya

```

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);
        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.получитьЧислоПосещениеПК));
    }
}

```

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 ПосещениеКафе у Лакомки можно запрашивать генерацию отчетов. Сначала напишем тесты, начнем с простейшего теста.

Листинг 16.10. 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 16.11. 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.г.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 получитьЛотреблениеБулочек()
    {return егоПотреблениеБулочек;}
private double егоИзменениеВеса;
private double егоВесНаБулочку;
private double егоСтоимостьБулочек;
private double егоПотреблениеБулочек;
}

```

Код в листинге 16.11 компилируется и запускается, но его недостаточно для того, чтобы прошли тесты. Нужен рефакторинг кода. Для начала сделаем минимально возможные изменения.

Листинг 16.12. 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());
    r.устПотреблениеБулочек(v.получитьБулочки());
    return r;
}
Отчет.java
public void устВесНаБулочку (double wpb)
    {егоВесНаБулочку = wpb;}
public void устИзменениеBec(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 свойств 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 16.13. 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 16.14. ТестЛакомки.jaya

```

public void тестОтчетаНесколькихПосещений()
{
    Лакомка g = new Лакомка();
    g.добавить(ПосещениеКафе(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 равным 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 16.15. ТестЛакомки.jaya

```

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 16.15.

Листинг 16.16. 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 общаяСтоиность = 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 1-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
        общаяСтоиность += v.получитьСтоиность();
        потреблениеБулочек += v.получитьБулочки();
    }
    double изменение = последнийЗамер - первыйЗамер;
    r.устВесНаБулочкуСизменение/потреблениеБулочек);
    r.устИзменениеВеса(изменение);
    r.устСтоиностьБулочек(общаяСтоиность);
    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 16.17. 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.устПотреблениеБулочекСпотреблениеБулочек);
    return r;
}
```

Теперь попытаемся сделать функцию короче и понятнее. Переместим фрагменты кода так, чтобы их можно было вынести в отдельные функции.

Листинг 16.18. 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();
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 16.18 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 16.19. 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 16.20. Лакомка.java

```

public Отчет создатьОтчет()
{
    Отчет г = 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()
{
    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 16.21. 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 - вычОбщуюС тонкость ();
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 16.22. 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 16.23. Лакомка.java

```

public Отчет создатьОтчет()
{
    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

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 свойствам.
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. Поясните суть предваряющего тестирования.
19. Какую роль в процессе экстремальной разработки играет рефакторинг?

ГЛАВА 17. АВТОМАТИЗАЦИЯ КОНСТРУИРОВАНИЯ ВИЗУАЛЬНОЙ МОДЕЛИ ПРОГРАММНОЙ СИСТЕМЫ

В современных условиях создание сложных программных приложений невозможно без использования систем автоматизированного конструирования 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 известной объектно-ориентированной CASE-системой является Rational Rose. В данной главе

рассматривается порядок применения Rational Rose при формировании требований, анализе, проектировании и генерации программного кода.

Общая характеристика CASE-системы Rational Rose

Rational Rose — это CASE-система для визуального моделирования объектно-ориентированных программных продуктов. Визуальное моделирование — процесс графического описания разрабатываемого программного обеспечения. Экран среды Rational Rose показан на рис. 17.1.

В его составе выделим шесть элементов: строку инструментов, панель «инструменты диаграммы», окно диаграммы, браузер, окно спецификации, окно документации.

Как показано на рис. 17.2, кнопки строки инструментов позволяют выполнять стандартные и специальные действия.

Содержание панели инструментов диаграммы меняется в зависимости от активной диаграммы. Окно активной диаграммы имеет синюю строку заголовка (рис. 17.3).

В окне диаграммы можно создавать, отображать и изменять диаграмму на языке UML.

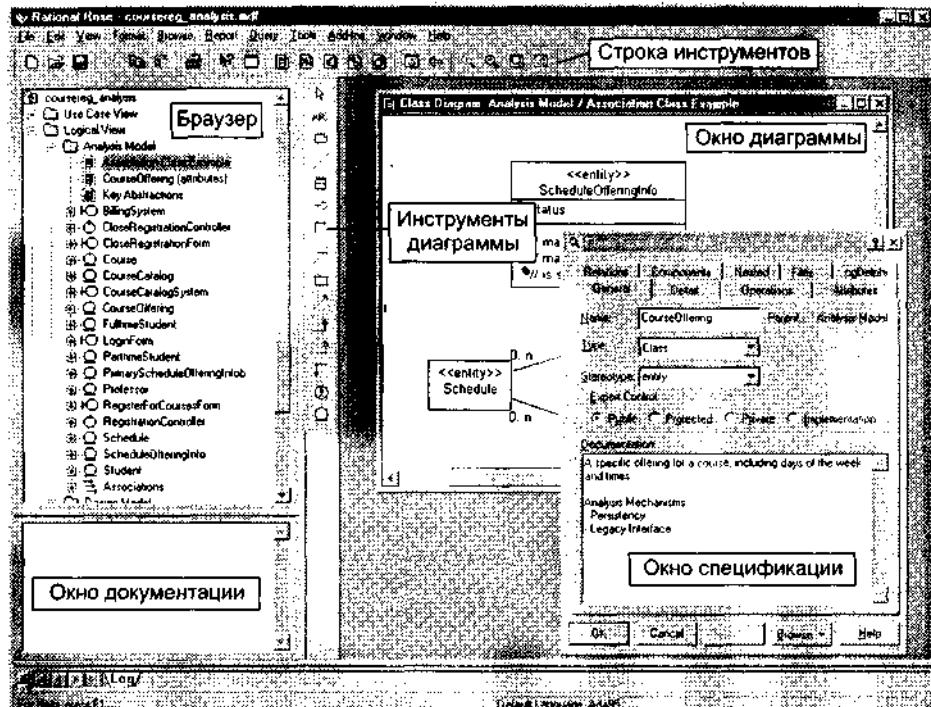


Рис. 17.1. Экран среды Rational Rose

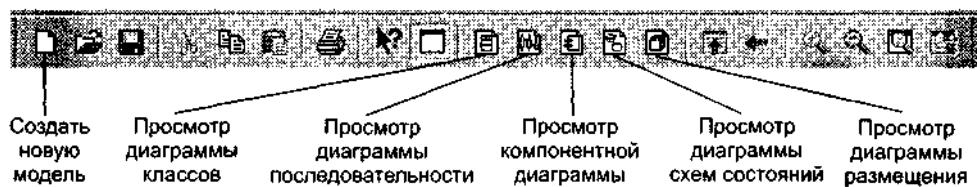


Рис. 17.2. Кнопки строки инструментов Rational Rose

Браузер Rational Rose является инструментом иерархической навигации, позволяющим просматривать названия и пиктограммы, отображающие диаграммы и элементы визуальной модели (рис. 17.4).

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.

Окно спецификации позволяет задавать характеристики элемента диаграммы (рис. 17.5).

В поле Documentation этого окна вводится словесное описание данного элемента. Это же описание можно вводить в Окно документации Rational Rose (когда данный элемент выделен в диаграмме).

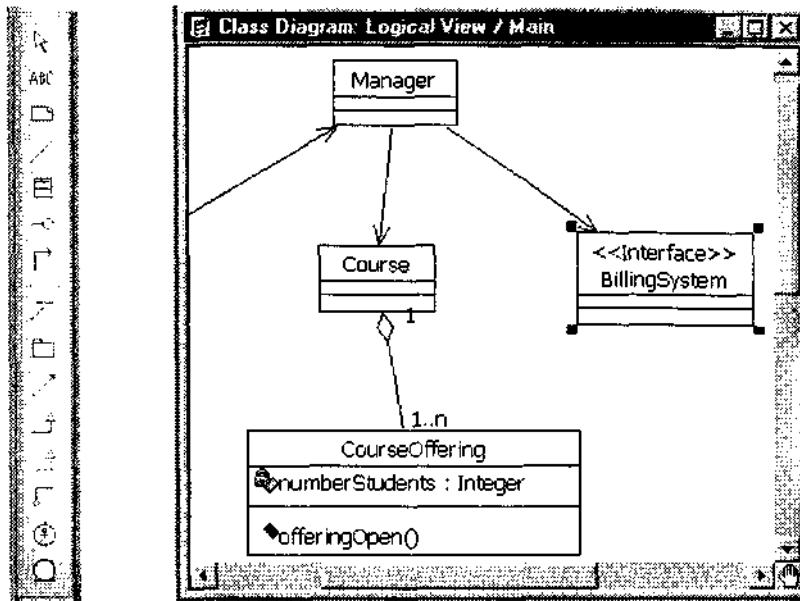


Рис. 17.3. Панель инструментов и окно активной диаграммы

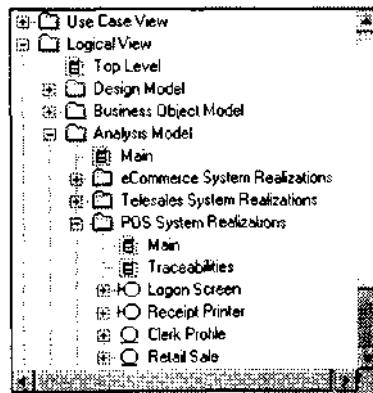


Рис. 17.4. Браузер Rational Rose

В качестве примера работы с Rational Rose рассмотрим построение модели университетской системы для регистрации учебных курсов (классический пример компании Rational), автор которой — Терри Кватрани [57].

Эта система используется:

- профессором — для задания читаемого курса;
- студентом — для выбора изучаемого курса;
- регистратором — для формирования учебного плана и расписания;
- учетной системой — для определения денежных затрат.

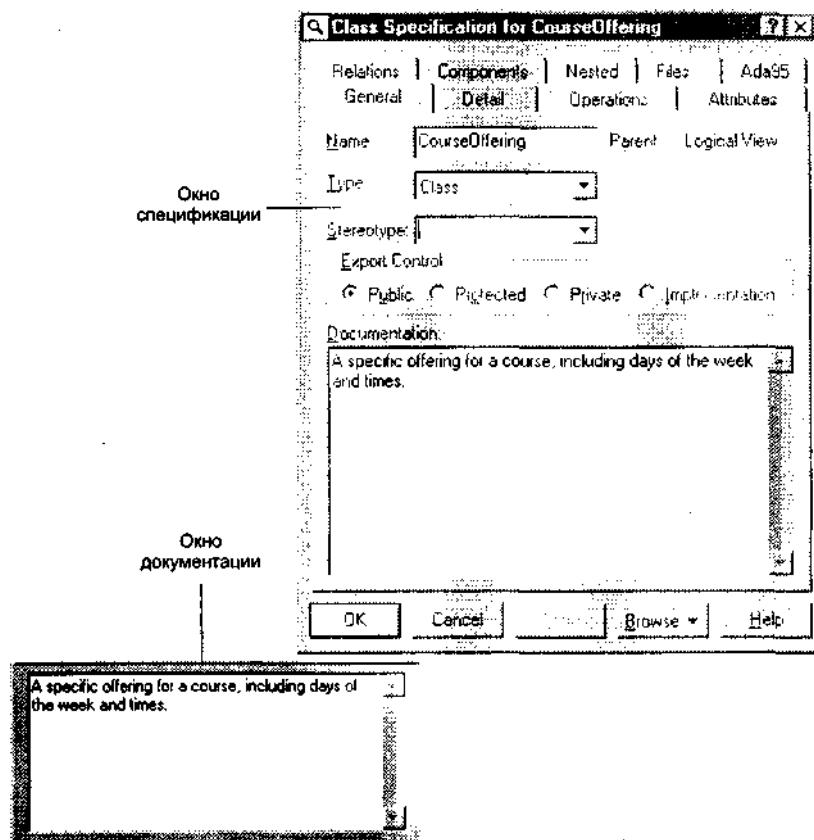


Рис. 17.5. Окно спецификации и окно документации Rational Rose

Создание диаграммы Use Case

Моделирование проблемы регистрации курсов начнем с создания диаграммы Use Case. Этот тип диаграммы представляется актерами, элементами Use Case и отношениями между ними. Откроем главную диаграмму Use Case (рис. 17.6).

1. В окне браузера щелкнем по значку + слева от пакета Use Case View.

2. Для открытия диаграммы выполним двойной щелчок по значку Main.

Первый шаг построения этой диаграммы состоит в определении актеров, фиксирующих роли внешних объектов, взаимодействующих с системой. В нашей проблемной области можно выделить 4 актера — Student (Студент), Professor (Преподаватель), Registrar (Регистратор) и Billing System (Учетная система) (рис. 17.7).

1. На панели инструментов щелкните по значку актера.

2. Для добавления актера в диаграмму щелкните в нужном месте диаграммы.

3. Пока актер остается выделенным, введите имя Student (Студент).

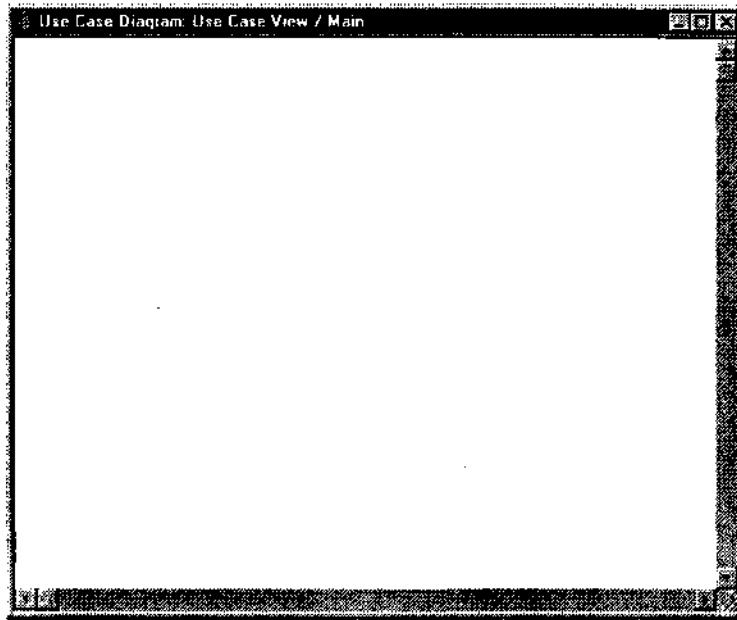


Рис. 17.6. Главная диаграмма Use Case

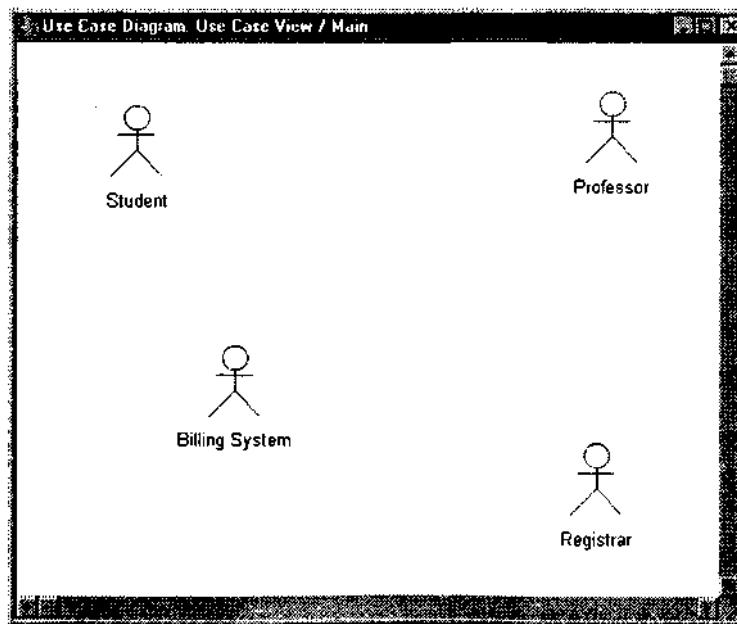


Рис. 17.7. Четыре актера

4. Повторите предыдущие шаги для ввода трех других актеров (Professor, Registrar и Billing System — Профессор, Регистратор, Учетная система).

Далее для каждого актера нужно определить соответствующие элементы Use Case. Элемент Use Case представляет определенную часть функциональности, обеспечиваемой системой. Вы можете идентифицировать элементы Use Case путем рассмотрения каждого актера и его взаимодействия с системой. В нашей модели актер Student хочет зарегистрироваться на курсы (Register for Courses). Актер Billing System получает информацию о регистрации. Актер Professor хочет запросить список курса (Request a Course Roster). Наконец, актер Registrar должен управлять учебным планом (Manage Curriculum) (рис. 17.8).

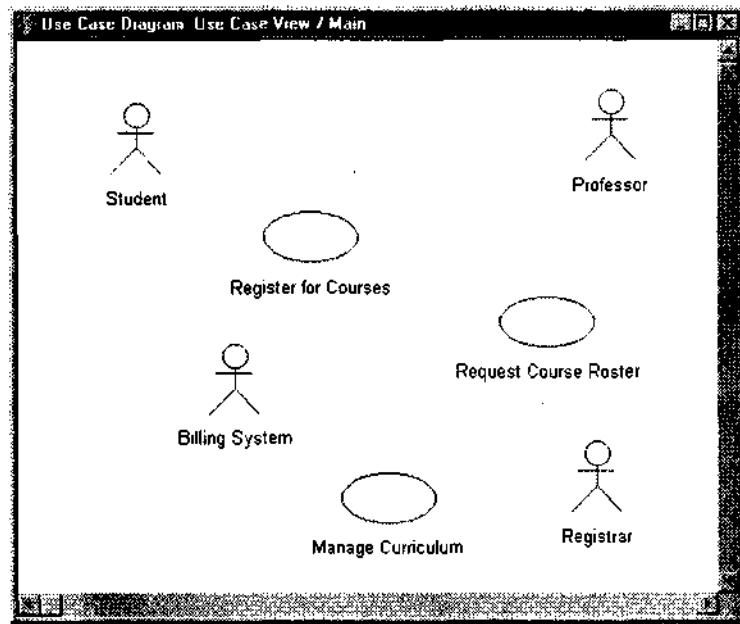


Рис. 17.8. Элементы Use Case для актеров

1. На панели инструментов щелкните по значку элемента Use Case.
2. Для добавления элемента Use Case в диаграмму щелкните в нужном месте диаграммы.
3. Пока элемент Use Case остается выделенным, введите имя Register for Courses.
4. Повторите предыдущие шаги для ввода других элементов Use Case (Request Course Roster, Manage Curriculum).

Далее между актерами и элементами Use Case рисуются отношения. Чтобы показать направление взаимодействия (кто инициирует взаимодействие), используются односторонние стрелки (unidirectional arrows). В системе регистрации курсов актер Student инициирует элемент Use Case Register for Courses, который, в свою очередь, взаимодействует с актером Billing System. Актер Professor инициирует элемент Use Case Request Course Roster. Актер Registrar инициирует элемент Use Case Manage Curriculum (рис. 17.9).

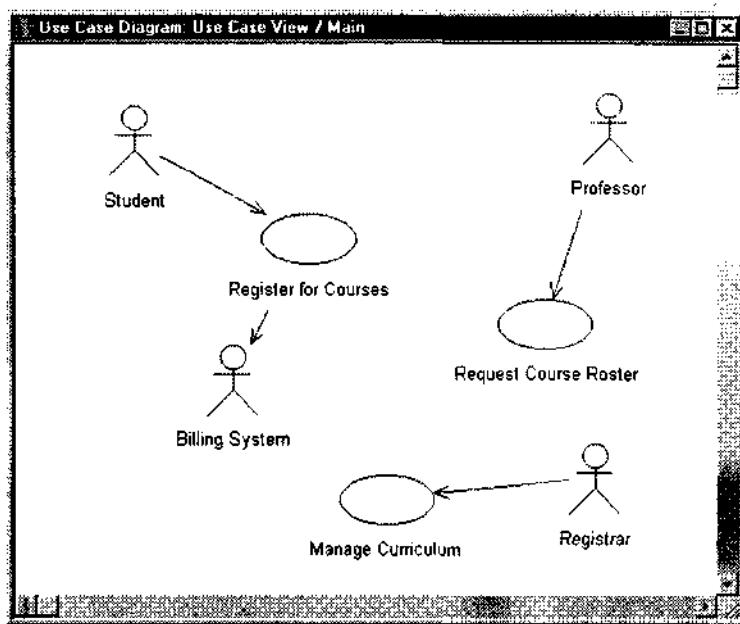


Рис. 17.9. Отношения между актерами и элементами Use Case

1. На панели инструментов щелкните по значку однонаправленной ассоциации (стрелке).
2. Щелкните по актеру Student и перетащите линию на элемент Use Case Register for Courses.
3. На панели инструментов щелкните по значку однонаправленной ассоциации (стрелке).
4. Щелкните по элементу Use Case Register for Courses и перетащите линию на актера Billing System.
5. Повторите предыдущие шаги для ввода других отношений (от актера Professor к элементу Use Case Request Course Roster и от актера Registrar к элементу Use Case Manage Curriculum).

Case Request Course Roster и от актера Registrar к элементу Use Case Manage Curriculum).

Создание диаграммы последовательности

Функциональность элемента Use Case отображается графически в диаграмме последовательности (Sequence Diagram). Эта диаграмма отображает один из возможных путей в потоках событий элемента Use Case — например, добавление студента к курсу. Диаграммы последовательности содержат объекты и сообщения между объектами, которые показывают реализацию поведения. Рассмотрим диаграмму последовательности Add a Course для элемента Use Case Register for Courses (рис. 17.10).

1. В окне браузера щелкните правой кнопкой по элементу Use Case Register for Courses.
2. В появившемся контекстном меню выберите команду New:Sequence Diagram.
3. В результате в дерево окна браузера будет добавлена диаграмма последовательности с именем New Diagram.
4. Пока значок новой диаграммы остается выделенным, введите имя Add a Course.

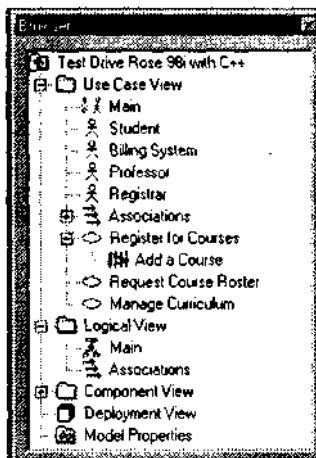


Рис. 17.10. Создание диаграммы последовательности Add a Course

Теперь мы будем добавлять в диаграмму такие объекты и сообщения, которые реализуют необходимую функциональность. Для открытия диаграммы два раза щелкнем по ее значку в окне браузера. Поскольку сценарий инициируется актером Student, перетащим этого актера в диаграмму (рис. 17.11). Экземпляру актера можно присвоить конкретное имя. Назовем нашего студента Joe.

1. Для открытия диаграммы последовательности выполним двойной щелчок по ее значку в окне браузера.
2. Щелкнем по значку актера Student в браузере и перетащим его в диаграмму последовательности.
3. Щелкнем по значку актера в диаграмме последовательности и введем его имя — Joe.

Сценарий, который мы собираемся формализовать с помощью диаграммы последовательности, уже существует — он является фрагментом текста, который содержит спецификация элемента Use Case Register for Courses.

В этом сценарии студент должен заполнить информацией (fill in info) регистрационную форму (registration form), а затем форма предъявляется на рассмотрение (submitted). Очевидно, что необходим объект registration form, который принимает информацию от студента (рис. 17.12). Создадим форму и добавим два сообщения, «fillin info» и «submit».

1. На панели инструментов щелкните по значку объекта (прямоугольнику).
2. Для добавления объекта в диаграмму щелкните в нужном месте диаграммы.
3. Пока объект остается выделенным, введите имя registration form.
4. На панели инструментов щелкните по значку объектного сообщения (стрелке).
5. Щелкните по пунктирной линии под актером Student и перетащите стрелку на пунктирную линию под объектом registration form.
6. Пока стрелка остается выделенной, введите имя сообщения — fill in information.
7. Повторите шаги 4-6 для создания сообщения submit.

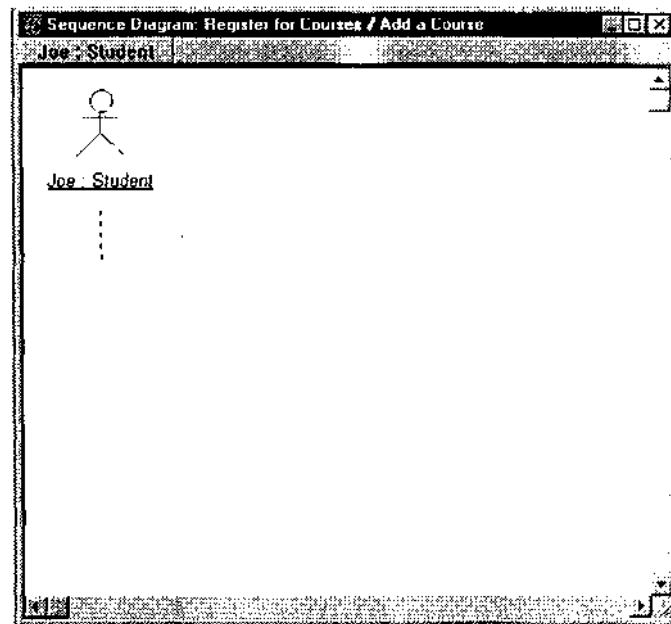


Рис. 17.11. Диаграмма последовательности — Joe

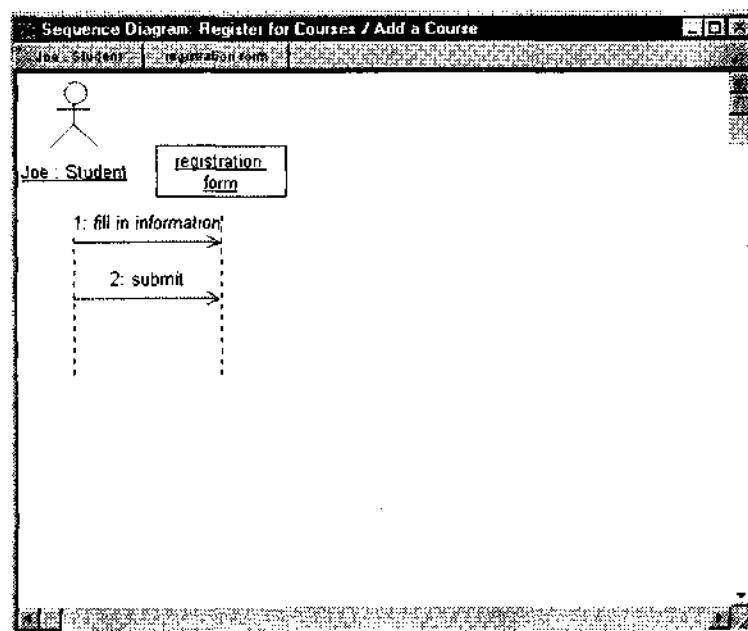


Рис. 17.12. Диаграмма последовательности — Registration Form

Очевидно, что объект регистрационная форма является промежуточным звеном в цепи передачи информации. Следующее звено — объект-менеджер. Его надо добавить в диаграмму.

Далее форма должна послать сообщение менеджеру (manager) (рис. 17.13). Таким образом менеджер узнает, что студент должен быть добавлен к курсу, — положим, что Joe хочет получить курс math 101.

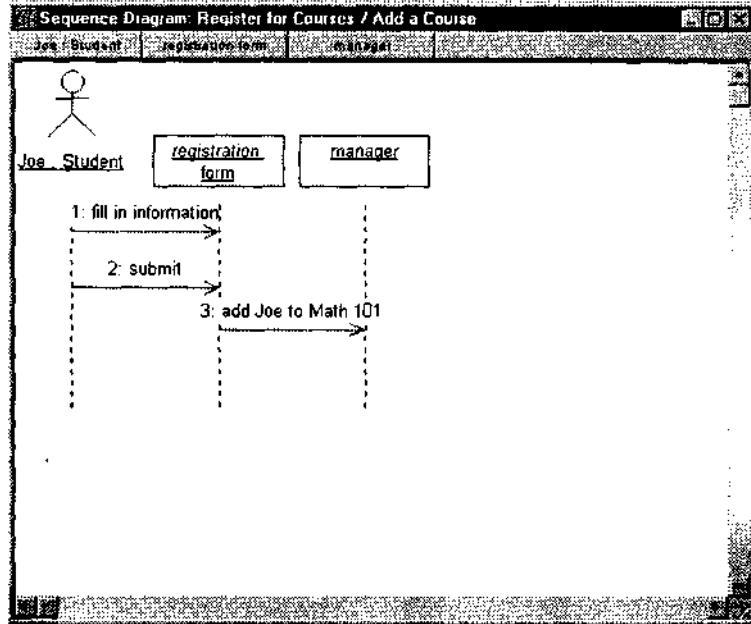


Рис. 17.13. Диаграмма последовательности — Manager

1. На панели инструментов щелкните по значку объекта (прямоугольнику).
2. Для добавления объекта в диаграмму щелкните в нужном месте диаграммы.
3. Пока объект остается выделенным, введите имя manager.
4. На панели инструментов щелкните по значку объектного сообщения (стрелке).
5. Щелкните по пунктирной линии под объектом registration form и перетащите стрелку на пунктирную линию под объектом manager.
6. Пока стрелка остается выделенной, введите имя сообщения — add Joe to Math 101.

Менеджер — один из объектов системы, который взаимодействует как с регистрационной формой, так и с набором объектов — учебных курсов. Для обслуживания нашего студента должен быть объект Math 101. Если такой объект существует, то Менеджер обязан передать объекту-курсу Math 101, что Joe должен быть добавлен к курсу (рис. 17.14).

1. На панели инструментов щелкните по значку объекта (прямоугольнику).
2. Для добавления объекта в диаграмму щелкните в нужном месте диаграммы.
3. Пока объект остается выделенным, введите имя math 101.
4. На панели инструментов щелкните по значку объектного сообщения (стрелке).
5. Щелкните по пунктирной линии под объектом manager и перетащите стрелку на пунктирную линию под объектом math 101.
6. Пока стрелка остается выделенной, введите имя сообщения — add Joe.

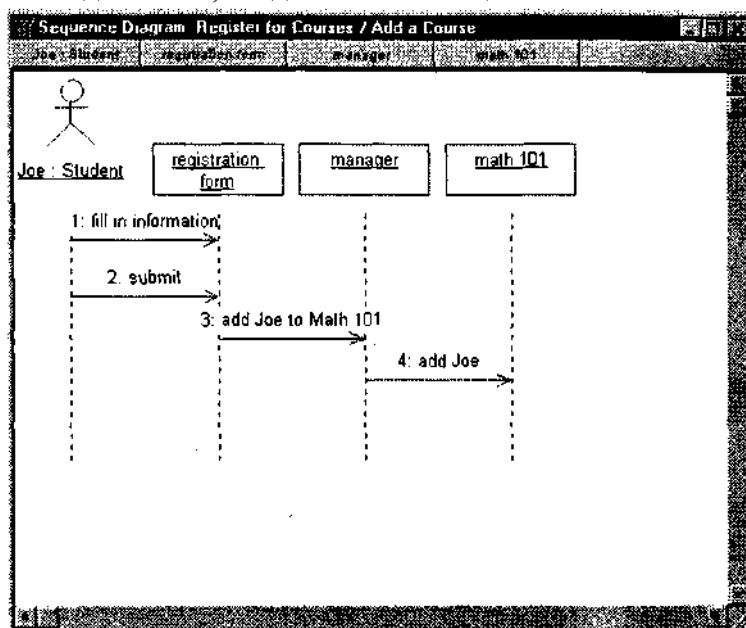


Рис. 17.14. Диаграмма последовательности — Math 101

Объект-курс не принимает самостоятельных решений о возможности добавления студентов. Этим занимается экземпляр класса Предложение курса (course offering). Назовем такой экземпляр (объект) Section 1.

Объект-курс обращается к объекту Section 1, если он открыт (в этом сценарии — ответ «да») с предложением добавить студента Joe (рис. 17.15).

1. На панели инструментов щелкните по значку объекта (прямоугольнику).
2. Для добавления объекта в диаграмму щелкните в нужном месте диаграммы.
3. Пока объект остается выделенным, введите имя — section 1.
4. На панели инструментов щелкните по значку объектного сообщения (стрелке).
5. Щелкните по пунктирной линии под объектом math 101 и перетащите стрелку на пунктирную линию под объектом section 1.
6. Пока стрелка остается выделенной, введите имя сообщения — accepting students?
7. Повторите шаги 4-6 для создания сообщения add Joe.

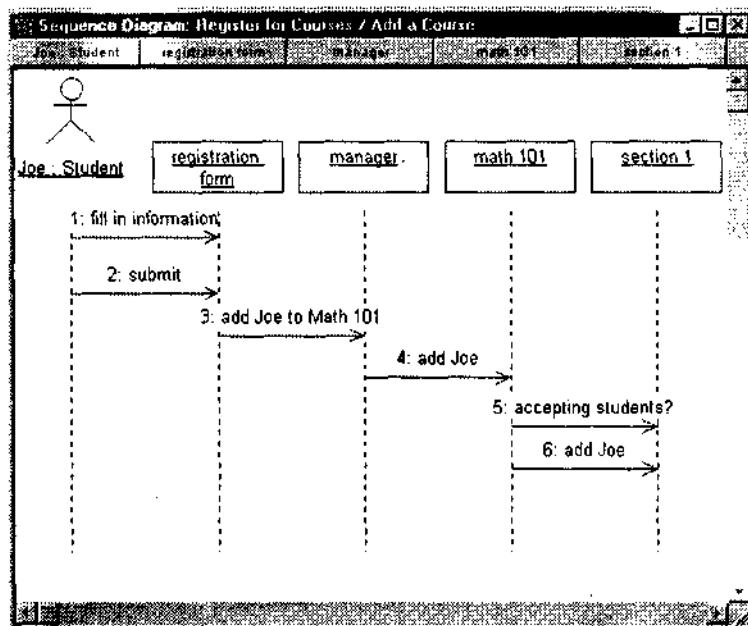


Рис. 17.15. Диаграмма последовательности — Section 1

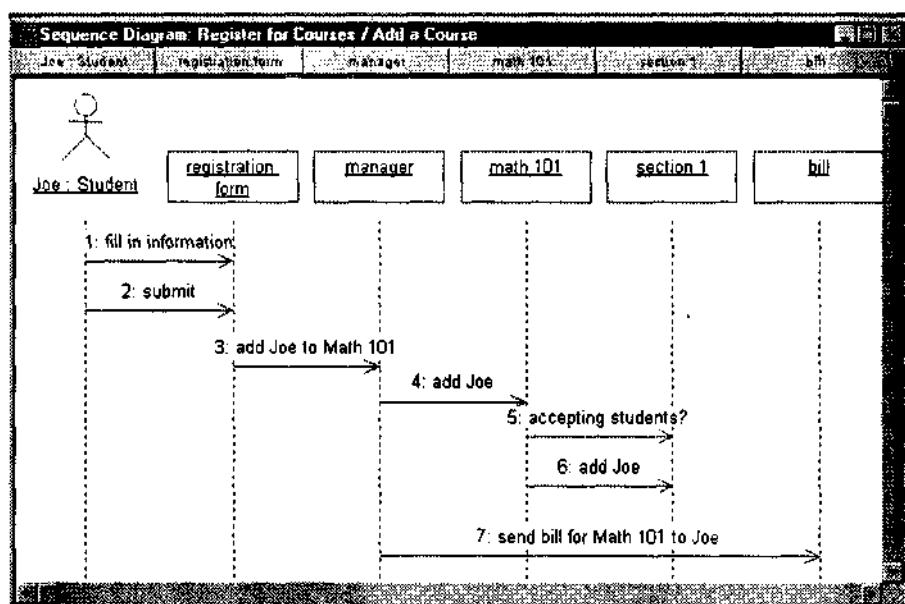


Рис. 17.16. Диаграмма последовательности — Billing System

Конечно, за учебу надо платить. Вопросами оплаты занимается Учетная система, а уведомляет ее о необходимости выписки счета менеджер. После того как менеджер удостоверился, что студенту Joe

предоставляется возможность изучать курс math 101 (на диаграмме рис. 17.16 это не показано), уведомляется Учетная система (billing system).

1. На панели инструментов щелкните по значку объекта (прямоугольнику).
2. Для добавления объекта в диаграмму щелкните в нужном месте диаграммы.
3. Пока объект остается выделенным, введите имя — bill.
4. На панели инструментов щелкните по значку объектного сообщения (стрелке).
5. Щелкните по пунктирной линии под объектом manager и перетащите стрелку на пунктирную линию под объектом bill.
6. Пока стрелка остается выделенной, введите имя сообщения — send bill for Math 101 to Joe.

Создание диаграммы классов

Объекты из диаграмм последовательности группируются в классы. Основываясь на нашей диаграмме последовательности, мы можем идентифицировать следующие объекты и классы:

- ❑ registration form является объектом класса RegForm;
- ❑ manager является объектом класса Manager;
- ❑ math 101 является объектом класса Course;
- ❑ section 1 является объектом класса CourseOffering;
- ❑ bill является интерфейсом к внешней учетной системе, поэтому мы будем использовать имя BillingSystem как имя его класса.

Классы создаются в логическом представлении системы (рис. 17.17).

1. В окне браузера щелкните правой кнопкой по значку пакета Logical View.
2. В появившемся контекстном меню выберите команду New:Class. В результате в дерево окна браузера будет добавлен класс с именем NewClass.
3. Пока значок класса остается выделенным, введите имя RegForm.
4. Повторите предыдущие шаги для добавления других классов: Manager, Course, CourseOffering и BillingSystem.

После создания классов они описываются (документируются). Описания добавляются с помощью Documentation Window (рис. 17.18).

1. В окне браузера щелкните по значку класса CourseOffering.
2. Введите описание класса в Documentation Window.

Процесс построения сценариев и нахождения классов продолжается до тех пор, пока вы не скажете: «Больше находить нечего — нет ни новых классов, ни новых сообщений».

Следующий шаг — построение диаграммы классов. Откроем главную диаграмму (рис. 17.19) классов и добавим в нее классы.

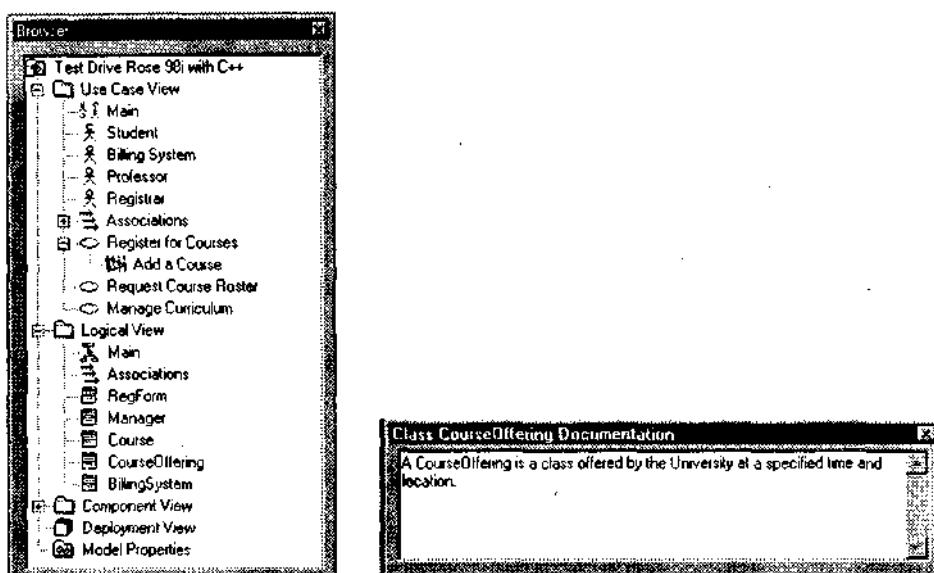


Рис. 17.17. Логическое представление — Рис. 17.18. Окно документации — Logical View Documentation Window

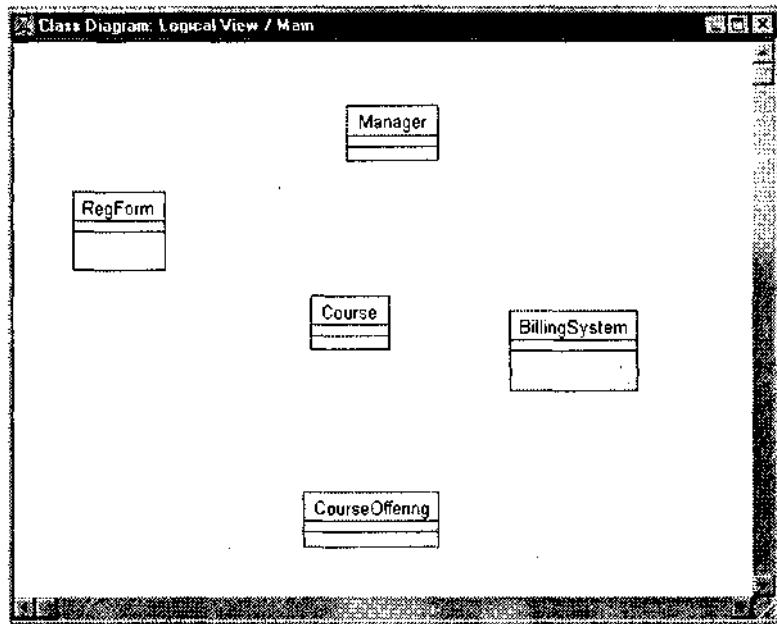


Рис. 17.19. Главная диаграмма классов

1. Для открытия диаграммы выполним двойной щелчок по значку Main в окне браузера.
2. В главном меню выберем команду Query:Add Classes.
3. Для добавления всех классов нажмем кнопку АИ» (выбрать все).
4. Для закрытия окна и добавления классов в диаграмму нажмем кнопку ОК.
5. 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 классу в единицу времени).

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 (рис. 17.20), так как этот класс определяет только интерфейс к внешней учетной системе (billing system).

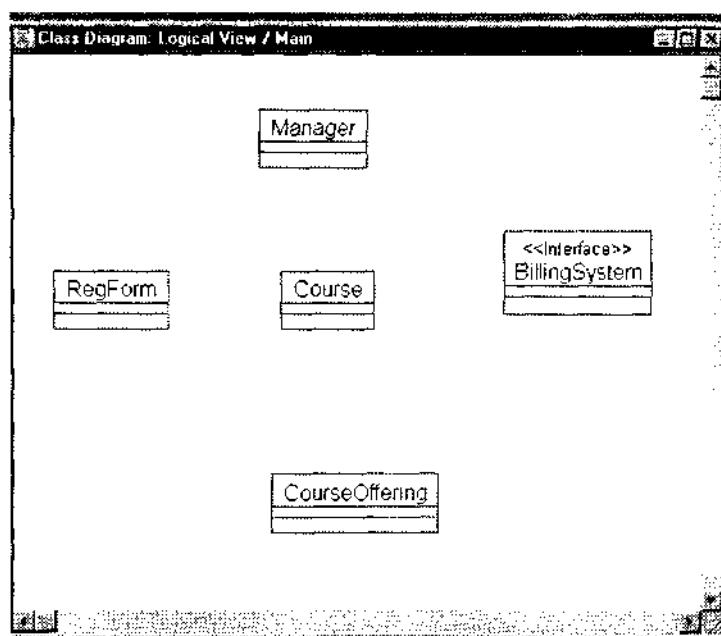


Рис. 17.20. Класс Billing System

1. В главной диаграмме классов выполним двойной щелчок по значку класса BillingSystem. В

результате появляется окно спецификации класса (Class Specification).

2. Щелкнем по стрелке раскрывающегося списка Stereotype.
3. Наберем на клавиатуре слово-стереотип Interface.
4. Закроем окно спецификации, нажав кнопку OK.

Для определения взаимодействия объектов нужно указать отношения между 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.

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. 17.21).

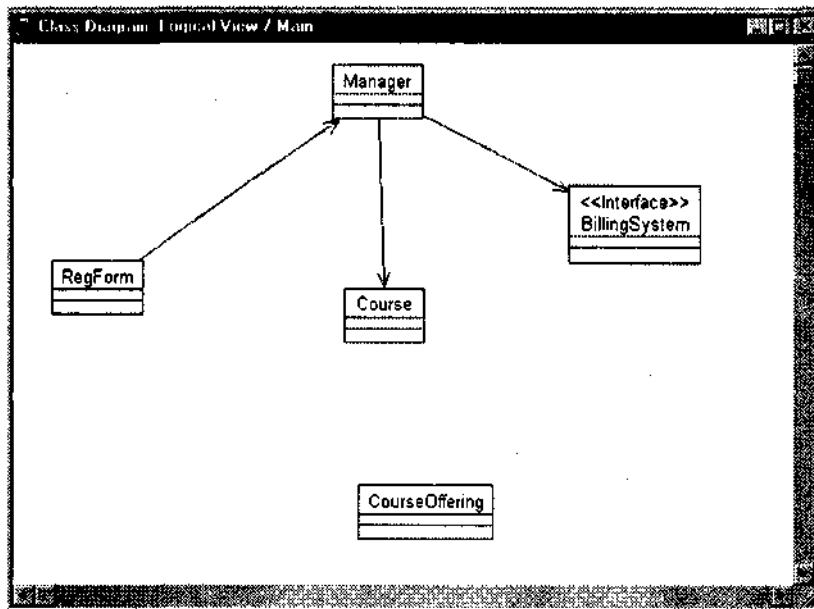


Рис. 17.21. Ассоциации между классами

1. На панели инструментов щелкните по значку односторонней ассоциации (стрелке).
2. Щелкните по классу RegForm и перетащите линию ассоциации на класс Manager.
3. Повторите 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. 11.22), так как предложение Курса CourseOfferings является частью агрегата — класса Course.

1. На панели инструментов щелкните по значку агрегации (линии с ромбиком).
2. Щелкните по классу, представляющему целое — Course.
3. Перетащите линию агрегации на класс, представляющий часть — CourseOffering.

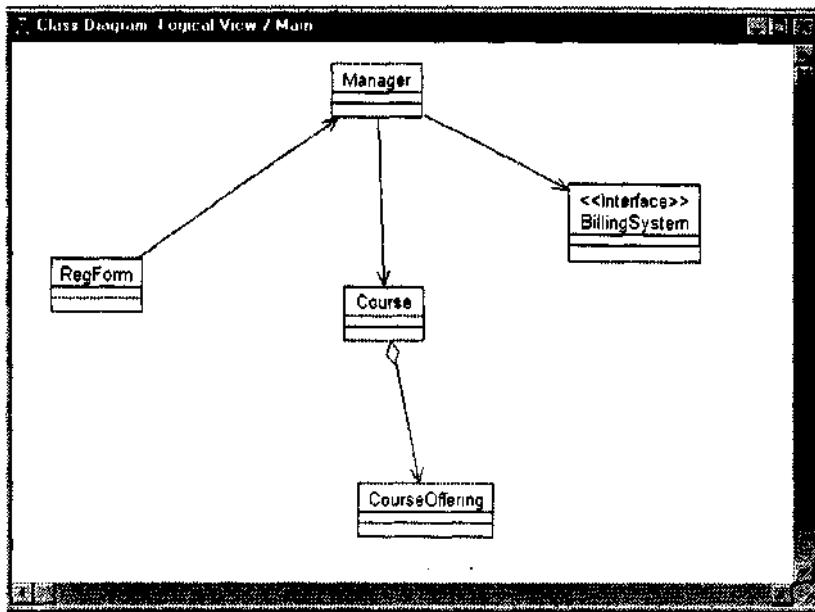


Рис. 17.22. Отношение агрегации

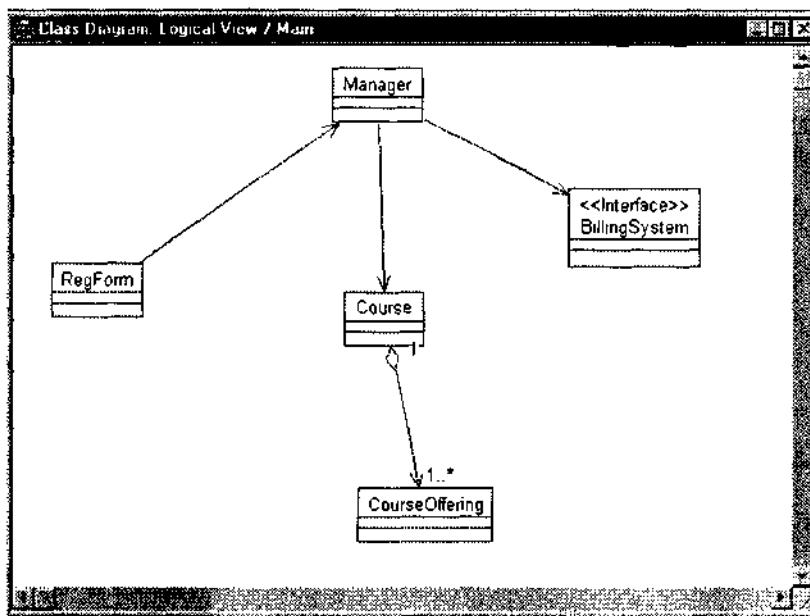


Рис. 17.23. Индикаторы мощности

Для отображения того, «как много» объектов участвует в отношении, к ассоциациям и агрегациям диаграммы могут добавляться индикаторы мощности (рис. 17.23).

1. Щелкните правой кнопкой по линии агрегации возле класса **CourseOffering**.
2. Из контекстного меню выберите команду **Multiplicity:One or More**.
3. Щелкните правой кнопкой по линии агрегации возле класса **Course**.
4. Из контекстного меню выберите команду **Multiplicity:1**.

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 свойств. Структура находится путем исследования проблемных требований и соглашений между разработчиками и заказчиками. В нашей модели каждое предложение курса (**CourseOffering**) является свойством (attribute) класса-агрегата **Course**.

Конечно, класс **CourseOffering** тоже имеет свойства (рис. 17.24). Определим одно из них — количество студентов.

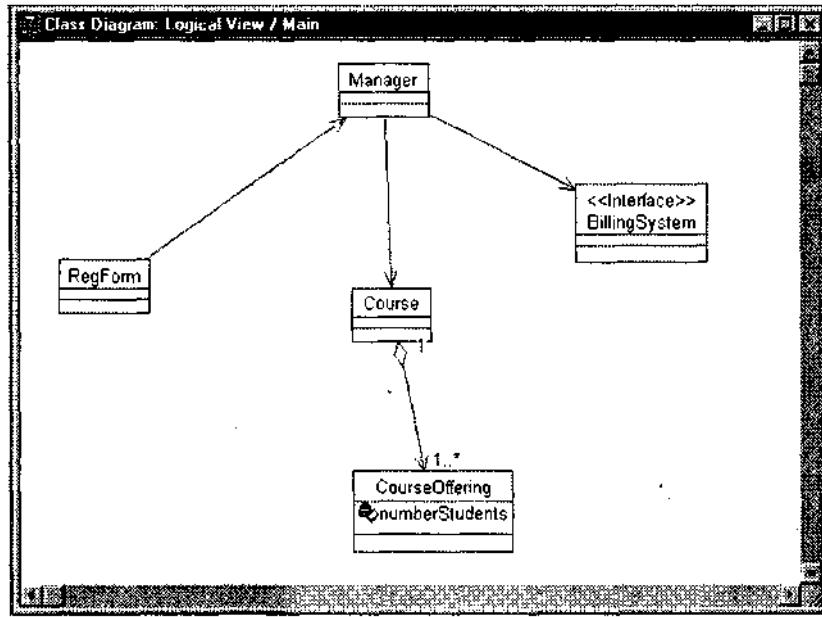


Рис. 17.24. Свойства

1. В диаграмме классов щелкните правой кнопкой по классу CourseOffering.
2. Из контекстного меню выберите команду Insert New Attribute. Это приведет к добавлению в класс свойства.
3. Пока новое свойство остается выделенным, введите его имя — 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.

Первое действие этого шага заключается в привязке объектов (из диаграмм последовательности) к конкретным классам. Выполним такую привязку для нашей модели (рис. 17.25).

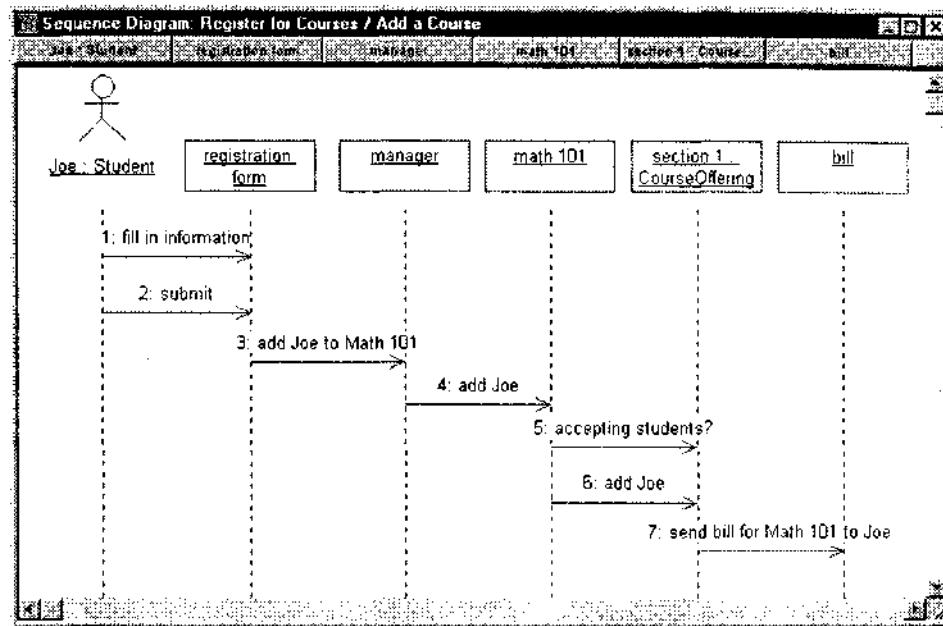


Рис. 17.25. Привязка объектов к классам

1. Для открытия диаграммы последовательности Add a Course выполним двукратный щелчок по ее значку в окне браузера.
 2. В окне браузера щелкнем по значку класса CourseOffering.
 3. Перетащим класс CourseOffering на объект section 1.
- Вот и все. Видим, что имя объекта удлинилось, в нем появились две части, разделенные двоеточием.

Слева от двоеточия записывается имя объекта, а справа — имя класса.

После назначения объекта классу выполняется второе действие — наполнение класса операциями. Как правило, в операции класса превращаются сообщения, получаемые его объектом. При этом обычно сообщения переименовываются — производится согласование имени сообщения и имени операции (рис. 17.26). 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.

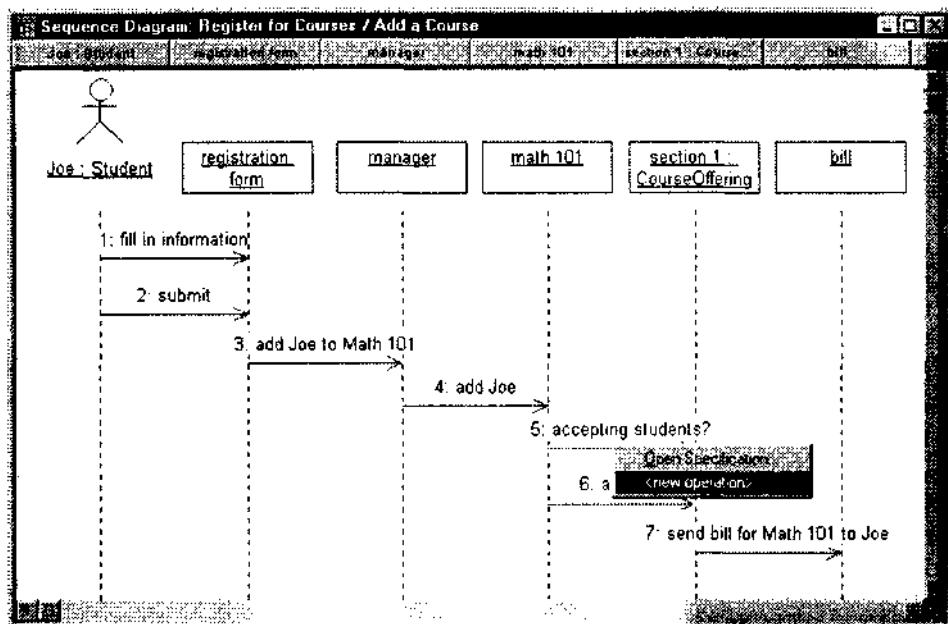


Рис. 17.26. Новая операция

1. Щелкните правой кнопкой по сообщению «add Joe». В результате станет видимым контекстное меню.
2. Выберите команду new operation. В результате станет видимой спецификация операции Operation Specification.
3. Введите имя новой операции — add.
4. Перейдите на вкладку Detail.
5. Щелкните правой кнопкой мыши по полю Arguments.
6. Выберите в контекстном меню команду Insert. В появившейся рамке наберите имя аргумента — Joe. Щелкните вне рамки.
7. Закройте окно спецификации, нажав кнопку OK.

Вы создали новую операцию класса и связали с ней сообщение — оно автоматически поменяло свое имя. Несмотря на переименование, это прямое действие — отталкиваясь от имени сообщения, получить имя операции.

Возможно и обратное действие — отталкиваясь от имени операции, получить имя сообщения. При этом реализуется такая последовательность: отдельно создается новая операция класса, а затем она отображается на существующее сообщение (рис. 17.27).

1. Щелкните правой кнопкой по классу в браузере.
2. В появившемся контекстном меню выберите команду New: Operation. Появляется рамка с надписью opname.
3. Вместо надписи opname наберите имя новой операции класса — offeringOpen.
4. На диаграмме последовательности щелкните правой кнопкой по сообщению «accepting students?». В результате станет видимым контекстное меню.
5. В меню выберите операцию offeringOpen() — сообщение переименовывается (на него отображается операция класса).

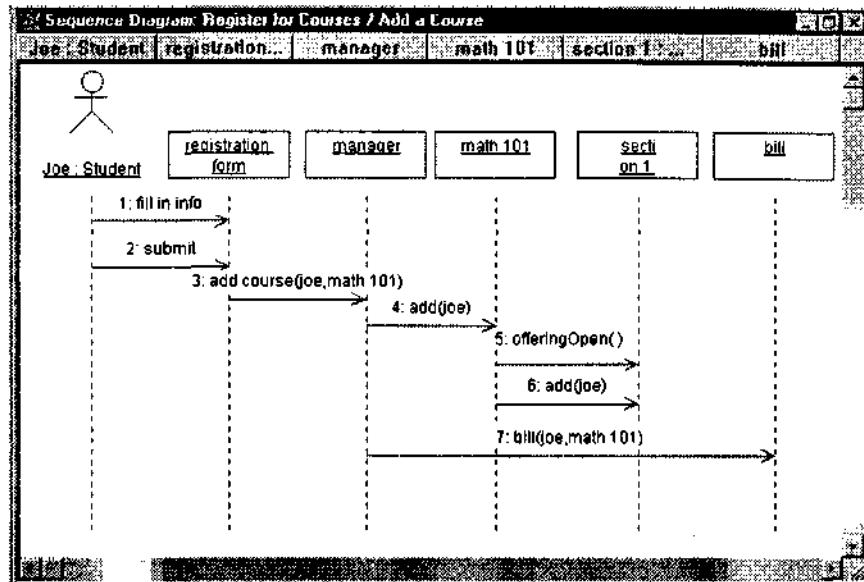


Рис. 17.27. Отображение операции на сообщение

Следующий шаг разработки состоит в настройке описаний классов на конкретный язык программирования. Сам язык выбирается по команде Tools:Options. В появившемся диалоговом окне переходят на вкладку Notation. Название языка выбирается из раскрывающегося списка Default Language. Для нашего примера используем язык Ada 95.

Итак, в ходе анализа и проектирования в визуальную модель добавляются проектные решения (рис. 17.28). После выбора языка программирования для свойств определяют типы данных, а для операций конкретизируют сигнатуры — имена и типы параметров, типы возвращаемых значений.

1. Выполним двойной щелчок по значку класса CourseOffering в окне браузера или диаграмме классов. В результате станет видимым окно спецификации класса.
2. Выберите страницу Attributes (свойства).
3. Щелкните по полю Type. В результате станет видимым раскрывающийся список.
4. Введите требуемый тип данных (Integer).
5. Закройте окно спецификации, нажав кнопку OK.

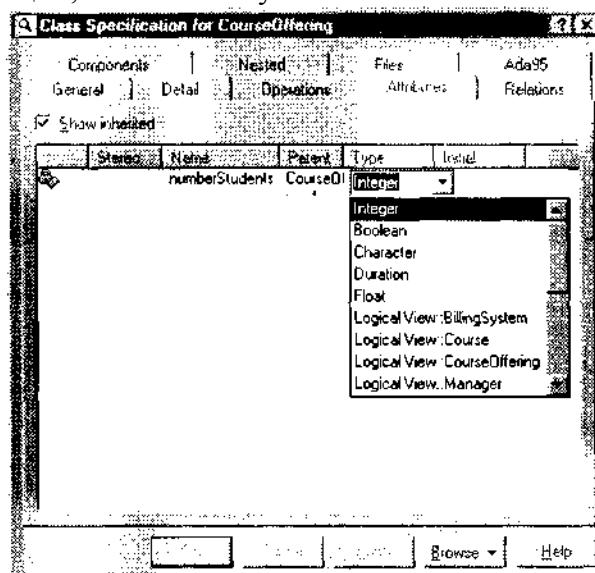


Рис. 17.28. Добавление проектных решений

ПРИМЕЧАНИЕ

Типы данных для свойств можно устанавливать, используя спецификацию Attribute или вводя их в строчке отображения свойства на диаграмме классов (формат attribute:data type).

Теперь зададим тип возвращаемого результата для операции offeringOpen (рис. 17.29).

1. Выполним двойной щелчок по значку класса CourseOffering в окне браузера или диаграмме

классов. В результате станет видимым окно спецификации класса.

2. Выберите страницу Operations.
3. Щелкните по полю Return type. В результате станет видимым раскрывающийся список.
4. Введите требуемый возвращаемый тип (Integer).
5. Закройте окно спецификации, нажав кнопку OK.

ПРИМЕЧАНИЕ

Аргументы операции устанавливают с помощью диалогового окна Operation Specification. Для перехода к этому окну нужно на вкладке (странице) Operations щелкнуть правой кнопкой по имени операции и в появившемся контекстном меню выбрать команду Specification. Далее в появившемся диалоговом окне следует перейти на вкладку Detail.

Аргументы операции и возвращаемый тип можно также установить, вводя их в строчке отображения операции на диаграмме классов (формат operation(argument name:data type):return type).

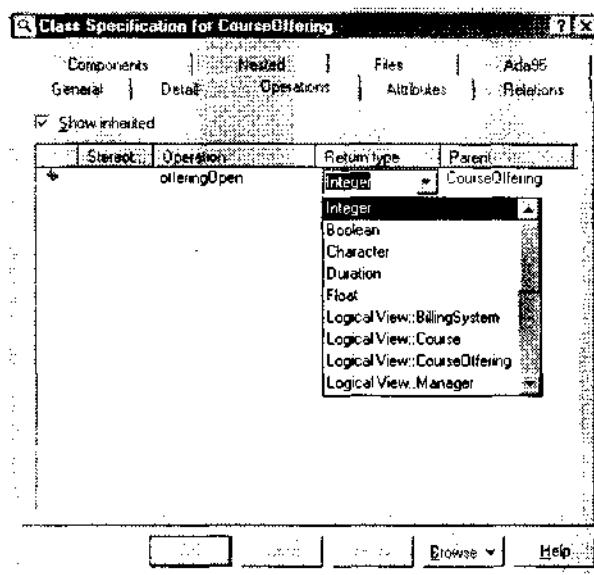


Рис. 17.29. Определение типа возвращаемого результата

Создание компонентной диаграммы

Допустим, что наступил момент, когда нужно генерировать коды для классов модели. Для определения компонентов исходного кода используют компонентное представление (Component View) (рис. 17.30). Среда Rational Rose автоматически создает главную компонентную диаграмму.

1. В окне браузера щелкнем по значку + слева от пакета Component View.

2. Для открытия главной компонентной диаграммы выполним двойной щелчок по значку Main.

В общем случае каждому классу должны соответствовать два компонента — компонент спецификации и компонент реализации. В будущем каждому компоненту будет соответствовать свой файл. Например, в языке C++ классу соответствуют два файла-компоненты: h-файл (файл спецификации) и cpp-файл (файл реализации).

В нашей модели мы создадим один компонент для представления файла спецификации по классу CourseOffering и один компонент для представления файла реализации по классу CourseOffering (рис. 17.31).

Эти файлы будут иметь расширения .ads и .adb соответственно. Файл .ads имеет стереотип Package Specification. Файл .adb имеет стереотип Package Body.

1. На панели инструментов щелкните по значку спецификации пакета Package Specification.
2. Для добавления компонента в диаграмму щелкните в нужном месте диаграммы.
3. Пока новый компонент остается выделенным, введите его имя — CourseOffering.
4. Повторите предыдущие шаги с использованием значка тела пакета Package Body.
5. На панели инструментов щелкните по значку отношения зависимости.
6. Щелкните по компоненту, представляющему .adb-файл (тело пакета), и перетащите стрелку на

компонент, представляющий .ads-файл (спецификация пакета).

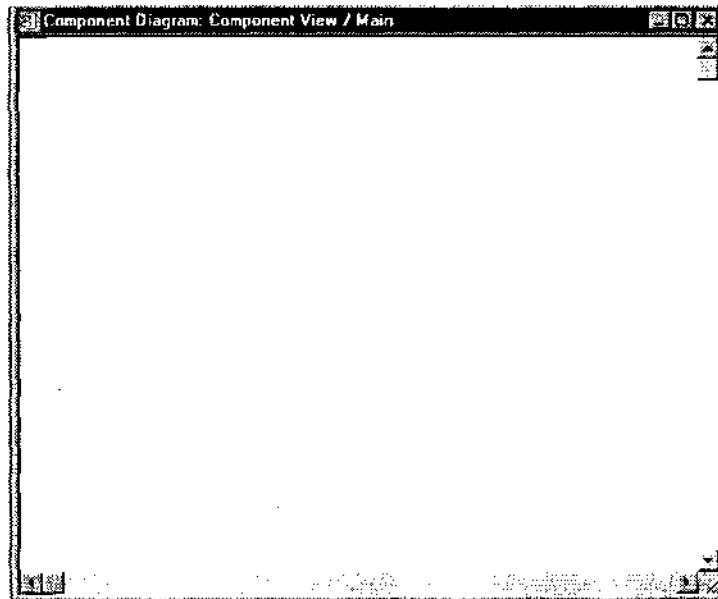


Рис. 17.30. Компонентное представление — Component View

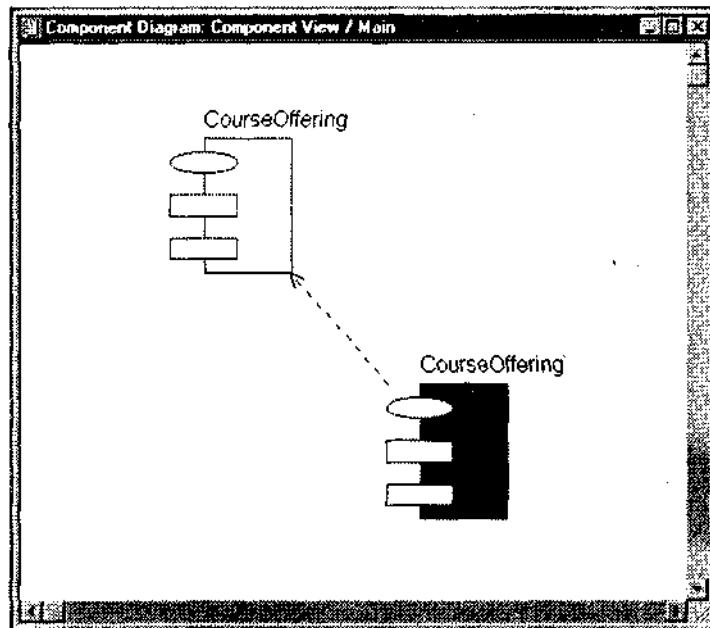


Рис. 17.31. Компонентное представление

После создания компонентов им должны быть назначены классы модели (рис. 17.32).

1. Выполним двукратный щелчок по значку компонента CourseOffering, представляющего .ads-файл (спецификацию пакета), в окне браузера или компонентной диаграмме. В результате станет видимым окно спецификации компонента.
2. Выберите страницу (вкладку) Realizes. Вы увидете список классов модели.
3. Щелкните правой кнопкой по классу CourseOffering. В результате станет видимым контекстное меню.
4. Выберите команду Assign.
5. Закройте окно спецификации, нажав кнопку OK.
6. Выполните аналогичные действия для тела пакета, представляющего .adb-файл.

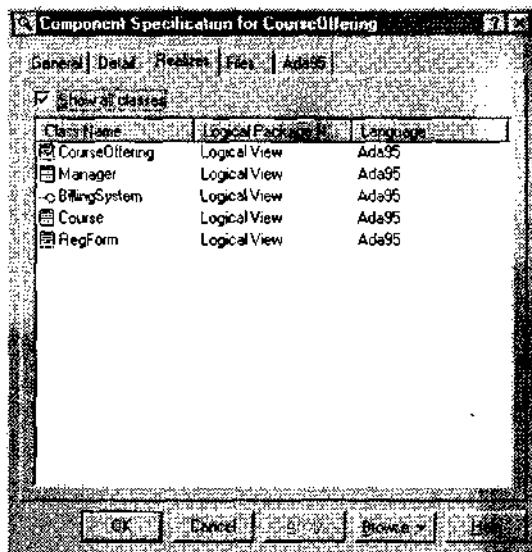


Рис. 17.32. Назначение классов компоненту

Генерация программного кода

Команды для генерации кода на языке Ada 95 содержит пункт Tools главного меню (рис. 17.33).

1. На компонентной диаграмме выделите оба компонента CourseOffering.
 2. Выберите команду Tools:Ada95: Code Generation из главного меню.
- Итоги генерации кода отображаются в окне Code Generation Status (рис. 17.34). Все ошибки заносятся в log-окно.
3. Для завершения процесса генерации кода нажмите кнопку Close.

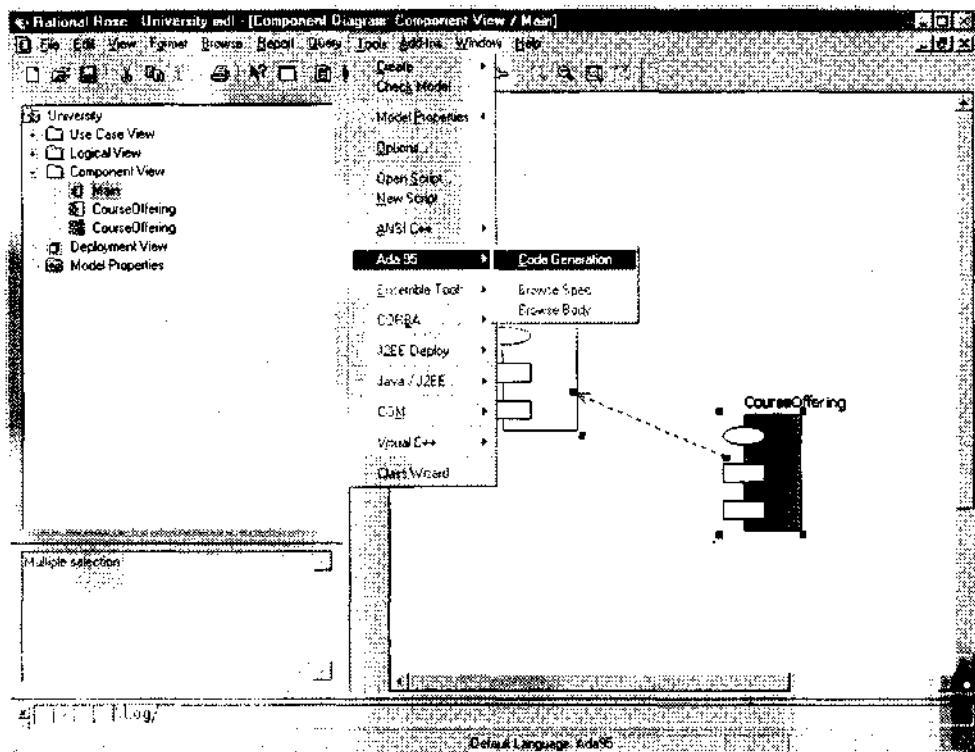


Рис. 17.33. Меню Tools: генерация кода на языке Ada 95

Рис. 17.34. Статус генерации кода

В процессе генерации Rational Rose отображает логическое описание класса в каркас программного кода — в коде появляются языковые описания имени класса, свойств 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, имеющей мост связи с системой Rational Rose. После каждого существенного дополнения программист с помощью возвратного проектирования, основанного на использовании моста связи, будет модифицировать диаграммы классов, вводя в них изменения, соответствующие результатам программирования.

Просмотрим код, сгенерированный средой Rational Rose.

Фрагмент содержания .ads-файла, отражающего спецификацию класса **CourseOffering**, представлен на рис. 17.35. Отметим, что в программный текст добавлено то описание, которое было внесено в модель через окно документации. Более того, система Rational Rose подготавливает код к многократной итеративной модификации, защищая выполняемых изменений. Стандартный раздел программного кода имеет вид

```
--##begin module.privateDeclarations preserve=yes  
--##end module.privateDeclarations
```

The screenshot shows the AdaGIDE interface with the file 'courseoffering.ads' open. The code is as follows:

```
package CourseOffering is  
  --##begin module.declarations preserve=no  
  --##end module.declarations  
  
  -- Class CourseOffering  
  -- Documentation:  
  --   CourseOffering is a class offered by the University  
  --   at a specified time and location  
  -- Concurrency: Sequential  
  -- Persistence: Transient  
  -- Cardinality: n  
  
  type Object is tagged private;  
  type Handle is access Object'Class;
```

Рис. 17.35. Код спецификации класса, сгенерированный средой Rational Rose

Запись module.privateDeclarations обозначает имя раздела. Элемент preserve=(yes/no) говорит системе, можно ли при повторной генерации кода этот раздел изменять или нельзя. После генерации кода программный текст добавляется между операторами ##begin и ##end.

Полный листинг сгенерированного кода спецификации выглядит так:

```
--##begin module.cp preserve=no  
--##end module.cp  
-- Specification CourseOffering (Package Specification)  
-- Dir : C:\Program Files\Rational\Rose\ada95\source  
-- File: courseoffering.ads  
--##begin module.withs preserve=yes  
--##end module.withs  
package CourseOffering is  
  --##begin module.declarations preserve=no  
  --##end module.declarations  
  -- Class CourseOffering  
  -- Documentation:  
  -- CourseOffering is  
  -- Concurrency: Sequential  
  -- Persistence: Transient  
  -- Cardinality: n  
  type Object is tagged private;  
  type Handle is access Object'Class;  
  -- Array declarations
```

```

type Array_Of_Object is
    array (Positive range <>) of Object;
type Access_Array_Of_Object is
    access Array_Of_Object;
-- Standard Operations
function Create return Object;
function Copy (From : in Object) return Object;
procedure Free (This : in out Object);
function Create return Handle;
function Copy (From : in Handle) return Handle;
procedure Free (This : in out Handle);
-- Accessor Operations for Associations
function Get_The_Course (This : in Object)
    return Course.Handle;
pragma Inline (Get_The_Course);
-- Association Operations
procedure Associate (This_Handle : in Handle;
    This_Handle : in Handle);
procedure Associate (This_Handle : in Handle;
    This_Array_Of_Handle : in Array_Of_Handle);
procedure Dissociate (This : in Handle);
procedure Dissociate (This : in Handle);
procedure Dissociate (This : in Array_Of_Handle);
-- Other Operations
function offeringOpen (This : in Object) return Integer;
--##begin module.additionalDeclarations preserve=yes
--##end module.additionalDeclarations
private
--##begin module.privateDeclarations preserve=yes
--##end module.privateDeclarations
type Object is tagged
    record
        -- Data Members for Class Attributes
        numberStudents : Integer;
        -- Data Members for Associations
        The_Course : Course.Handle;
        --##begin CourseOffering.private preserve=no
        --##end CourseOffering.private
    end record;
--##begin module.additionalPrivateDeclarations preserve=yes
--##end module.additionalPrivateDeclarations
end CourseOffering;

```

Соответственно, фрагмент содержания .adb-файла, отображающего тело класса CourseOffering, представлен на рис. 17.36.

```

AdaGIDE [courseoffering.adb]
File Edit Compile Stub Tools Window Help
package body CourseOffering is
    --##begin module.declarations preserve=no
    --##end module.declarations

    -- Standard Operations

    --##begin CourseOffering.Create%Object.documentation preserve=yes
    --##end CourseOffering.Create%Object.documentation

    function Create return Object is
        --##begin CourseOffering.Create%Object.declarations preserve=no
        --##end CourseOffering.Create%Object.declarations
    begin
        --##begin CourseOffering.Create%Object.statements preserve=yes
        [statement]
        --##end CourseOffering.Create%Object.statements
    end Create;
    --##begin CourseOffering.Copy%Object.documentation preserve=yes
    --##end CourseOffering.Copy%Object.documentation

    function Copy (From : in Object) return Object is
        --##begin CourseOffering.Copy%Object.declarations preserve=no
        --##end CourseOffering.Copy%Object.declarations
    begin
        --##begin CourseOffering.Copy%Object.statements preserve=no
        [statement]
        --##end CourseOffering.Copy%Object.statements
    end Copy;
    --##begin CourseOffering.Free%Object.documentation preserve=yes
    --##end CourseOffering.Free%Object.documentation

    procedure Free (This : in out Object) is

```

Рис. 17.36. Код тела класса, сгенерированный Rational Rose

Полный листинг сгенерированного кода для тела класса выглядит следующим образом:

```

--##begin module.cp preserve=no
--##end module.cp
-- Body CourseOffering (Package Body)
-- Dir : C:\Program Files\Rational\Rose\ada95\source
-- File: courseoffering.adb
with Unchecked_Deallocation;
--##begin module.withs preserve=yes
--##end module.withs
package body CourseOffering is
    --##begin module.declarations preserve=no
    --##end module.declarations
    -- Standard Operations
    --##begin CourseOffering.CreatefcObject.documentation preserve=yes
    --##end CourseOffering.Create%Object.documentation
    function Create return Object is
        --##begin CourseOffering.Create%Object.declarations preserve=no
        --##end CourseOffering.Create%Object.declarations
    begin
        --##begin CourseOffering.Create%Object.statements preserve=no
        [statement]
        --##end CourseOffering.Create%Object.statements
    end Create;
    --##begin CourseOffering.Copy%Object.documentation preserve=yes
    --##end CourseOffering.Copy%Object.documentation
    function Copy (From : in Object) return Object is
        --##begin CourseOffering.Copy%Object.declarations preserve=no
        --##end CourseOffering.Copy%Object.declarations
    begin
        --##begin CourseOffering.Copy%Object.statements preserve=no
        [statement]
        --##end CourseOffering.Copy%Object.statements
    end Copy;
    --##begin CourseOffering.Free%Object.documentation preserve=yes
    --##end CourseOffering.Free%Object.documentation
    procedure Free (This : in out Object) is

```

```

--##begin CourseOffering.Free%Object.declarations preserve=yes
--##end CourseOffering.Free%Object.declarations
begin
  --##begin CourseOffering.Free%Object.statements preserve=no
  [statement]
  --##end CourseOffering.Free%Object.statements
end Free;
--##beginCourseOffering.Create%Handle.documentation preserve=yes
--##end CourseOffering.Create%Handle.documentation
function Create return Handle is
  --##begin CourseOffering.Create%Handle.declarations preserve=no
  --##end CourseOffering.Create%Handle.declarations
begin
  --##begin CourseOffering.Create%Handle.statements preserve=no
  [statement]
  --##end CourseOffering.Create%Handle.statements
end Create;
--##begin CourseOffering.Copy%Handle.documentation preserve=yes
--##end CourseOffering.Copy%Handle.documentation
function Copy (From : in Handle) return Handle is
  --##begin CourseOffering.Copy%Handle.declarations preserve=no
  --##end CourseOffering.Copy%Handle.declarations
begin
  --##begin CourseOffering.Copy%Handle.statements preserve=no
  [statement]
  --##end CourseOffering.Copy%Handle.statements
end Copy;
--##begin CourseOffering.Free%Handle.documentation preserve=yes
--##end CourseOffering.Free%Handle.documentation
procedure Free (This : in out Handle) is
  --##begin CourseOffering.Free%Handle.declarations preserve=yes
  --##end CourseOffering.Free%Handle.declarations
begin
  --##begin CourseOffering.Free%Handle.statements preserve=no
  [statement]
  --##end CourseOffering.Free%Handle.statements
end Free;
-- Other Operations
--##begin CourseOffering.offeringOpen%Object.940157546.documentation preserve=yes
--##end CourseOffering.offeringOpen%Object.940157546.documentation
function offeringOpen (This : in Object) return Integer is
  --##begin CourseOffering.offeringOpen%Object.940157546.declarations preserve=yes
  --##end CourseOffering.offeringOpen%Object.940157546.declarations
begin
  --##begin CourseOffering.offeringOpen%Object.940157546.statements preserve=yes
  [statement]
  --##end CourseOffering.offeringOpen%Object.940157546.statements
end offeringOpen;
-- Accessor Operations for Associations
--##begin CourseOffering.Get_The_Course%Object.documentation preserve=yes
--##end CourseOffering.Get_The_Course%Object.documentation
function Get_The_Course (This : in Object) return Course.Handle is
  --##begin CourseOffering.Get_The_Course%Object.declarations preserve=no
  --##end CourseOffering.Get_The_Course%Object.declarations
begin
  --##begin CourseOffering.Get_The_Course%Object.statements preserve=no
  return This.The_Course;
  --##end CourseOffering.Get_The_Course%Object.statements
end Get_The_Course;
-- Association Operations

```

```

--##begin module.associations preserve=no
generic
    type Role_Type is tagged private;
    type Access_Role_Type is access Role_Type'Class;
    type Index is range <>;
    type Array_Of_Access_Role_Type is
        array (Index range <>) of Access_Role_Type;
    type Access_Array_Of_Access_Role_Type is
        access Array_Of_Access_Role_Type;
package Generic_Tagged_Association is
    procedure Set (This_Access_Array : in out
                  Access_Array_Of_Access_Role_Type;
                  This_Array : Array_Of_Access_Role_Type;
                  Limit : Positive := Positive
                           ((Index'Last - Index'First) + 1));
    function Is_Unique (This_Array : Array_Of_Access_Role_Type;
                       This : Access_Role_Type) return Boolean;
    function Unique (This_Array : in Array_Of_Access_Role_Type)
                    return Array_Of_Access_Role_Type;
    pragma Inline (Set, Is_Unique, Unique);
end Generic_Tagged_Association;
package body Generic_Tagged_Association is
    procedure Free is new Unchecked_Deallocation
        (Array_Of_Access_Role_Type,
         Access_Array_Of_Access_Role_Type);
    procedure Set (This_Access_Array : in out
                  Access_Array_Of_Access_Role_Type;
                  This_Array : Array_Of_Access_Role_Type;
                  Limit : Positive := Positive
                           ((Index'Last - Index'First) + 1)) is
        Valid : Boolean;
        Pos : Index := This_Array'First;
        Last : constant Index := This_Array'Last;
        Temp_Access_Array : Access_Array_Of_Access_Role_Type;
begin
    if Positive(This_Array'Length) > Limit then
        raise Constraint_Error;
    end if;
    if This_Access_Array = null then
        -- Allocate entries.
        This_Access_Array := new
            Array_Of_Access_Role_Type'(This_Array);
        return;
    end if;
    for I in This_Array'Range loop
        Valid := Is_Unique (Th1s_Access_Array.all. This_Array (I));
        pragma Assert (Valid);
    end loop;
    -- Reuse any empty slots.
    for I in This_Access_Array'Range loop
        if This_Access_Array (I) = null then
            This_Access_Array (I) := This_Array (Pos);
            Pos := Pos + 1;
            if Pos > Last then return;
            end if;
        end if;
    end loop;
    if Positive(This_Access_Array'Length + (Last - Pos + 1)) > Limit then raise
        Constraint_Error;
    end if;

```

```

-- For any remaining entries, combine by reallocating.
Temp_Access_Array := new Array_Of_Access_Role_Type'
(This_Access_Array.all & This_Array (Pos .. Last));
Free (This_Access_Array);
This_Access_Array := Temp_Access_Array;
end Set;
function Is_Unique (This_Array : Array_Of_Access_Role_Type;
                    This : Access_Role_Type) return Boolean is
begin
  if This = null then return False;
  end if;
  for I in This_Array'Range loop
    if This_Array (I) = This then return False;
    end if;
  end loop;
  return True;
end Is_Unique;
function Unique (This_Array : in Array_Of_Access_Role_Type)
                return Array_Of_Access_Role_Type is
  First : constant Index := This_Array'First;
  Count : Index := First;
  Temp_Array : Array_Of_Access_Role_Type (This_Array'Range);
begin
  for I in This_Array'Range loop
    if Is_Unique (Temp_Array (First .. Count - 1),
                  This_Array (I)) then
      Temp_Array (Count) := This_Array (I);
      Count := Count + 1;
    end if;
  end loop;
  return Temp_Array (First .. Count - 1);
end Unique;
end Generic_Tagged_Association;
package Role_Object is new Generic_Tagged_Association
  (Object,
   Handle,
   Positive,
   Array_Of_Handle,
   Access_Array_Of_Handle);
--##end module.associations
procedure Associate (This_Handle : in Handle; This_Handle
                     : in Handle) is
  --#begin Associate%38099E7D0190.declarations preserve=no
  use Role_Object;
  --##end Associate%38099E7D0190.declarations
begin
  --##begin Associate%38Q99E70Q190.statements preserve=no
  pragma Assert (This_Handle /= null);
  pragma Assert (This_Handle /= null);
  pragma Assert (This_Handle.The_Course = null or
                 else This_Handle.The_Course = This_Handle);
  This_Handle.The_Course := This_Handle;
  Set (This_Handle.The_CourseOffering, Array_Of_Handle'(1 =>
                                         This_Handle));
  --##end Associate3%38099E7D0190. statements
end Associate;
procedure Associate (This_Handle : in Handle;
                     This_Array_Of_Handle : in Array_Of_Handle) is
  --##begin Associate%(1,N)38099E7D0190.declarations preserve=no
  use Role_Object;

```

```

Temp_Array_Of_Handle : constant Array_Of_Handle := Unique
    (This_Array_Of_Handle);
--«end Associate%(l,N)38099E7D0190.declarations
begin
    --##begin Associate%(l.N)38099E7D0190.statements preserve=no
    pragma Assert (This_Handle /= null);
    pragma Assert (Temp_Array_Of_Handle'Length > 0);
    for I in Temp_Array_Of_Handle'Range loop
        pragma Assert (Temp_Array_Of_Handle (I).The_Course = null or else
Temp_Array_Of_Handle (I).The_Course - This_Handle);
        Temp_Array_Of_Handle (I).The_Course := This_Handle;
        end loop;
        Set (This_Handle.The_CourseOffering, Temp_Array_Of_Handle);
--##end Associate%(1,N)38099E7D0190.statements
end Associate;
procedure Dissociate (This : in Handle) is
    --##begin Dissociate«38099E7D0190.declarations preserve=yes
    --##end Dissociate«38099E7D0190.declarations
begin
    --##begin Dissociate%38099E7D0190.statements preserve=no
    pragma Assert (This /= null);
    for I in This.The_CourseOffering'Range loop
        if This.The_CourseOffering (I) /= null then
            if This.The_CourseOffering (I).The_Course = This then
                This.The_CourseOffering (I).The_Course := null;
                This.The_CourseOffering (I) := null;
            end if;
        end if;
        end loop;
    --##end Dissociate%38099E7D0190.statements
end Dissociate;
procedure Dissociate (This := in Handle) is
    --##begin Dissociate%38099E7D0190.declarations preserve=yes
    --##end Dissociate%38099E7D0190.declarations
begin
    --##begin Dissociate%38099E7D0190.statements preserve=no
    pragma Assert (This /= null);
    for I in This.The_Course.The_CourseOffering'Range loop
        if This.The_Course.The_CourseOffering (I) = This then
            This.The_Course.The_CourseOffering (I) :=null;
            This.The_Course :=null;
            exit;
        end if;
    end loop;
    --##end Dissociat%38099E7D0190.statements
end Dissociate;
procedure Dissociate (This : in Array_Of_Handle) is
    --##begin Dissociate%(M)38099E7D0190.declarations preserve=yes
    --##end Dissociate%(M)38099E7D0190.declarations
begin
    --##begin Dissociate%(M)38099E7D0190.statements preserve=no
    for I in This'Range loop
        if This (I) /= null then
            Dissociate (This (I));
        end if;
    end loop;
    --##end Dissociate%(M)38099E7D0190.statements
end Dissociate;
--##begin module.additionalDeclarations preserve=yes
--##end module.additionalDec!arations

```

```
begin
--##begin module.statements preserve=no
null;
--##end module.statements
end CourseOffering;
```

Отметим, что в теле есть стандартные методы, которые не задавались в модели, — например, методы constructor, destructor и get/set. Их автоматическая генерация была задана настройкой среды — свойствами генерации. Система обеспечивает настройку параметров генерации для уровней класса, роли, свойства (атрибута) и проекта в целом. Более подробную информацию о свойствах генерации кода можно получить из help-файла.

ЗАКЛЮЧЕНИЕ

Современная программная инженерия (Software Engineering) — молодая и быстро развивающаяся 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 «программе-рам» 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 конструирования 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. Реорганизация (рефакторинг), особенности конструирования web-приложений, работа с базами данных — вот неполный перечень тем, обсудить которые не позволили ресурсные ограничения. Хотелось бы обратить внимание на новейшие постобъектные методологии — аспектно-ориентированное и многомерное проектирование и программирование. Они представляют собой новую высоту в стремительном полете 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, я верю в вас, я горжусь вами. Дерзайте, творите, разочаровывайтесь и очаровывайтесь! Я уверен, вы построите достойное информационное обеспечение

человеческого общества!

ПРИЛОЖЕНИЕ А.

ФАКТОРЫ ЗАТРАТ ПОСТАРХИТЕКТУРНОЙ МОДЕЛИ СОСМО II

Значительную часть времени при использовании модели СОСМО 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 $D=\text{SQRT}(B^{**2}-4*A*C)$	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	Использование стандартных математических и статистических подпрограмм. Базовые матричные / векторные	Обработка ввода- вывода, включающая выбор устройства, проверку состояния и обработку ошибок

	уровня — поддержка распределенной обработки	операции	
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 неструктурный 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 относительно требований жизненного цикла	Очень 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 ≤ 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	Сверхвысокий
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 — ежемесячные изменения — каждые 12мес.; незначительные — ежемесячные изменения — каждые 6 мес.; мес.; недели	Значительные изменения — ежемесячные изменения — каждые 2 нед.; недели	Значительные изменения — ежемесячные изменения — каждые 2 недели	Значительные изменения — ежемесячные изменения — 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	3года	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:	Один телефон, почта	Индивидуальные почты	Узкополосный mail FAX	e- 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 восстанавливающие потери	Умеренная, восстанавливающие потери	Высокая, финансовые потери	Риск для человеческой жизни	
DATA		байты БД/LOC 10≤D/P<100 прогр. <10,93	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

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 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	
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	Один телефон, почта 1,25	Индивидуальные телефоны, FAX 1,10	Узкополосный e-mail 1,00	Широкополое коммуникации 0,92	Широкополосные коммуникации, иногда видеоконференции 0,84	Интерактивные 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.

ПРИЛОЖЕНИЕ Б. ТЕРМИНОЛОГИЯ ЯЗЫКА UML И УНИФИЦИРОВАННОГО ПРОЦЕССА

В данном приложении приведен словарь основных терминов языка UML и унифицированного процесса разработки, описываемого в учебнике.

Словарь терминов

Абстрактный класс (abstract class)	Класс, объект которого не может быть создан непосредственно
Агрегат (aggregate)	Класс, описывающий «целое» в отношении агрегации
Агрегация (aggregation)	Специальная форма ассоциации, определяющая отношение «часть-целое» между агрегатом (целым) и частями
Актер (actor)	Связанный набор ролей, исполняемый пользователями при взаимодействии с элементами Use Case
Активация (activation)	Выполнение соответствующего действия
Активный класс (active class)	Класс, экземпляры которого являются активными объектами. См. <i>процесс, задача, поток</i>
Активный объект (active object)	Объект, являющийся владельцем процесса или потока, которые инициируют управляющую деятельность
Артефакт (artifact)	Документ, отчет или выполняемый элемент. Артефакт может вырабатываться, обрабатываться или потребляться
Асинхронное действие (asynchronous action)	Запрос, отправляемый объекту без паузы для ожидания результата
Ассоциация (association)	Семантическое отношение между классификаторами, задающее набор связей между их экземплярами
Бизнес-модель (business model)	Определяет абстракцию организации, для которой создается система
Бинарная ассоциация (binary association)	Ассоциация между двумя классами
Взаимодействие (interaction)	Поведение, заключающееся в обмене набором сообщений между набором объектов (в определенном контексте и для достижения определенной цели)
Видимость (visibility)	Показывает, как может быть увидено и использовано другими данное имя
Временный объект (transient object)	Объект, существующий только во время выполнения задачи или процесса, которые его создали
Действие (action)	Исполняемое атомарное вычисление. Действие инициируется при получении объектом сообщения или изменении значения его свойства. В результате действия изменяется состояние объекта
Делегирование (delegation)	Способность объекта посыпать сообщение другому объекту в ответ на прием чужого сообщения
Деятельность (activity)	Состояние, в котором проявляется некоторое поведение
Диаграмма (diagram)	Графическое представление набора элементов, обычно в виде связного графа, в вершинах которого находятся предметы, а дуги представляют собой их отношения
Диаграмма Use Case (use case diagram)	Диаграмма, показывающая набор элементов Use Case, актеров и их отношений. Диаграмма Use Case относится к статическому представлению Use Case, создаваемому для системы

Диаграмма взаимодействия (interaction diagram)	Диаграмма, показывающая взаимодействие, включающее в себя набор объектов и их отношений, а также пересылаемые между объектами сообщения. Диаграммы взаимодействия относятся к динамическому представлению системы. Это общий термин, применяемый к различным видам диаграмм, на которых изображено взаимодействие объектов, включая диаграммы сотрудничества и диаграммы последовательности
Диаграмма деятельности (activity diagram)	Диаграмма, показывающая переходы от одного вида деятельности к другому. Диаграммы деятельности относятся к динамическому представлению системы. Диаграмма деятельности является специальной разновидностью диаграммы схем состояний, в которой все или большинство состояний являются состояниями действий, а все или большинство переходов срабатывают при завершении действий в исходных состояниях
Диаграмма классов (class diagram)	Диаграмма, показывающая набор классов, интерфейсов, коопераций, а также их отношения. Диаграмма классов относится к статическому проектному представлению системы. Эта диаграмма показывает набор декларативных (статических) элементов
Диаграмма объектов (object diagram)	Диаграмма, показывающая набор объектов и их отношений в некоторый момент времени. Диаграмма объектов относится к статическому проектному представлению или статическому представлению процессов системы
Диаграмма последовательности (sequence diagram)	Диаграмма взаимодействия, выделяющая временную последовательность передачи сообщений
Диаграмма размещения (deployment diagram)	Диаграмма, показывающая набор узлов и их отношения. Диаграмма размещения относится к статическому представлению размещения системы
Диаграмма сотрудничества (collaboration diagram)	Диаграмма взаимодействия, которая выделяет структурную организацию объектов, посылающих и принимающих сообщения; диаграмма, которая демонстрирует организацию взаимодействия между экземплярами и их связи друг с другом
Диаграмма схем состояний (statechart diagram)	Диаграмма, показывающая конечный автомат. Диаграммы схем состояний относятся к динамическому представлению системы
Единица дистрибуции (distribution unit)	Набор объектов или компонентов, которые предназначены для выполнения одной задачи или работы на одном процессоре
Зависимость (dependency)	Семантическое отношение между двумя предметами, при котором изменение одного предмета (независимого предмета) влияет на семантику другого предмета (зависимого предмета)
Задача (task)	Единичный путь выполнения программы, динамической модели или другого представления потока управления; нить или процесс
Запустить (fire)	Выполнить переход из состояния в состояние
Иерархия вложенности (containment hierarchy)	Иерархия пространств имен, содержащих элементы и отношения вложенности между ними

Импорт (import)	В контексте пакетов — зависимость, показывающая, на классы какого пакета могут ссылаться классы данного пакета (включая пакеты, рекурсивно вложенные в данный)
Имя (name)	То, как вы называете предмет, отношение или диаграмму; строка, используемая для идентификации элемента
Интерфейс (interface)	Набор операций, используемых для описания услуг класса или компонента
Исполняемый модуль (executable)	Программа, которая может выполняться в узле
Использование (usage)	Зависимость, при которой один элемент (клиент) для корректного функционирования нуждается в присутствии другого элемента (поставщика)
Кардинальное число (cardinality)	Число элементов в наборе
Каркас (framework)	Архитектурный паттерн, предоставляющий расширяемый шаблон приложения в какой-либо предметной области
Класс (class)	Описание набора объектов, имеющих одинаковые свойства, операции, отношения и семантику
Класс-ассоциация (association class)	Элемент моделирования, имеющий одновременно характеристики класса и ассоциации. Класс-ассоциация может рассматриваться как ассоциация, имеющая также характеристики класса, или как класс, обладающий характеристиками ассоциации
Классификатор (classifier)	Механизм описания структурных и поведенческих характеристик. Классификаторами являются интерфейсы, классы, типы данных, компоненты и узлы
Клиент (client)	Классификатор, запрашивающий услуги у другого классификатора
Композит (composite)	Класс, связанный с одним или более классами отношением композиции
Композиция (composition)	Сильная форма агрегации, при которой время жизни частей и целого совпадают. Части не существуют отдельно и при удалении композита должны быть уничтожены
Компонент (component)	Физическая заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию набора интерфейсов
Компонентная диаграмма (component diagram)	Диаграмма, показывающая набор компонентов и их отношений. Компонентные диаграммы относятся к статическому компонентному представлению системы
Конечный автомат (state machine)	Поведение, которое определяется последовательностью состояний, через которые проходит объект в течение своей жизни в ответ на поступление сообщений, вместе с его реакцией на эти сообщения
Конкретный класс (concrete class)	Класс, для которого возможно создание экземпляров
Контейнер (container)	Объект, создаваемый для хранения других объектов и предоставляющий операции для доступа к своему содержимому в определенном порядке
Контекст (context)	Набор связанных элементов, ориентированных на достижение определенной цели, например, определение операции

Кооперация (collaboration)	Сообщество классов, интерфейсов и других элементов, работающих вместе с целью реализации некоторого кооперативного поведения. Кооперация больше, чем простая сумма элементов. Описание того, как элементы, такие как элементы Use Case или операции, реализуются набором классификаторов и ассоциаций, играющих определенные роли определенным образом См. линия жизни объекта
«Линия жизни» (lifeline) Линия жизни объекта (object lifeline)	Линия на диаграмме последовательности, которая отражает существование объекта в течение некоторого периода времени
Местоположение (location)	Место размещения компонента в узле
Метакласс (metaclass)	Класс, экземпляры которого являются классами
Метод (method)	Реализация операции. Определяет алгоритм или процедуру, обеспечивающую операцию.
Механизм расширения (extensibility mechanism)	Один из трех механизмов (стереотипы, теговые величины и ограничения), который может использоваться для контролируемого расширения UML
Множественная классификация (multiple classification)	Семантическая вариация обобщения, в которой объект может принадлежать более чем одному классу
Множественное наследование (multiple inheritance)	Семантическая вариация обобщения, в которой тип может иметь более одного супертипа
Множественность (multiplicity)	Спецификация диапазона возможных кардинальных чисел набора
Модель (Model)	Семантически ограниченное абстрактное представление системы
Модель Use Case (Use case model)	Определяет функциональные требования к системе
Модель анализа (analysis model)	Интерпретирует требования к системе в терминах проектной модели
Модель области определения (domain model)	Фиксирует контекстное окружение системы
Модель процессов (process model)	
Модель размещения (deployment model)	
Модель реализации (implementation model)	Определяет параллелизм в системе и механизмы синхронизации
Наследование (inheritance)	Определяет аппаратную топологию, в которой исполняется система
Наследование интерфейса (interface inheritance)	Определяет части, которые используются для сборки и реализации физической системы
Нить (thread)	Механизм, при помощи которого более специализированные элементы включают в себя структуру и поведение более общих элементов
Область действия (scope)	Наследование интерфейса более специализированным элементом, не включает наследования реализации
Обобщение (generalization)	Облегченный поток управления, который может выполняться параллельно с другими нитями того же процесса
Объект (object)	Контекст, который придает имени определенный смысл
Объект длительного хранения (persistent object)	Отношение обобщения/специализации, когда объекты специализированного элемента (подтипа) могут замещать объекты обобщенного элемента (супертипа) См. экземпляр
	Объект, сохраняющийся после завершения процесса или задачи, в ходе которой он был создан

Объектный язык ограничений (object constraint language (OCL))	Формальный язык, используемый для создания ограничений, не имеющих побочных эффектов
Обязанность (responsibility)	Контракт или обязательство типа или класса
Ограничение (constraint)	Расширение семантики элемента UML, позволяющее добавлять к нему новые правила или изменять существующие
Одиночное наследование (single inheritance)	Семантический вариант обобщения, при котором каждый тип может иметь только один супертип
Операция (operation)	Обслуживание, которое может запрашиваться у объекта. Операция имеет сигнатуру, которая задает допустимые фактические параметры
Отношение (relationship)	Семантическая связь между элементами
Отношение трассировки (trace)	Зависимость, указывающая на историческую связь или связь обработки между двумя элементами, представляющими одну и ту же концепцию, без определения правил вывода одного элемента из другого
Отправитель (сообщения) (sender)	Объект, посылающий экземпляр сообщения объекту-получателю
Отправление (send)	Посылка экземпляра сообщения от отправителя получателю
Пакет (package)	Механизм общего назначения для группировки элементов
Параллелизм (concurrency)	Осуществление двух или более видов деятельности в один и тот же временной интервал. Параллелизм может быть осуществлен путем квантования процессорного времени или одновременного выполнения двух или более потоков
Параметр (parameter)	Определение переменной, которая может изменяться, передаваться или возвращаться
Паттерн (pattern)	Паттерн является решением типичной проблемы в определенном контексте
Переход (transition)	Отношение между двумя состояниями, показывающее, что объект, находящийся в первом состоянии, в случае некоторого события и выполнения определенных условий совершил некоторые действия и перейдет во второе состояние
Плавательная дорожка (swim lane)	Область на диаграмме деятельности для назначения ответственного за действие
Побуждение (stimulus)	Операция или сигнал
Подсистема (subsystem)	Группировка элементов, в которой каждый элемент содержит описание поведения, предоставляемого другим элементам подсистемы
Подтип (subtype)	В отношении обобщения — специализация другого типа, супертипа
Получатель (receiver)	Объект, обрабатывающий экземпляр сообщения, поступивший от объекта-отправителя
Полюс (конец) ассоциации (association end)	Конечная точка ассоциации, которая связывает ассоциацию с классификатором
Полюс (конец) связи (link end)	Экземпляр полюса (конца) ассоциации
Поставщик (supplier)	Тип, класс или компонент, предоставляющие услуги, используемые другими
Постусловие (postcondition)	Условие, которое должно выполняться после завершения операции
Представление (view)	Проекция модели, рассматриваемая с определенной

Предусловие (precondition)	точки зрения, в которой показаны существенные и опущены несущественные детали
Прием (receive)	Условие, которое должно выполняться при вызове операции
Примечание (comment)	Обработка экземпляра сообщения, поступившего от объекта — отправителя
Примечание (note)	Примечание, добавляемое к элементу или группе элементов
Примитивный тип (primitive type)	Комментарий, добавляемый к элементу или набору элементов
Проектная модель (design model)	Предопределенный базовый тип, например целое число или строка
Пространство имен (namespace)	Определяет словарь проблемы и ее решение
Процесс (process)	Часть модели, в которой могут определяться и использоваться имена. Внутри пространства имен каждое имя имеет единственный смысл
Рабочий поток процесса (process workflow)	Тяжеловесный поток управления, который может выполнять параллельно с другими процессами
Реализация (realization)	Логическая группировка действий
Роль (role)	Семантическое отношение между классификаторами, когда один классификатор определяет контракт, который другие классификаторы должны гарантированно выполнять
Свойство (attribute)	Определенное поведение сущности в определенном контексте
Связывание (binding)	Именованная характеристика классификатора, задающая набор возможных значений, которые определяют состояния экземпляров классификатора (например, объектов)
Связь (link)	Создание конкретного элемента на основе шаблона (путем сопоставления параметрам шаблона конкретных аргументов)
Сигнал (signal)	Семантическая связь между объектами, экземпляр ассоциации
Сигнатура (signature)	Спецификация асинхронного стимула, передаваемого от экземпляра к экземпляру
Синхронное действие (synchronous action)	Имя и параметры характеристики поведения
Система (system)	Запрос, при работу, ожидая результата которому отправивший его объект прерывает
Событие (event)	Набор подсистем, организованный для достижения определенной цели и описываемый набором моделей с разных точек зрения
Сообщение (message)	Определение значимого происшествия, ограниченного во времени и пространстве, в контексте конечных автоматов. Событие может запустить переход из одного состояния в другое состояние
Состояние (state)	Спецификация передачи информации между объектами в ожидании того, что будет обеспечена требуемая деятельность. Получение экземпляра сообщения обычно рассматривается как экземпляр события
	Условия или ситуация в течение жизни объекта, когда он удовлетворяет некоторому условию, выполняет некоторую деятельность или ждет некоторого события

Состояние действия (action state)	Состояние, которое представляет собой исполнение единичного действия, обычно вызов операции
Спецификация (specification)	Текстовая запись синтаксиса и семантики определенного строительного блока, описание того, что он из себя представляет или что он делает
Стереотип (stereotype)	Расширение словаря UML, позволяющее нам создавать новые типы строительных блоков, порождая их от существующих. Новые блоки специализированы для решения определенных проблем
Сторожевое условие (guard condition)	Условие, которое должно быть выполнено для запуска ассоциированного с ним перехода
Супертип (supertype)	В отношении обобщения — обобщение другого типа, подтипа
Сценарий (scenario)	Определенная последовательность действий, иллюстрирующая поведение
Теговая величина (tagged value)	Расширение характеристик элемента UML, позволяющее помещать в спецификацию элемента новую информацию
Тестовая модель (test model)	Определяет тестовые варианты для проверки системы
Тип (type)	Стереотип класса, используемый для определения предметной области объекта и операций (но не методов), применимых к этому объекту
Тип данных (datatype)	Тип, задающий набор неидентифицированных значений и операций для их обработки. Типы данных включают в себя как простые встроенные типы (такие, как числа и строки), так и перечислимые типы (например, логический тип)
Узел (node)	Физический элемент, существующий во время работы системы и предоставляющий вычислительный ресурс, обычно имеющий память, а часто — и возможность выполнения операций
Украшение (adornment)	Детализация спецификации элемента, добавляемая к его основной графической нотации
Фасад (facade)	Фасад — это стереотипный пакет, не содержащий ничего, кроме ссылок на элементы модели, находящиеся в другом пакете. Он используется для обеспечения «публичного» представления некоторой части содержимого пакета
Фокус управления (focus of control)	Символ на диаграмме последовательности, указывающий период времени, в течение которого объект выполняет действие
Характеристика (property)	Именованная величина, обозначающая характеристику элемента
Шаблон (template)	Параметризованный элемент
Экземпляр (instance)	Конкретная реализация абстракции, сущность, к которой может быть применен набор операций, она имеет состояние для сохранения результатов применения операций. Синоним объекта
Экспорт (export)	В контексте пакетов — действие, делающее элемент видимым вне его собственного пространства имен
Элемент (element)	Единичная составная часть модели
Этап Конструирование (Construction phase)	Этап построения программного продукта в виде серии инкрементных итераций
Этап Начало (Inception phase)	Этап спецификации представления продукта

Этап Переход (Transition phase)	Этап внедрения программного продукта в среду пользователя (промышленное производство, доставка и применение)
Этап Развитие (Elaboration phase)	Этап планирования необходимых действий и требуемых ресурсов
n-арная ассоциация (n-ary association)	Ассоциация между n классами. Если n равно двум, ассоциация бинарная. См. <i>бинарная ассоциация</i>
Элемент Use Case (use case)	Описание набора, состоящего из нескольких последовательностей действий системы, которые производят для отдельного актера видимый результат

ПРИЛОЖЕНИЕ В. ОСНОВНЫЕ СРЕДСТВА ЯЗЫКА ПРОГРАММИРОВАНИЯ ADA 95

Ada 95 — современный язык программирования, имеющий максимальный набор средств описания данных и действий. Его средства обеспечивают все технологические потребности профессионального программирования. Конструкции языка поддерживают как традиционный, императивный стиль программирования, так и объектно-ориентированный стиль, позволяют создавать как последовательные, так и параллельные процессы.

Типы и объекты данных

Тип данных задает набор возможных значений и набор операций, допустимых над этими значениями. Все типы данных Ada 95 разделяются на две большие группы: элементарные и составные. Данные элементарного типа имеют значения, которые логически неразделимы. Данные составного типа имеют значения, которые составлены из значений компонентов.

В свою очередь, элементарные типы делятся на скалярные типы (дискретные и вещественные) и ссылочные типы (чью значения являются указателями на данные и подпрограммы). Дискретные типы включают целые типы (знаковые и беззнаковые) и перечисляемые типы. Вещественные типы включают типы с плавающей точкой и типы с фиксированной точкой (двоичные и десятичные).

Составные типы данных подразделяются на комбинированные типы (записи), расширения типа запись, регулярные типы (массивы), задачные типы, защищенные типы. Задачные и защищенные типы используются при программировании параллельных процессов.

Описание типа приводится в декларативной части программы. Общая форма объявления типа имеет вид

```
type <ИмяТипа> is <ОпределениеТипа>;
```

где в угловых скобках указывается название, которое в реальной программе заменяется конкретной конструкцией (именем, выражением, оператором).

Приведем примеры объявления типов:

- целый знаковый тип
type Temperature is range -70..70;
- модульный целый тип
type Time_of_Day is mod 86400;
type Day_of_Month is mod 32;
- вещественный тип с плавающей точкой — задает значения, представляемые восемью десятичными цифрами
type Distance is digits 8;
- двоичный вещественный тип с фиксированной точкой — задает значения с погрешностью 0,001 в диапазоне от 0,00 до 200,00
type Price is delta 0.001 range 0.00 ..200.00;
- десятичный вещественный тип с фиксированной точкой — задает значения, представляемые восемью десятичными цифрами с погрешностью 0,1 (то есть значения до 9999999,9)
type Miles is delta 0.1 digits 8;
- перечисляемый тип
type Day is (mon, tue, wed, thu, fri, sat, sun);
type Colour is (red, blue, green, black);
- тип записи

```

type Date_Type is
record
    Day : Day_Type;
    Month : Month_Day;
    Year : Year_Type;
end record;

```

□ тип массива

```
type Week is array ( 1 .. 7 ) of Day;
```

Некоторые типы в языке предопределены. Предопределенные типы не нужно объявлять в декларативной части программы. К ним относятся:

- целый тип Integer с диапазоном значений -32 767...+32 768;
- вещественный тип с плавающей точкой Float;
- перечисляемые типы Boolean (логический), Character (символьный);
- регулярный тип String (задает массивы из элементов символьного типа).

После того как тип объявлен, можно объявлять экземпляры этого типа. Экземпляры типов называются *объектами*. Объекты содержат значения. Значения объектов-переменных могут изменяться, значения объектов-констант постоянны.

Общая форма объявления объекта имеет вид

```
<ИмяОбъекта> : [constant] <ИмяТипа> [:НачальноеЗначение];
```

где в квадратных скобках указаны необязательные элементы, а НачальноеЗначение — некоторое выражение соответствующего типа.

Примеры объявлений объектов-переменных:

- символьный объект с начальным значением
- ```
Symbol : Character := 'A';
```

### **ПРИМЕЧАНИЕ**

Значение символьного объекта записывается в апострофах.

- строковый объект с начальным значением
- ```
Name : String ( 1 .. 9 ) := "Aleksandr";
```

ПРИМЕЧАНИЕ

Значение строкового объекта записывается в кавычках.

- объект перечисляемого типа
- ```
Car_Colour : Colour := red;
```
- объект модульного типа
- ```
Today : Day_of_Month := 31;
```

ПРИМЕЧАНИЕ

Значение этого объекта может изменяться в диапазоне от 0 до 31. К модльному типу применяется модульная арифметика, поэтому после оператора Today := Today + 1 объект Today получит значение 0.

Примеры объявлений объектов-констант:

```
Time : constant Time_of_Day := 60;
```

```
Best_Colour : constant Colour := blue;
```

Отметим, что если константа является именованным числом (целого и вещественного типа), то имя типа можно не указывать:

```
Minute : constant := 60;
```

```
Hour : constant := 60 * Minute;
```

Текстовый и числовой ввод-вывод

Ada 95 — это язык для разработки программ реального мира, программ, которые могут быть очень большими (включать сотни тысяч операторов). При этом желательно, чтобы отдельный программный файл имел разумно малый размер. Для обеспечения такого ограничения Ada 95 построена на идеи

библиотек и пакетов. В библиотеку, составленную из пакетов, помещаются программные тексты, предназначенные для многократного использования.

Пакеты ввода-вывода

Пакет подключается к программе с помощью указания (спецификатора) контекста, имеющего вид
with <Имя_Пакета>;

Размещение в пакетах процедур ввода-вывода (для величин предопределенных типов) иллюстрирует табл. В.1.

Таблица В.1. Основные пакеты ввода-вывода

Имя пакета	Тип вводимых-выводимых величин
Ada.Text_IO	Character, String
Ada.Integer_Text_IO	Integer
Ada.Float_Text_IO	Float

Для поддержки ввода-вывода величин других типов, определяемых пользователем, используются шаблоны (заготовки) пакетов — родовые пакеты. Родовой пакет перед использованием должен быть настроен на конкретный тип.

Как показано в табл. В.2, родовые пакеты ввода-вывода всегда находятся внутри пакета-библиотеки Ada.Text_IO.

Таблица В.2. Внутренние пакеты из пакета-библиотеки Ada.Text_IO

Имя родового пакета	Нужна настройка на тип
Integer_IO	Любой целый тип со знаком
Float_IO	Любой вещественный тип с плавающей точкой
Enumeration_IO	Любой перечисляемый тип
Fixed_IO	Любой двоичный вещественный тип с фиксированной точкой
Decimal_IO	Любой десятичный вещественный тип с фиксированной точкой
Modular_IO	Любой целый тип без знака

Для обращения к внутренним родовым пакетам используют составные имена, например Ada.Text_IO.Modular_IO, Ada.Text_IO.Enumeration_IO.

Процесс настройки родового пакета называют конкретизацией. В результате конкретизации образуется экземпляр родового пакета. Экземпляр помещается в библиотеку и может быть подключен к любой программе.

Конкретизация задается предложением следующего формата:

```
with <Полное имя родового пакета>;
package <Имя пакета-экземпляра> is
    new <Имя родового пакета> (<Имя типа настройки>);
```

Например, введем перечисляемый тип:

```
Type Summer is ( may, jun, jul, aug );
```

Создадим экземпляр пакета для ввода-вывода величин этого типа:

```
with Ada.Text_IO.Enumeration_IO;
package Summer_IO is new Ada.Text_IO.Enumeration_IO (Summer);
```

Теперь пакет SummerIO может использоваться в любой программе.

Процедуры ввода

Формат вызова процедуры:

```
<ИмяПакета>.Get (<ФактическиеАргументы>);
```

Например, для ввода величины типа Character записывается оператор вызова

```
Ada.Text_IO.Get (Var);
```

В результате переменной Var (типа Character) присваивается значение символа, введенного с клавиатуры. Пробел считается символом, нажатие на клавишу Enter не учитывается.

Еще один пример оператора вызова:

```
Ada.Integer_Text_IO.Get (Var2);
```

В этом случае в переменную Var2 типа Integer заносится строка числовых символов. Все ведущие пробелы и нажатия на клавишу Enter игнорируются. Первым символом, отличным от пробела, может быть знак +, - или цифра. Стока данных прекращается при вводе нечислового символа или нажатии на клавишу Enter.

Процедуры вывода

В операторе вызова можно указывать не только фактические параметры, но и сопоставления формальных и фактических параметров в виде

```
Put (<ФактическийПараметр1>,
      <ФормальныйПараметр3> => <ФактическийПараметр3>, ...);
```

При такой форме порядок записи параметров безразличен.

Например, по оператору вызова

```
Ada.Text_IO.Put ( Item => Var3 )
```

значение переменной Var3 (типа Character) отображается на дисплее, а курсор перемещается в следующую позицию.

По оператору вызова

```
Ada.Integer_Text_IO.Put ( Var4. Width => 4 )
```

на экране отображается значение целой переменной Var4, используя текущие Width позиций (в примере — 4). Если значение (включая знак) занимает меньше, чем Width, позиций, ему предшествует соответствующее количество пробелов. Если значение занимает больше, чем Width, позиций, то используется реальное количество позиций. Если параметр Width пропущен, то используется ширина, заданная компилятором по умолчанию.

Основные операторы

Оператор присваивания

```
<ИмяПеременной> := <Выражение>;
```

предписывает: вычислить значение выражения и присвоить это значение переменной, имя которой указано в левой части.

Условный оператор

```
if <условие 1> then
```

```
    <последовательность операторов 1>
```

```
elsif <условие 2> then
```

```
    <последовательность операторов 2>
```

```
else
```

```
    последовательность операторов 3>
```

```
end if;
```

обеспечивает ветвление — выполнение операторов в зависимости от значения условий.

ПРИМЕЧАНИЕ

Возможны сокращенные формы оператора (отсутствует ветвь elsif, ветвь else).

Оператор выбора позволяет сделать выбор из произвольного количества вариантов, имеет вид

```
case <выражение> is
    when <список выбора 1> =>
        <последовательность операторов 1>
    ...
    when <список выбора n> =>
        <последовательность операторов n>
    when others =>
        <последовательность операторов n+1>
end case;
```

Порядок выполнения оператора:

- 1) вычисляется значение выражения;

- 2) каждый список выбора (от первого до последнего) проверяется на соответствие значению;
- 3) если найдено соответствие, то выполняется соответствующая последовательность операторов, после чего происходит выход из оператора case;
- 4) если не найдено соответствие, то выполняются операторы, указанные после условия when others.

Элементы списка выбора отделяются друг от друга вертикальной чертой ('|') и могут иметь следующий вид:

- <выражение>;
- <выражение n>..<выражение m>.

Примеры:

```
case Number is
    when 1 | 7 => Put ("Is 1 or 7");
    when 5 => Put ("Is 5");
    when 25..100 => Put ("Is number between 25 and 100");
    when others => Put ("Is unknown number");
end case;
case Answer is
    when 'A'..'Z' | 'a'..'z' => Put_Line ("It's a letter!");
    when others => Put_Line ("It's not a letter!")
end case;
```

Оператор блока объединяет последовательность операторов в отдельную структурную единицу, имеет вид

```
declare
    <последовательность объявлений>
begin
    <последовательность операторов>
end;
```

ПРИМЕЧАНИЕ

Объявления из раздела declare действуют только внутри раздела операторов блока.

Пример:

```
declare
    Ch : Character;
begin
    Ch := 'A';
    Put ( Ch );
end;
```

Операторы цикла

Оператор цикла loop

```
loop
    <последовательность операторов 1>
    exit when <условие выхода>
    <последовательность операторов 2>
end loop;
```

служит для организации циклов с заранее неизвестным количеством повторений.

Порядок выполнения.

1. Выполняется последовательность операторов 1.
2. Вычисляется значение условия выхода. Если значение равно True, происходит выход из цикла.
3. Выполняется последовательность операторов 2. Осуществляется переход к пункту 1.

ПРИМЕЧАНИЕ

Операторы тела повторяются, пока условие равно False. В теле должен быть оператор, влияющий на значение условия, иначе цикл будет выполняться бесконечно. В теле цикла возможно использование безусловного оператора выхода exit или условного оператора выхода exit when <условие>.

Пример:

```
Count := 1;
loop
    Ada.Integer_Text_IO.Put ( Count );
    exit when Count = 10;
    Count := Count + 1;
end loop;
```

При выполнении цикла на экран выводится:

1 2 3 4 5 6 7 8 9 10

Аналогичные вычисления можно задать в следующем виде:

```
Count := 1
loop
    Ada.Integer_Text_IO.Put ( Count );
    if Count = 10 then
        exit;
    end if;
    Count := Count + 1;
end loop;
```

Оператор цикла while также позволяет определить цикл с заранее неизвестным количеством повторений, имеет вид

```
while <условие продолжения> loop
    <последовательность операторов>
end loop;
```

Порядок выполнения.

1. Вычисляется значение условия. Если значение равно True, выполняется переход к пункту 2. В противном случае (при значении False) происходит выход из цикла.

2. Выполняются операторы тела цикла. Осуществляется переход к пункту 1.

Таким образом, это цикл с предусловием.

Перечислим характерные особенности оператора while.

1. Операторы тела могут выполняться нуль и более раз.

2. Операторы тела повторяются, пока условие равно True.

3. В теле должен быть оператор, влияющий на значение условия (для исключения бесконечного повторения).

Пример:

```
Count := 1;
loop
    while Count <= 10 loop
        Put ( Count );
        Count := Count + 1;
    end loop;
```

При выполнении цикла на экран выводится:

1 2 3 4 5 6 7 8 9 10

Оператор цикла for обеспечивает организацию циклов с известным количеством повторений.

Используются две формы оператора.

Первая форма оператора for имеет вид:

```
for <параметр цикла> in <дискретный диапазон> loop
    <операторы тела цикла>
end loop;
```

Параметр цикла — это переменная, которая заранее не описывается (в программе). Данная переменная определена только внутри оператора цикла. Параметру цикла последовательно присваиваются значения из дискретного диапазона. Дискретный диапазон всегда записывается в порядке возрастания в виде

min .. max;

Операторы тела повторяются для каждого значения параметра цикла (от минимального до максимального).

Пример:

```
for Count in 1 .. 10 loop
```

```
Put ( Count );
```

```
end loop;
```

При выполнении цикла на экран выводится:

```
1 2 3 4 5 6 7 8 9 10
```

Вторая форма оператора for имеет вид

```
for <параметр цикла> in reverse <дискретный диапазон> loop
```

```
    <операторы тела цикла>
```

```
end loop;
```

Отличие этой формы состоит в том, что значения параметру присваиваются в порядке убывания (от максимального к минимальному). Диапазон же задается по-прежнему, в порядке возрастания.

Пример:

```
for Count in reverse 1 .. 10 loop
```

```
    Put ( Count );
```

```
end loop;
```

При выполнении цикла на экран выводится:

```
1 0 9 8 7 6 5 4 3 2 1
```

ПРИМЕЧАНИЕ

Операторы exit и exit when могут использоваться и в операторах цикла while, for.

Основные программные модули

Ada-программа состоит из одного или нескольких программных модулей. Программным модулем Ada 95 является:

- подпрограмма — определяет действия — подпроцесс (различают две разновидности: процедуру и функцию);
- пакет — определяет набор логически связанных описаний объектов и действий, предназначенных для совместного использования;
- задача — определяет параллельный, асинхронный процесс;
- защищенный модуль — определяет защищенные данные, разделяемые между несколькими задачами;
- родовой модуль — настраиваемая заготовка пакета или подпрограммы.

Родовой модуль имеет формальные родовые параметры, обеспечивающие его настройку в период компиляции. Родовыми параметрами могут быть не только элементы данных (объекты), но и типы, подпрограммы, пакеты. Поэтому общие модули, рассчитанные на использование многих типов данных, следует оформлять как родовые.

Как правило, модули можно компилировать отдельно. Обычно в модуле две части:

- *спецификация* (содержит сведения, видимые из других модулей);
- *тело* (содержит детали реализации, невидимые из других модулей).

Спецификация и тело также могут компилироваться отдельно. Все это дает возможность проектировать, кодировать и тестировать программу как набор слабо зависимых модулей.

Функции

Функция — разновидность подпрограммы, которая возвращает значение результата.

Спецификация функции имеет вид

```
function <ИмяФункции> (<СписокФормальныхПараметров>)  
    return <ТипРезультата>;
```

Список формальных параметров объявляет аргументы, которые принимает функция. Элементы списка отделяются друг от друга точкой с запятой. Каждый элемент (формальный параметр) записывается в виде

```
<ИмяПеременной>:<ТипДанных> := <ЗначениеПоУмолчанию>
```

Значение по умолчанию может не задаваться.

Пример спецификации:

```
function Box_Area (Depth : Float; Width : Float) return Float;
```

Тело функции включает спецификацию функции, объявления локальных переменных и констант, а

также раздел исполняемых операторов. В общем случае тело функции имеет вид

```
function <ИмяФункции> (<СписокФормальныхПараметров>)
    return <ТипРезультата> is
<объявления локальных переменных и констант>
begin
    <операторы>
    return <результат>; -- оператор возврата результата
end <ИмяФункции>;
Пример тела функции:
function Box_Area (Depth : Float; Width ; Float) return Float is
    Result : Float;
begin
    Result := Depth * Width;
    return Result; -- возврат вычисленного значения
end Box_Area;
```

Описание тела функции само по себе действий не производит. Для выполнения функции необходимо ее вызвать. Чтобы вызвать функцию, записывают ее имя и список фактических параметров, запись помещается в правую часть оператора присваивания:

<ИмяПеременной> := <ИмяФункции> (<СписокФактическихПараметров>);

Таким образом, вызов функции является элементом выражения. Фактические параметры в списке вызова отделяются друг от друга запятой.

Пример вызова:

My_Box := Box_Area (2.0, 4.15);

Фактические параметры задают фактические значения, то есть значения, обрабатываемые при выполнении функции.

Процедуры

Процедуры, в отличие от функций, не возвращают результат в точку вызова. Спецификация процедуры задает минимальный набор сведений, необходимый для клиентов процедуры. Она имеет вид

procedure <ИмяПроцедуры> (<СписокФормальныхПараметров>);

Для записи каждого формального параметра принят следующий формат:

<Имя> : <Вид> <Тип данных>;

где <Вид> указывает направление передачи информации между формальным и фактическим параметрами (in — передача из фактического в формальный параметр, out — из формального в фактический параметр, in out — двунаправленная передача).

ПРИМЕЧАНИЕ

Пометку in разрешается не указывать (она подразумевается по умолчанию), поэтому в спецификации функции вид параметра отсутствует. Для формального параметра вида in разрешается задавать начальное значение, присваиваемое по умолчанию.

Пример спецификации:

```
procedure Sum ( Op1 : in Integer := 0; Op2 : in Integer := 0;
    Op3 : in Integer := 0; Res : out Integer );
```

Тело процедуры в общем случае имеет вид

```
procedure <ИмяПроцедуры>
    (<СписокФормальныхПараметров>) is
<объявления локальных переменных и констант>
begin
    <операторы>
end <ИмяПроцедуры>;
```

Пример тела:

```
procedure Sum ( Op1 : in Integer := 0; Op2 : in Integer := 0;
    Op3 : in Integer := 0; Res : out Integer ) is
begin
    Res := Op1 + Op2;
```

```
Res := Res + Op3;
end Sum;
```

В данной процедуре три формальных параметра имеют значения по умолчанию. Это дает интересные возможности.

Обращаются к процедуре с помощью оператора вызова, он имеет вид
 <ИмяПроцедуры> (<СписокФактическихПараметров>);

Примеры операторов вызова:

```
Sum (4. 8, 12. d); -- переменная d получит значение 24
Sum (4. 8. Res => d); -- переменная d получит значение 12
```

ПРИМЕЧАНИЕ

В первом операторе вызова задано 4 фактических параметра, во втором операторе — 3 фактических параметра. Во втором операторе использованы как традиционная (позиционная) схема, так и именная схема сопоставления формального и фактического параметров.

Пакеты

Пакет — основное средство для поддержки многократности использования программного текста. При проектировании программ пакеты позволяют применить подход клиент-сервер. Пакет действует как сервер, который предоставляет своим клиентам (программам и другим пакетам) набор услуг.

Спецификация пакета объявляет предлагаемые услуги, а тело содержит реализацию этих услуг.

Спецификация пакета записывается в виде

```
package <ИмяПакета> is
  <объявления типов, переменных, констант>
  <спецификации процедур и функций>
end <ИмяПакета>;
```

Пример спецификации:

```
package Рисование is
  type Точка is array ( 1 .. 2 ) of Integer;
  -- описание точки в прямоугольной системе координат
  procedure Переход ( из : in Точка; в : in Точка );
  -- переход из одной точки в другую точку
  procedure Рисовать_Линию ( от : in Точка; до : in Точка );
  -- рисуется сплошная линия между заданными точками
  procedure Рисовать_Пунктирную_Линию ( от : in Точка; до ; in Точка );
  -- рисуется пунктирная линия
end Рисование;
```

Данная спецификация предлагает клиентам один тип данных и три процедуры.

Тело пакета представляется в виде

```
package body <ИмяПакета> is
  <объявления локальных переменных, констант, типов>
  <тела процедур и функций>
end <ИмяПакета>;
```

Еще раз отметим, что содержание тела пакета клиентам недоступно.

Пример тела:

```
package body Рисование is
  -- локальные объявления
  procedure Переход ( из : in Точка; в : in Точка ) is
    -- локальные объявления
  begin
    -- операторы
  end Переход;
  procedure Рисовать_Линию(от : in Точка: до ; in Точка) is
    -- локальные объявления
  begin
    -- операторы
  end Рисовать_Линию;
  procedure Рисовать_Пунктирную_Линию ( от : in Точка;
```

```

до : in Точка ) is
-- локальные объявления
begin
    -- операторы
end Рисовать_Пунктирную_Линию;
end Рисование;

```

В спецификации пакета может быть полузакрытая (приватная) часть. Эта часть отделяется от обычной (открытой) части служебным словом `private`. Содержимое приватной части пользователю (клиенту) недоступно. В эту часть помещают скрываемые от пользователя операции и детали описания типов данных. Заметим, что из тела пакета доступно содержание как открытой, так и приватной части спецификации.

Производные типы

Объявление производного типа имеет вид

```

type <ИмяПроизводногоТипа> is
    new <ИмяРодительскогоТипа> [<ОграничениеРодительскогоТипа>];

```

где ограничение на значения родительского типа могут отсутствовать.

Производный тип наследует у родительского-типа его значения и операции. Набор родительских значений наследуется без права изменения. Наследуемые операции, называемые примитивными операциями, являются подпрограммами, имеющими формальный параметр или результат родительского типа и объявленными в том же пакете, что и родительский тип.

В заголовке каждой из унаследованных операций выполняется автоматическая замена указаний родительского типа на указания производного типа.

Например, пусть сделаны объявления:

```

type Integer is ...; -- определяется реализацией
function "+" ( Left, Right : Integer ) return Integer;

```

Тогда любой тип, производный от `Integer`, вместе с реализацией родительского типа автоматически наследует функцию `<+>`:

```

type Length is new Integer;
-- function "+" ( Left, Right : Length ) return Length;

```

Здесь символ комментария `(--)` показывает, что операция `<+>` наследуется автоматически, то есть от программиста не требуется ее явное объявление. Любая из унаследованных операций может быть переопределена, то есть может быть обеспечена ее новая реализация. О такой операции говорят, что она перекрыта:

```

type Угол is new Integer;
function "+" ( Left, Right : Угол ) return Угол;

```

В этом примере для функции `<+>` обеспечена новая реализация (учитывается модульная сущность сложения углов).

По своей сути производный тип — это новый тип данных со своим набором значений и операций, а также со своей содержательной ролью. По значениям, операциям производный тип несовместим ни с родительским типом, ни с любым другим типом, производным от этого же родителя.

По сравнению с родительским типом в производном типе:

- набор значений может быть сужен (за счет ограничений при объявлении);
- набор операций может быть расширен (за счет объявления операций в определяющем для производного типа пакете).

Примеры объявления производных типов:

```

type Год Is new Integer range 0 .. 2099;
type Этаж i s new Integer range 1 .. 100;

```

Если теперь мы введем два объекта:

```

A : Год;
B : Этаж;

```

и попытаемся выполнить присваивание

```

A := B;

```

то будет зафиксирована ошибка.

Подтипы

Очень часто для повышения надежности программы приходится ограничивать область значений типов и объектов, не затрагивая при этом допустимые операции. Для такого ограничения удобно использовать понятие подтипа.

Подтип — это сочетание типа и ограничения на допустимые значения этого типа. Объявление подтипа имеет вид

```
subtype <ИмяПодтипа> is <ИмяТипа> range <Ограничение>;
```

Характерные особенности подтипов:

- подтип наследует все операции, которые определены для его типа;
- объект подтипа совместим с любым объектом его типа, удовлетворяющим указанному ограничению;
- содержательные роли объектов различных подтипов для одного типа аналогичны.

Таким образом, объекты типа и его подтипов могут свободно смешиваться в арифметических операциях, операциях сравнения и присваивания.

Например, если в программе объявлен перечисляемый тип День_Недели, то можно объявить подтип

```
subtype Рабочий_День is День_Недели range ПОНЕДЕЛЬНИК..ПЯТНИЦА;
```

При этом гарантируется, что объекты подтипа Рабочий_День будут совместимы с объектами типа День_Недели.

Расширяемые типы

Основная цель расширяемых типов — обеспечить повторное использование существующих программных элементов (без необходимости перекомпиляции и перепроверки). Они позволяют объявить новый тип, который уточняет существующий родительский тип наследованием, изменением или добавлением как существующих компонентов, так и операций родительского типа. Иначе говоря, идея расширяемого типа — это развитие идеи производного типа. В качестве расширяемых типов используются теговые типы (разновидность комбинированного типа).

Рассмотрим построение иерархии геометрических объектов. На вершине иерархии точка, имеющая два свойства (координаты X и Y):

```
type Точка Is tagged
  record
    X_KoopD : Float;
    Y_Koopfl : Float;
  end record;
```

Другие типы объектов можно произвести (прямо или косвенно) от этого типа.

Например, можно ввести новый тип, наследник точки:

```
type Окружность is new Точка with -- новый теговый тип;
  record
    Радиус : Float;
  end record;
```

Данный тип имеет три свойства: два свойства (координаты X и Y) унаследованы от типа Точка, а третье свойство (Радиус) нами добавлено. Дочерний тип Окружность наследует все операции родительского типа Точка, причем некоторые операции могут быть переопределены. Кроме того, для дочернего типа могут быть введены новые операции.

СПИСОК ЛИТЕРАТУРЫ

1. Боэм Б. У. Инженерное проектирование программного обеспечения. М.: Радио и связь, 1985. 511 с.
2. Липаев В. В. Отладка сложных программ: Методы, средства, технология. М.: Энергоатомиздат, 1993. 384 с.
3. Майерс Г. Искусство тестирования программ. М.: Финансы и статистика, 1982. 176с.
4. Орлов С. А. Принципы объектно-ориентированного и параллельного программирования на языке Ada 95. Рига: TSI, 2001. 327 с.
5. Чеппел Д. Технологии ActiveX и OLE. М.: Русская редакция, 1997. 320 с.
6. 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.
7. Albrecht, A. J. Measuring Application Development Productivity. Proc. IBM Application Development Symposium, Oct. 1979, pp. 83-92.
 8. Ambler, S. W. The Object Primer. 2nd ed. Cambridge University Press, 2001. 541 pp.
 9. Beck, K., and Cunningham, W. A Laboratory for Teaching Object-oriented Thinking. SIGPLAN Notices vol. 24 (10), October 1989, pp 1-7.
 10. Beck, K. Embracing Change with Extreme Programming. IEEE Computer, Vol. 32, No. 10, October 1999, pp. 70-77.
 11. Beck, K. Extreme Programming Explained. Embrace Change. Addison-Wesley, 1999. 211pp.
 12. Beck, K, Fowler, M. Planning Extreme Programming. Addison-Wesley, 2001. 156pp.
 13. Beizer, B. Software Testing Techniques, 2nd ed. New York: International Thomson Computer Press, 1990. 503 pp.
 14. Beizer, B. Black-Box Testing: Techniques for Functional Testing of Software and Systems. New York: John Wiley & Sons, 1995. 320 pp.
 15. 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.
 16. Binder, R. V. Testing object-oriented systems: a status report. American Programmer 7 (4), April 1994, pp. 22-28.
 17. Binder, R. V. Design for Testability in Object-Oriented Systems. Communications of the ACM, vol. 37, No 9, September 1994, pp. 87-101.
 18. Binder, R. V. Testing Object-Oriented Systems. Models, Patterns, and Tools. Addison-Wesley, 1999. 1298 pp.
 19. Boehm, B. W. A spiral model of software development and enhancement. *IEEE Computer*, 21 (5), 1988, pp. 61-72.
 20. Boehm, B. W. Software Risk Management: Principles and Practices. *IEEE Software*, January 1991: pp. 32-41.
 21. Boehm, B. W. *et al.* Software Cost Estimation with Cocomo II. Prentice Hall, 2001. 502 pp.
 22. Booch, G. Object-Oriented analysis and design. 2nd Edition. Addison-Wesley, 1994. 590 pp.
 23. Booch, G., Rumbaugh, J., Jacobson, I. The Unified Modeling Language User Guide. Addison-Wesley, 1999. 483 pp.
 24. Chidamber, S. R. and Kemerer, C. F. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, vol. 20: 476-493. No. 6, June 1994.
 25. Cockburn, A. Agile Software Development. Addison-Wesley, 2001. 220 pp.
 26. Coplien, J. O. Multi-Paradigm Design for C++. Addison-Wesley, 1999. 297 pp.
 27. DeMarco, T.. Structured Analysis and System Specification. Englewood Cliffs, NJ: Prentice-Hall, 1979.
 28. Fenton, N. E., Pfleeger S. L. Software Metrics: A Rigorous & Practical Approach. 2nd Edition. International Thomson Computer Press, 1997. 647 pp.
 29. Fowler, M. The New Methodology <http://www.martinfowler.com>, 2001.
 30. Fowler, M. Is Design Dead? Proceedings of the XP 2000 conference, the Mediterranean island of Sardinia, 11 pp., June 2000.
 31. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995. 410 pp.
 32. Graham, I. Object-Oriented Methods. Principles & Practice. 3rd Edition. Addison-Wesley, 2001. 853 pp.
 33. Halstead, M. H. Elements of Software Science. New York, Elsevier North-Holland, 1977.
 34. Hatley, D., and Pirbhai, I. Strategies for Real-Time System Specification. New York, NY: Dorset House, 1988.
 35. 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.
 36. Highsmith, J. A. Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. Dorset House Publishing, 2000. 392 pp.
 37. Highsmith, J. A. Extreme programming, e-business Application Delivery, vol. XII, No. 2; February 2000, pp 1-16.
 38. Hitz, M., Montazeri, B. Measuring Coupling in Object-Oriented Systems. *Object Currents*, vol. 2: 17 pp., No 4, Apr 1996.
 39. Jackson, M. A. Principles of Program Design. London: Academic Press, 1975.

40. Jacobcon, I., Booch, G., Rumbaugh, J. The Unified Software Development Process. Addison-Wesley, 1999. 463 pp.
41. Jacobcon, I., Christerson, M., Jonsson, P., Overgaard, G. J. Object-Oriented Software Engineering. Addison-Wesley, 1993. 528 pp.
42. Jorgensen, P. C. and Erickson, C. Object Oriented Integration. Communications of the ACM, vol. 37, No 9, September 1994, pp. 30-38.
43. 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.
44. Kruchten, Phillip B. The 4+1 View Model of Architecture. IEEE Software, Vol. 12 (6), November 1995, pp. 42-50.
45. Lorenz, M. and Kidd, J. Object-Oriented Software Metrics. Prentice Hall, 1994. 146pp.
46. Marick, B. Notes on Object-Oriented Testing. Part 1: Fault-Based Test Design. Testing Foundations Inc., 1995. 7 pp.
47. Marick, B. Notes on Object-Oriented Testing Part 2: Scenario-Based Test Design. Testing Foundations Inc., 1995. 4 pp.
48. Martin, Robert C. RUP/XP Guidelines: Test-first Design and Refactoring. Rational Software White Paper, 2000.
49. McCabe, T. J. A Complexity Measure. IEEE Transactions on Software Engineering, vol. 2: pp. 308-320. No.4, Apr 1976.
50. 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.
51. McGregor, J. D., Sykes, D. A. A Practical Guide to Testing Object-Oriented Software. Addison-Wesley, 2001. 407 pp.
52. Myers, G. Composite Structured Design. New York, NY: Van Nostrand Reinhold, 1978.
53. OMG Unified Modeling Language Specification. Version 1.4. Object Management Group, Inc., 2001.566pp.
54. Orr, K. T. Structured Systems Analysis. Englewood Cliffs, NJ: Yourdon Press, 1977.
55. 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.
56. Oviedo, E. I. Control Flow, Data Flow and Program Complexity. Proc. IEEE COMPSAC, Nov. 1980, pp. 146-152.
57. Quatrani, T. Visual Modeling with Rational Rose and UML. Addison-Wesley, 1998. 222pp.
58. Page-Jones, M. The Practical Guide to Structured Systems Design. Englewood Cliffs, NY: Yourdon Press, 1988.
59. Page-Jones, M. Fundamentals of Object-Oriented Design in UML. Addison - Wesley, 2001. 479 pp.
60. 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.
61. 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.
62. Paulk, M. C. Extreme Programming from a CMM Perspective. XP Universe, Raleigh, NC, 23-25 July 2001, 8 pp.
63. Poston, R. M. Automated Testing from Object Models. Communications of the ACM, vol. 37, No 9, September 1994, pp. 48-58.
64. Pressman, R. S. Software Engineering: A Practitioner's Approach. 5th ed. McGraw-Hill, 2000. 943 pp.
65. Royce, Walker W. Managing the development of large software systems: concepts and techniques. *Proc. IEEE WESTCON, Los Angeles*, August 1970, pp. 1-9.
66. Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W. Object Oriented Modeling and Design. Prentice Hall, 1991. 500 pp.
67. Rumbaugh, J., Jacobcon, I., Booch, G., The Unified Modeling Language Reference Manual. Addison-Wesley, 1999. 567 pp.
68. Shalloway, A., Trott, J. R. Design Patterns Explained. A New Perspective on Object-Oriented Design. Addison - Wesley, 2002. 361 pp.
69. Sommerville, I. Software Engineering. 6th ed. Addison-Wesley, 2001. 713 pp.
70. Stevens, W., Myers, G., and Constantine, L. 1979. Structured Design. IBM Systems Journal, Vol. 13(2),

- 1974, pp. 115-139.
- 71. Vliet, J. C. van. Software Engineering: Principles and Practice. John Wiley & Sons, 1993.558pp.
 - 72. Tai, K., and Su, H. Test Generation for Boolean Expressions. Proc. COMPSAC'87, October 1987, pp. 278-283.
 - 73. 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.
 - 74. Warnier, J. D. Logical Construction of Programs. New York: Van Nostrand Reinhold, 1974.
 - 75. Wells, J. D. Extreme Programming: A gentle introduction, <http://www.extreme-programming.org>, 2001.
 - 76. Wirfs-Brock, R., Wilkerson, B., and Wiener, L. Designing Object-oriented Software. Englewood Cliffs, New Jersey: Prentice Hall, 1990. 341 pp.
 - 77. Yourdon, E., and Constantine, L. Structured Design: fundamentals of a discipline of computer program and systems design. Englewood Cliffs, NJ: Prentice-Hall, 1979.

Оглавление

Введение	3
От издательства.....	5
ГЛАВА 1. Организация процесса конструирования.....	5
Определение технологии конструирования программного обеспечения.....	5
Классический жизненный цикл	6
Макетирование	7
Стратегии конструирования ПО.....	8
Инкрементная модель.....	9
Быстрая разработка приложений.....	9
Спиральная модель	10
Компонентно-ориентированная модель.....	11
Тяжеловесные и облегченные процессы.....	12
XP-процесс	13
Модели качества процессов конструирования.....	16
Контрольные вопросы	18
ГЛАВА 2. Руководство программным проектом	18
Процесс руководства проектом	19
Начало проекта.....	19
Измерения, меры и метрики.....	19
Процесс оценки	19
Анализ риска	20
Планирование	20
Трассировка и контроль	20
Планирование проектных задач	20
Размерно-ориентированные метрики.....	21
Функционально-ориентированные метрики.....	22
Выполнение оценки в ходе руководства проектом	27
Выполнение оценки проекта на основе LOC- и FP-метрик	28
Конструктивная модель стоимости	28
Модель композиции приложения	29
Модель раннего этапа проектирования	30
Модель этапа постархитектуры	33
Предварительная оценка программного проекта.....	35
Анализ чувствительности программного проекта	37
Сценарий понижения зарплаты	38
Сценарий наращивания памяти	38
Сценарий использования нового микропроцессора	39
Сценарий уменьшения средств на завершение проекта	39
Контрольные вопросы	40
ГЛАВА 3. Классические методы анализа	41
Структурный анализ	41
Диаграммы потоков данных	41
Описание потоков данных и процессов	42
Расширения для систем реального времени	43
Расширение возможностей управления	44
Модель системы регулирования давления космического корабля.....	45
Методы анализа, ориентированные на структуры данных	47
Метод анализа Джексона	48
Методика Джексона.....	48
Шаг объект-действие	48
Шаг объект-структура	49
Шаг начального моделирования	50
Контрольные вопросы	52
ГЛАВА 4. Основы проектирования программных систем	52
Особенности процесса синтеза программных систем	52
Особенности этапа проектирования.....	53
Структурирование системы	54
Моделирование управления	55
Декомпозиция подсистем на модули	56
Модульность.....	57
Информационная закрытость.....	57
Связность модуля.....	58
Функциональная связность	59
Информационная связность	59

Коммуникативная связность	60
Процедурная связность	60
Временная связность	61
Логическая связность	61
Связность по совпадению	62
Определение связности модуля	62
Сцепление модулей.....	63
Сложность программной системы.....	63
Характеристики иерархической структуры программной системы.....	64
Контрольные вопросы	65
ГЛАВА 5. Классические методы проектирования.....	66
Метод структурного проектирования	66
Типы информационных потоков	66
Проектирование для потока данных типа «преобразование».....	67
Проектирование для потока данных типа «запрос»	69
Метод проектирования Джексона	70
Доопределение функций	70
Учет системного времени.....	72
Контрольные вопросы	73
ГЛАВА 6. Структурное тестирование программного обеспечения	73
Основные понятия и принципы тестирования ПО	73
Тестирование «черного ящика»	75
Тестирование «белого ящика»	75
Особенности тестирования «белого ящика»	75
Способ тестирования базового пути	76
Потоковый граф	76
Цикломатическая сложность	78
Шаги способа тестирования базового пути	78
Способы тестирования условий.....	80
Тестирование ветвей и операторов отношений	81
Способ тестирования потоков данных	83
Тестирование циклов	84
Простые циклы.....	85
Вложенные циклы.....	85
Объединенные циклы	86
Неструктурированные циклы	86
Контрольные вопросы	86
ГЛАВА 7. Функциональное тестирование программного обеспечения	87
Особенности тестирования «черного ящика»	87
Способ разбиения по эквивалентности	88
Способ анализа граничных значений	89
Способ диаграмм причин-следствий.....	92
Контрольные вопросы	95
ГЛАВА 8. Организация процесса тестирования программного обеспечения.....	95
Методика тестирования программных систем	95
Тестирование элементов	96
Тестирование интеграции	99
Нисходящее тестирование интеграции	100
Восходящее тестирование интеграции	101
Сравнение нисходящего и восходящего тестирования интеграции.....	102
Тестирование правильности.....	102
Системное тестирование	103
Тестирование восстановления	103
Тестирование безопасности	104
Стрессовое тестирование	104
Тестирование производительности	104
Искусство отладки	105
Контрольные вопросы	106
ГЛАВА 9. Основы объектно-ориентированного представления программных систем	107
Принципы объектно-ориентированного представления программных систем	107
Абстрагирование	107
Инкапсуляция	108
Модульность.....	109
Иерархическая организация	109
Объекты	111
Общая характеристика объектов	111

Виды отношений между объектами	113
Связи	113
Видимость объектов	115
Агрегация	115
Классы	116
Общая характеристика классов	116
Виды отношений между классами	117
Ассоциации классов.....	118
Наследование.....	119
Полиморфизм	120
Агрегация.....	121
Зависимость	122
Конкретизация.....	122
Контрольные вопросы	123
ГЛАВА 10. Базис языка визуального моделирования.....	124
Унифицированный язык моделирования.....	124
Предметы в UML	125
Отношения в UML	127
Диаграммы в UML	128
Механизмы расширения в UML	129
Контрольные вопросы	131
ГЛАВА 11. Статические модели объектно-ориентированных программных систем	131
Вершины в диаграммах классов	131
Свойства.....	132
Операции	132
Организация свойств и операций	133
Множественность.....	133
Отношения в диаграммах классов.....	134
Деревья наследования.....	137
Примеры диаграмм классов	138
Контрольные вопросы	140
ГЛАВА 12. Динамические модели объектно-ориентированных программных систем	141
Моделирование поведения программной системы.....	141
Диаграммы схем состояний	141
Действия в состояниях	143
Условные переходы	143
Вложенные состояния	143
Диаграммы деятельности	144
Диаграммы взаимодействия.....	146
Диаграммы сотрудничества	146
Диаграммы последовательности	149
Диаграммы Use Case	151
Актеры и элементы Use Case	151
Отношения в диаграммах Use Case	152
Работа с элементами Use Case	154
Спецификация элементов Use Case	154
Главный поток.....	154
Подпотоки.....	154
Альтернативные потоки	155
Пример диаграммы Use Case	155
Построение модели требований	158
Кооперации и паттерны	161
Паттерн Наблюдатель.....	163
Паттерн Компоновщик	165
Паттерн Команда.....	167
Бизнес-модели	168
Контрольные вопросы	169
ГЛАВА 13. Модели реализации объектно-ориентированных программных систем	170
Компонентные диаграммы.....	170
Компоненты.....	171
Интерфейсы	172
Компоновка системы	172
Разновидности компонентов	173
Использование компонентных диаграмм	174
Моделирование программного текста системы	174
Моделирование реализации системы.....	175

Основы компонентной объектной модели	177
Организация интерфейса СОМ	178
Unknown — базовый интерфейс СОМ	180
Серверы СОМ-объектов	181
Преимущества СОМ	181
Работа с СОМ-объектами	182
Создание СОМ-объектов	182
Повторное использование СОМ-объектов	183
Маршалинг	184
IDL-описаниеи библиотека типа	185
Диаграммы размещения	186
Узлы	186
Использование диаграмм размещения	187
Контрольные вопросы	189
ГЛАВА 14. Метрики объектно-ориентированных программных систем.	190
Метрические особенности объектно-ориентированных программных систем	190
Локализация	190
Инкапсуляция	190
Информационная закрытость	191
Наследование	191
Абстракция	191
Эволюция мер связи для объектно-ориентированных программных систем	191
Связность объектов	191
Сцепление объектов	196
Набор метрик Чидамбера и Кемерера	197
Использование метрик Чидамбера-Кемерера	201
Метрики Лоренца и Кидда	202
Метрики, ориентированные на классы	202
Операционно-ориентированные метрики	204
Метрики для ОО-проектов	205
Набор метрик Фернандо Абреу	205
Метрики для объектно-ориентированного тестирования	209
Метрики инкапсуляции	209
Метрики наследования	209
Метрики полиморфизма	210
Контрольные вопросы	210
ГЛАВА 15. Унифицированный процесс разработки объектно-ориентированных ПС	211
Эволюционно-инкрементная организация жизненного цикла разработки	211
Этапы и итерации	212
Рабочие потоки процесса	212
Модели	213
Технические артефакты	213
Управление риском	214
Идентификация риска	214
Анализ риска	215
Ранжирование риска	215
Планирование управления риском	215
Разрешение и наблюдение риска	216
Этапы унифицированного процесса разработки	217
Этап НАЧАЛО (Inception)	217
Этап РАЗВИТИЕ (Elaboration)	217
Этап КОНСТРУИРОВАНИЕ (Construction)	219
Этап ПЕРЕХОД (Transition)	219
Оценка качества проектирования	219
Пример объектно-ориентированной разработки	220
Этап НАЧАЛО	220
Этап РАЗВИТИЕ	221
Этап КОНСТРУИРОВАНИЕ	225
Разработка в стиле экстремального программирования	231
XP-реализация	231
XP-итерация	233
Элемент XP-разработки	233
Коллективное владение кодом	234
Взаимодействие с заказчиком	235
Стоимость изменения и проектирование	236
Контрольные вопросы	238

ГЛАВА 16. Объектно-ориентированное тестирование	239
Расширение области применения объектно-ориентированного тестирования	239
Изменение методики при объектно-ориентированном тестировании	240
Особенности тестирования объектно-ориентированных «модулей».....	240
Тестирование объектно-ориентированной интеграции.....	241
Объектно-ориентированное тестирование правильности	241
Проектирование объектно-ориентированных тестовых вариантов	241
Тестирование, основанное на ошибках	242
Тестирование, основанное на сценариях	243
Тестирование поверхностной и глубинной структуры	244
Способы тестирования содержания класса	244
Стохастическое тестирование класса.....	245
Тестирование разбиений на уровне классов	245
Способы тестирования взаимодействия классов	246
Стохастическое тестирование	247
Тестирование разбиений	248
Тестирование на основе состояний	248
Предваряющее тестирование при экстремальной разработке	249
Контрольные вопросы	263
ГЛАВА 17. Автоматизация конструирования визуальной модели программной системы	263
Общая характеристика CASE-системы Rational Rose	264
Создание диаграммы Use Case.....	266
Создание диаграммы последовательности	269
Создание диаграммы классов	273
Создание компонентной диаграммы	280
Генерация программного кода.....	282
Заключение	290
Приложение А	291
Факторы затрат постархитектурной модели СОСМО II	291
Приложение Б. Терминология языка UML и унифицированного процесса.....	296
Приложение В. Основные средства языка программирования Ada 95	303
Список литературы	313

Орлов Сергей Александрович
Технологии разработки программного обеспечения:
Учебник

Главный редактор *E. Строганова*
Заведующий редакцией *И. Корнеев*
Руководитель проекта *A. Васильев*
Литературный редактор *E. Ваулина*
Художник *Н. Биржаков*
Корректоры *H. Пронина, H. Роцина*
Верстка *A. Келле-Пелле*

Лицензия ИД № 05784 от 07.09.01.

Подписано в печать 22.08.02. Формат 70x100/16. Усл. п. л. 37,41.

Тираж 4500 экз. Заказ № 1221.

ООО «Питер Принт». 196105, Санкт-Петербург,
ул. Благодатная, д. 67в.

Налоговая льгота - общероссийский классификатор
продукции ОК 005-93, том2; 953005 - литература учебная.

Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.