Учебники НГТУ

Серия основана в 2001 году

Е. Л. РОМАНОВ

ПРОГРАММНАЯ ИНЖЕНЕРИЯ

НОВОСИБИРСК
2017

УДК 004.41 (075.8)
Р693

Рецензенты:
д-р техн. наук, профессор, заведующий кафедрой
компьютерных технологий НГУ *О.И. Потатуркин*
д-р техн. наук, доцент, заведующий кафедрой
вычислительных систем СибГУТИ *С.Н. Мамойленко*

Романов Е.Л.
Р 693 Программная инженерия: учебное пособие / Е.Л. Романов. – Новосибирск: Изд-во НГТУ, 2017. – 395 с.; илл. – (Серия «Учебники НГТУ»).

ISBN 978-5-7782-3455-0

Программный код – основная «материальная ценность» любого проекта, а программисты – его непосредственные производители. Поэтому взаимопонимание между ними и остальными участниками процесса производства программного продукта, а также качество кода являются определяющими факторами успеха проекта в целом.

В предлагаемом издании основы программной инженерии, структура жизненного цикла проекта, технологические дисциплины и их артефакты рассматриваются с точки зрения взаимосвязи с создаваемым кодом. Излагается содержание методологий гибкого проектирования и унифицированного процесса, идеи разработки проекта «от кода», базис грамотного программирования: эффективность алгоритмов, ООП, шаблоны проектирования, метрика кода, тестирование.

Программная архитектура рассматривается начиная от конкретики клиент-серверных приложений и прикладных протоколов и заканчивая общими вопросами проектирования и преодоления «архитектурной пропасти» между функционалом и реализацией.

Учебник рекомендуется студентам, обучающимся по направлениям, связанным с информационными технологиями, как изложение основ программной инженерии; будущим программистам – для понимания роли и места профессии в процессе разработки и представления о смежных видах деятельности, которыми при необходимости придется заниматься по совместительству; всем остальным – для понимания специфики разработки качественного кода, особенностей процесса программирования, его организации и контроля.

УДК 004.41 (075.8)

ISBN 978-5-7782-3455-0

ОГЛАВЛЕНИЕ

С благодарностью моему
Учителю и наставнику
Владимиру Константиновичу Шмидту

ВВЕДЕНИЕ

Мы все учились понемногу
Чему-нибудь и как-нибудь.

А.С. Пушкин. «Евгений Онегин»

Программная инженерия. Кому и почему важен «взгляд от кода»?

Программная инженерия – дитя «нулевых» годов нашего века. Производство программного обеспечения превратилось в отдельную отрасль со своими стандартами, разделением труда и производственным циклом. В профессиональных стандартах [1] можно найти множество профессий, так или иначе связанных с программным обеспечением (ПО). Их можно разделить на группы в зависимости от рода деятельности:

- распространение, использование, обслуживание программных продуктов, касающееся исключительно их потребительских свойств: менеджер по ИТ, менеджер продуктов в области ИТ, специалист по технической поддержке;
- поддержка инфраструктуры, информационное наполнение и сборка информационных систем из готовых компонент: специалист по информационным системам, администратор БД, сетевой администратор, системный администратор, специалист по информационным ресурсам;
- вспомогательные и второстепенные профессии, связанные с жизненным циклом ПО: специалист по тестированию, технический писатель, специалист по дизайну графического и пользовательского интерфейса;
- профессии, связанные собственно с производством программного продукта: системный программист, программист, системный аналитик, архитектор ПО, руководитель разработки.

В системе высшего образования также имеет место многообразие направлений подготовки, связанных с информационными технологиями, разработкой информационных систем и программированием:

- бизнес-информатика;
- математическое обеспечение и администрирование информационных систем;
- прикладная информатика;
- прикладная математика и информатика;
- информационные системы и технологии;
- информатика и вычислительная техника;
- программная инженерия.

Между направлениями подготовки и профессиями нет однозначного соответствия: нигде не выпускают непосредственно программистов или архитекторов ПО. Значительную часть занимает общее образование, остальное – понемногу обо всем, ориентируясь на пару основных профессий и затрагивая несколько вспомогательных в отдельных дисциплинах.

Предлагаемый материал может быть полезен для различной целевой аудитории:

- для всех направлений подготовки – как изложение основ программной инженерии и основных ее артефактов;
- будущим программистам – для понимания роли и места профессии в процессе разработки в больших проектах, представления о смежных видах деятельности, которыми по необходимости придется заниматься по совместительству;
- всем остальным – для понимания специфики разработки качественного кода, особенностей процесса программирования, его организации и контроля. В этом случае материал по программированию (гл. 3, 4, 8) можно читать по диагонали, фиксируя только ключевые моменты.

Программист: творец или рабочая лошадка?

Среди перечисленного многообразия профессия программиста как-то затерялась, а роль ее оценивается обществом весьма противоречиво:

- программист – непосредственный создатель материальных благ в этой отрасли. Как бы ни были важны спецификации, модели, документы, без кода, который исполняет задуманное, они бесполезны;
- программист – рабочая лошадка, ремесленник, формальный исполнитель требований, спецификаций и задач, поставленных остальными участниками проекта.

Очевидно, истина находится где-то посередине либо вообще плавает по всему диапазону от проекта к проекту. К тому же относительно профессии программиста в самом обществе сложилось немало стереотипов, живучесть которых объясняется сложностью и неочевидностью оценки не только качества и количества труда программиста, но иногда даже его результатов.

Самый последний по времени общественный стереотип – **«тыжпрограммист»** – программист отвечает за всех и за всё, игнорирует сам факт разделения труда в отрасли. Но, с другой стороны, доля истины в нем есть. При разработке кода могут возникать проблемы, корни которых уходят глубоко в аппаратные средства и архитектуру, поэтому хороший программист – это все-таки «тыжпрограммист».

Другой интенсивно эксплуатируемый миф о якобы исключительно творческом, креативном характере труда программиста. Корни его уходят в далекие 50–60-е гг. прошлого века, когда компьютеры были сродни синхрофазотронам, а работа с ними – среднее между священнодействием и трудом физика-теоретика. Приведу довольно пространную цитату на этот счет:

«Творчество в программировании начинается с определения целей программы и заканчивается только тогда, когда в ее коде, написанном на каком-либо языке программирования, поставлена последняя точка. Попытки разделять программистов на творческую элиту, архитекторов и проектировщиков и нетворческих программистов-кодеров не имеют под собой объективных оснований. Даже если алгоритм программы строго определен математически, два разных программиста его закодируют по-разному, и полученная программа будет иметь разные потребительские качества» [35].

То же самое касается предмета труда программиста: «Предметом деятельности ученых являются упрощенные модели, в которых они могут абстрагироваться от большинства деталей реального мира, не существенных для их целей… Программист тоже работает с абстракциями, но ему приходится держать в голове гораздо больше абстракций, чем любому ученому. Абстракции сопутствуют программисту на всех уровнях разработки программы от описания ее целей до исполняемого машинного кода…» [Там же].

Страшно становится от одной мысли: «Где же взять столько одаренных личностей, например для сопровождения 1С-Бухгалтерии или для разработки сайтов?». Реальность все-таки проще и прозаичней. Есть разные задачи и разные программисты. Есть традиционная сложившаяся система образования и обучения, в лучшем случае программированию. Чем дальше, тем больше требований к продукту труда программиста – коду и к его качеству предъявляется со стороны различных процессов программной инженерии. Рассматривать эти

требования абстрактно, вне практики программирования как минимум схоластика. Отсюда и основная цель предлагаемого материала: **изложение основ программной инженерии в процессе последовательного усложнения и улучшения качества разрабатываемого кода.**

Еще одно замечание. Условно, в сапожном деле можно выделить три категории: те, кто умеет тачать сапоги, и те, которые знают, как это делать и те, которые видели, как это делают другие. В учебном процессе это называется *умения* (опыт), *знания* и *представления* соответственно. Программирование в информационных технологиях – то же самое, что электротехника в энергетике. Пожалуй, только «эффективный менеджер» может не знать законов предметной области, в которой он работает, руководствуясь чистым менеджментом. В разных видах деятельности, связанных с программной инженерией, знание процессов, связанных с функционированием программных систем, необходимо в разной степени:

- бизнес-аналитика – не обязательно;
- системная аналитика – желательно, так как любое требование необходимо оценивать с позиций реализуемости;
- архитектура – безусловно, так как архитектура – это штучный продукт из доступных решений, каждое из которых нужно уметь оценивать изнутри;
- кодирование или собственно программирование – по определению;
- тестирование – в зависимости от рода деятельности.

Последовательность освоения

> Учитель сказал: если хороший человек учил людей семь лет, их можно посылать в сражение.
>
> *Конфуций*

Понимание сущности процессов в программной инженерии не может возникнуть на пустом месте и в короткий срок. Для этого необходимо систематическое *восходящее* освоение практики программирования с нуля в несколько этапов [83]:

- базовое программирование «на уровне Бейсика»: язык программирования, паттерны анализа и разработки простого программного кода: структурное, «историческое», «грязное» программирование [9];
- продвинутое классическое программирование: динамические структуры данных, рекурсия, оптимизация по времени и памяти, динамическое программирование, трудоемкость;

- объектно-ориентированное программирование;
- паттерны конструирования, инжиниринг ПО;
- основы дисциплин жизненного цикла ПО (один-три предмета типа «Проектирование ПС», «Конструирование и тестирование мобильных приложений»);
- управление программными проектами, коллективное владение кодом, системный подход к проектированию.

Крайне желательно, чтобы каждый этап был подкреплен соответствующей дисциплиной учебного плана (как минимум один семестр на каждую).

Приемы изложения

Часто по ходу изложения возникают замечания, аналогии, факты и ассоциации, для которых каждый раз необходимо делать вводную фразу. Вместо этого используются соответствующие заголовки абзацев.

Стоп-кадр. Иллюстрация из истории развития информационных технологий, художественный или социальный штамп.

Так бывает. Конкретный пример из практики, иллюстрирующий изложение.

Замечание по теме. Некоторый частный факт, имеющий отношение к материалу.

Эмоции и ничего личного. Факт, относящийся к делу, но имеющий особую эмоциональную окраску.

Баня, музыка, кино и мост. Аналогия из другой области, где имеет место проектная деятельность – исполнение музыки, съемка фильма, строительство бани, возведение моста. Разработка ПО – один из видов проектной деятельности. Чтобы понять специфику программной инженерии, лучше всего привести аналогию из другой области проектной деятельности. Например, описать процесс постройки бани SCRUM-командой, оценить роль проектной документации на мост и на программный проект, посмотреть на формы организации музыкального коллектива – от симфонического оркестра до джаз-банда.

Философия и программирование. Любое обобщение в предметной области имеет свои аналоги в других областях знаний, а также вписывается в общие законы бытия.

Проект средней руки. Чтобы проиллюстрировать некоторые решения, необходимо анализировать проект средней сложности объемом ориентировочно 15–20 тысяч строк кода. Для малых проектов они не являются строго необходимыми. В качестве основного примера используется система контроля рейтинга успеваемости студентов [96–101].

Программная инженерия и проблемы образования: действительные и мнимые

С позиций программной инженерии общие проблемы образования видны особенно ярко. Некоторые из них традиционны для нашего менталитета и образования, другие – приметы нового времени. Все они в той или иной мере повлияли на содержание и изложение материала.

Учебные задачи и деловые игры. Отраслевики и инноваторы от учебного процесса постоянно требуют ставить реальные задачи в противовес типовым тематическим заданиям. Тематические задания упрощают и концентрируют проблему. Для качественного решения реальной задачи может не хватить квалификации. Игры опасны тем, что в них можно реально «заиграться», если упрощенную игровую ситуацию отождествлять с реальностью.

Навыки ремесла. С самого начала обучения дисциплинам цикла программирования многие требования к соблюдению качества кода воспринимаются как придирки со стороны преподавателя. К ним относятся: документирование ключевых моментов разработки, соблюдение стиля кодирования, модульности, отделение функционала от представления, преемственность и повторное использование кода.

Украл или «скопипастил». Плагиат (компиляция, копипаст) является большой проблемой не только для объективной оценки выполненной работы, но и для поддержания качественного уровня и дисциплины в учебном процессе. В программировании с этим связана еще одна проблема. Для решения задачи можно воспользоваться сторонней библиотекой – закрытым кодом, прямо скопировать открытый пример либо адаптировать его под задачу. Методические пособия и публикации на программистских форумах для этого, собственно, и предназначены. Требования преподавателя здесь должны быть сбалансированы и разумны:

- стоит требовать *изобрести велосипед*, т. е. своими руками сделать известное решение, если это необходимо для понимания основных принципов организации кода, сущности алгоритмов и структур данных, соблюдения интерфейсов и спецификаций;
- при использовании готовых решений следует оценивать привнесенную *прибавочную стоимость*: какие изменения внесены в исходный код, насколько студент понимает все нюансы используемого кода, способен адаптировать реализацию под новые требования, протестировать и оценить эффективность и затраты ресурсов. В обязательном порядке требовать ссылки как на оригинал программного кода, так и на описания его применения.

Ориентация на готовые решения. Еще одна неприятная тенденция общества потребления. Многие студенты убеждены, что любое программное решение можно «нагуглить» в виде подходящей библиотеки. При этом не анализируется, насколько оно соответствует требованиям, перспективно, затратно и вообще подходит для решения поставленной задачи.

Уровень абстрагирования. В качественном программном проекте всегда имеется значительная доля абстракций, метауровней описания, иерархий слоев и компонент, общесистемных решений. Отсутствие необходимой технологической культуры у студента не позволяет ему не просто начать работу над проектом, но даже просто понять его сущность.

Проблема сложности разработки. Сложность практических и лабораторных заданий связана с ограниченностью лимита времени на их решение. Одним из вариантов развития навыков разработки объемного и сложного кода и обеспечения его преемственности является лабораторный практикум *нарастающим итогом*. На каждом занятии к проекту добавляется новая архитектурная компонента или развивается старая.

Схоластика в образовании тоже вносит свою лепту. Самое очевидное, что структура учебного процесса и наполнение учебных дисциплин консервативны и не отражают текущих реалий. Другой проблемой является отсутствие *рамочных знаний* о предмете: сущность, основные законы и артефакты, границы применимости, другими словами, неразвитость *прагматического подхода*. Зачастую это соседствует с усвоением значительного объема фактического материала. Например, из курса «Математическая логика и теория алгоритмов» студенты знают о машине Тьюринга и даже проектируют ее на практических занятиях, но не знают, что она представляет собой, с одной стороны, исторический артефакт программной системы, а с другой стороны, используется для доказательства алгоритмической разрешимости. В методологическом плане этот формализм лежит в основе правильного понимания проблем автоматизации программирования, тестирования, отладки, т. е. весьма прагматичен.

В отсутствие четко сформулированных рамочных знаний обучение скатывается в одну из крайностей: формальное *доказательство ради доказательства* или, наоборот, *вульгаризация представлений*.

Особенностью программной инженерии является то, что в ней отсутствуют фундаментальные рамочные законы, а во многих случаях достаточно *приближенных и асимптотических оценок*. В связи с этим важна практика оценочного применения математики и упрощенной формализации. Например, студенты решают сложные логарифмические уравнения, знакомые по школе,

но не понимают сущности логарифмической шкалы или причин логарифмической трудоемкости алгоритма.

Неразвитость индуктивного подхода. Зачастую в подходе к решению задач используется дедуктивный подход и непродуктивное абстрагирование, в то же время не применяются приемы индуктивного обобщения и установления закономерностей при анализе конкретного поведения изучаемого объекта.

Фундамент vs практика. С одной стороны, среди программистов имеется значительное количество практикующих самоучек, с другой – профессиональное образование в программировании не гарантирует успешной работы в этой области. На программистских форумах периодически вспыхивают дискуссии о пользе или бесполезности фундаментального образования, о достаточности практики использования прогрессивных технологий и средств разработки. Причем у противников фундаментального образования особой нелюбовью почему-то пользуются красно-черные деревья. В данном случае не правы обе стороны. Без отслеживания технологий, безусловно, нельзя. В области разработки ПО удачные разовые решения быстро становятся общедоступными инструментами. В то же время с помощью любого инструмента можно написать сколь угодно плохую программу либо использовать его максимально неэффективно, если не знать основных законов его работы.

ГЛАВА 1

ГОМЕОПАТИЧЕСКАЯ ДОЗА ПРОГРАММНОЙ ИНЖЕНЕРИИ

1.1. О самоценности программного кода

Программисты и студенты не любят писать комментарии. Программисты и студенты не любят писать отчеты и документацию. Основной довод: «Из кода и так все ясно». Многие студенты искренне убеждены, что описание программы – это набор скриншотов и пояснений к ним. В манифесте экстремального программирования записано, что работающий код важнее документации к нему. И так далее. Если уж попытаться сформулировать требования к минимуму документов по описанию и сопровождению программного проекта, то это выглядит так:

> **Абсолютный минимум документации:** в документации должно быть отражено то, что нельзя непосредственно увидеть в тексте программы.

Фразу нельзя непосредственно увидеть, нужно понимать, как нельзя получить тривиальным анализом кода. Например:

- требуются специальные знания – предметная область, специфические алгоритмы;
- требуется анализ значительной части кода или углубленный анализ его поведения;
- имеются важные неявные допущения и ограничения;
- имеются сведения об узких местах, полученные в процессе отладки и тестирования.

Можно перечислять до бесконечности. Все это можно объединить под общим названием **ключевые моменты.** Очевидно, чем сложней проектируемая

система, тем больше в ней неочевидных элементов и общих принципов, которые следует оговорить до начала кодирования.

Резюме: объем требуемой документации обратно пропорционален качеству кода.

Проектирование «от кода»

И все-таки множество проектов разрабатывается «от кода», т. е. сам код является единственным документом и спецификацией проекта. Сюда можно отнести небольшие по объему проекты, производимые по принципу *все в одном*, проекты, исполняемые группой разработчиков, понимающих друг друга с полуслова (если машинного слова, то short).

Для описания процесса разработки следует оговорить основные элементы техники работы с кодом.

Артефакт – любой искусственно созданный элемент программной системы в процессе ее проектирования: исполняемый файл, исходные текст, веб-страница, справочный файл, сопроводительный документ, файл с данными, модель, база данных.

Ad hoc (ад хок) – к этому, для данного случая, для этой цели (лат). Частное техническое решение, принятое для конкретного случая. Программный код, решающий проблему частным образом по месту возникновения.

Каркас проекта. Набор интерфейсов и абстракций для основных компонент и сущностей проекта:

- позволяет более или менее независимо наполнять проект согласованным содержимым;
- является аналогом документации по проекту, поскольку в его коде зафиксированы многие положения, соглашения и артефакты, касающиеся проекта.

Программный прототип. Проект, исполняющий основную или критическую часть функциональности целевого проекта, но не имеющий потребительских свойств. Иногда прототип может разрабатываться для отдельной физической или функциональной компоненты для проверки, тестирования и оценки количественных характеристик проекта, например, производительности, надежности.

Программный макет. Проект, воспроизводящий *внешнюю сторону* программной системы, формальное поведение, «пустышка». Одним из вариантов макета является прототип графического интерфейса.

Рефакторинг. Изменение структуры кода в сторону улучшения его качества (модульность, управляемость, формальные показатели) *без изменения функциональности.* Рефакторинг следует отличать от *оптимизации*, которая

направлена на улучшение количественных характеристик исполнения программы (быстродействие, используемая память).

Реинжиниринг. Коренная перестройка функциональности, архитектуры или структуры кода проекта в целом с использованием существующего программного кода.

Релиз – выпуск готового программного продукта, готовый программный продукт в очередной версии.

Качество кода – формальное соответствие кода определенному набору правил:

- оформление – форматирование, документирование, комментирование, стилистика;
- метрические показатели структуры кода (метрика) – цикломатическая сложность, связность, сцепление, отсутствие дубликатов;
- покрытие кода тестами.

Поддержка качества *кода не гарантирует его эксплуатационных свойств* – надежности, устойчивости, вероятности ошибок. Однако при прочих равных условиях способствует улучшению этих свойств в процессе разработки. Иными словами, некачественный код может быть сколь угодно «хорошим», но для разработки «хорошего» кода желательно поддерживать его качество.

Копипаст (от copy / paste) – идентичные участки кода, свидетельствуют об отсутствии необходимых общих или универсальных решений и использовании вместо них решений ad hoc. Копипастом можно считать *изоморфный* код, например полученный заменой имен.

Жизненный цикл разработки «от кода»

Весь мир насилья мы разроем
до основанья, а затем
Мы наш, мы новый мир построим…

Пролетарский гимн «Интернационал»

Процесс разработки «от кода» может выглядеть по-разному. В самом простом и худшем случае это выглядит так. Разрабатываем прототип графического интерфейса – дизайн окон приложения, на его основе создаем оконные классы, в них начинаем заливать код по принципу наименьшего сопротивления, т. е. туда, где понятно, что писать или по принципу *лесом еду – лес пою*. Аналогия такого процесса разработки в проектировании отдельного алгоритма – «историческое» программирование [9]. Конечно, это очень утрированное представление, но определенный процент такого подхода имеется всегда.

А теперь попробуем описать процесс грамотного ведения разработки от кода (рис. 1.1). В первый этап разработки спецификаций пока включим оптом все артефакты – документы проектирования, ограничив его началом написания программного кода. На следующем этапе создается **каркас проекта.** Основное требование: в нем должны быть определены все интерфейсы и абстракции, соответствующие функциональным и архитектурным спецификациям. Таким образом, спецификации проекта *материализуются* в коде. Следующим этапом, значительным по объему, является итеративное наполнение каркаса кодом, который реализует требуемый пользовательский и архитектурный функционал. В процессе уточнения и согласования содержимого программных компонент производится их **рефакторинг** – качественная чистка без изменения функционала. Финалом процесса является появление **прототипа** или **релиза.** Хотя они отличаются отсутствием и наличием готового пользовательского интерфейса, сущность наполняющего их кода одинакова – он в большей части является готовым к употреблению.

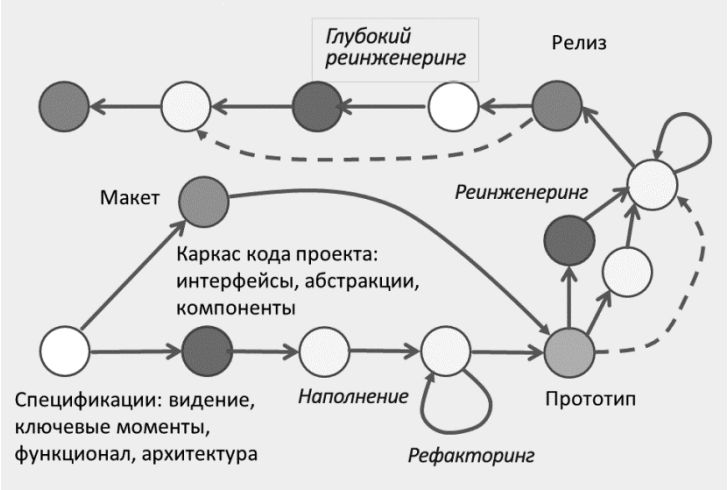


Рис. 1.1. Процесс разработки «от кода»

В процессе развития проекта может возникнуть необходимость коренной перестройки структуры кода проекта в целом – **реинжиниринга кода** – по ряду причин, в том числе:

- существенных изменений или пробелов в описании функционала, принципиальных недостатков или неэффективности архитектуры;
- накопления критической массы временных решений ad hoc, «костылей» и «заплаток»;

- общего снижения качества кода, его слабой структурированности, обилия копипаста и т. д.

Реинжиниринг может касаться изменения каркаса проекта и его повторного наполнения, а может быть и более глубоким, т. е. с возвращением на этап спецификаций. Причины появления некачественного кода обсудим ниже.

О необходимости использования общих решений и абстракций

Откуда же берется некачественный код? В качестве простого примера рассмотрим обработку исключений в клиент-серверном приложении. Предположим, что эта процедура никак не оговаривается предварительно и в процессе разработки кода возникает необходимость включения в код обработчиков исключений. Каждый программист в каждом конкретном случае будет принимать собственное оригинальное решение ad hoc. Возможно, кто-то из них создаст универсальный компонент для обработки исключений и в процессе рефакторинга значительная часть программного кода будет его использовать. Скорее всего, все равно будут иметь место ситуации, *не разрешимые стандартными средствами*, в которых в каждом случае придется использовать обходной прием (workaround, паллиатив, в простонародье «костыль»). Подобные временные решения бывают и в других областях техники, однако в программном проекте они могут иметь массовый характер. В конце концов это может привести:

- к уменьшению надежности и устойчивости приложения;
- к ухудшению качества кода, а отсюда к увеличению вероятности внесения в него ошибок.

Чтобы показать, насколько многогранной является эта проблема, определим ситуации, в которых фигурирует обработка исключений:

- **исключение или состояние валидности объекта.** К исключениям должны приводить ситуации, когда продолжение нормального исполнения программы невозможно. В случае недопустимого значения объекта логичнее ввести отдельное состояние неопределенного значения объекта. Например, в случае с GPS-координатами возможны два состояния: отсутствие координат и устаревшие координаты. Тогда компонента, получившая невалидный объект, корректирует свое поведение, например, объект просто не отображается на карте. Хорошим решением является шаблон-контейнер, вводящий состояние невалидности объекта для любого класса (см. раздел 3.3);
- **классификация исключений.** Исключения разделяются не только по их источнику, но и по степени влияния на работоспособность текущего сценария, приложения и системы в целом. Например, можно выделить предупреждения, не отменяющие текущий сценарий, ошибки, исправляемые повторным

обращением (например, отсутствие доступа к сети), ошибки доступа к данным (отсутствие файла или недопустимый формат), ошибки программирования, фатальные ошибки;

- **внешняя и внутренняя, клиентская и серверная обработка исключений.** Реакция на исключения может быть разной «для внешнего и внутреннего употребления». Для пользователя она должна сопровождаться возможными вариантами изменения сценария, а также быть как-то связанной с системой помощи для поиска причин и принятия нужных решений – сообщения типа «файл не может быть открыт» мало что дают в таком случае. Внутренняя обработка исключения состоит в его классификации, фиксации и определении процедур восстановления для затронутых программных компонент. Исключение в клиентском приложении может касаться единственного клиента, а в серверном – нескольких клиентов, например, при аварийном завершении службы, обрабатывающей очередь запросов от клиентов;
- **логирование исключений при сопровождении.** В процессе бета-тестирования исключения, происходящие в приложениях конечных пользователей, требуют более сложной обработки. Необходимо их фиксировать в БД сервера, а также сохранять необходимый контекст – данные пользователя для воспроизведения исключения. При значительном количестве однотипных исключений не следует фиксировать повторные и т. д.

Из уже перечисленного становится ясно, что обработка исключений не является простой процедурой, которая может быть реализована с использованием первых попавшихся средств. Для этого необходим общий подход, вот его основные идеи:

- система обработки исключений – составная часть архитектурного аспекта проектирования – *устойчивости системы.* Она должна рассматриваться в комплексе с другими решениями – восстановление, доступность данных;
- требований к обработке исключений как таковых не существует, они являются следствием общих требований к атрибутам качества – *надежности, устойчивости, доступности*;
- основные решения по обработке исключительных ситуаций должны быть сформулированы как минимум до начала написания кода;
- ключевые моменты системы обработки исключений должны быть описаны в *архитектурных спецификациях* программного проекта;
- в *каркасе проекта* должны быть реализованы необходимые интерфейсы и базовые абстракции.

Очевидный вывод: нельзя просто начинать писать код. Нужно начинать с создания каркаса.

Документирование: производственная необходимость и чувство меры

Необходимость документирования бывает обусловлена факторами, не зависящими прямо от существа проекта: обоснование финансовых документов, спецификация работ, отдаваемых на аутсорсинг, составление отчетов «для начальства». Если же отбросить эти формально-бюрократические неудобства, то в качестве проектной документации, пожалуй, можно ограничиться неканоническими набросками архитектуры (рис. 1.2) и прототипами графического интерфейса.

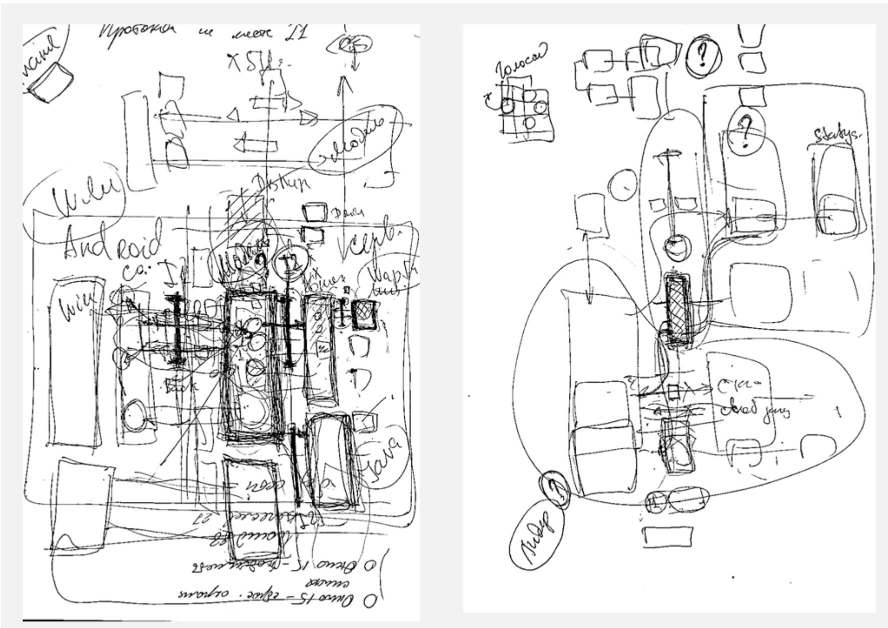


Рис. 1.2. Документ совещания по обсуждению архитектуры сервера и ее набросок

Даже если отказаться от необходимости проектной документации как таковой, есть еще ряд резонов для ее использования в процессе разработки:

- **документирование разработки задним числом**: зафиксировать ключевые моменты организации разработанной системы для последующего сопровождения, реинжиниринга;
- **восстановление контекста**: чтобы не забыть или легче вспомнить при возвращении к работе над проектом;

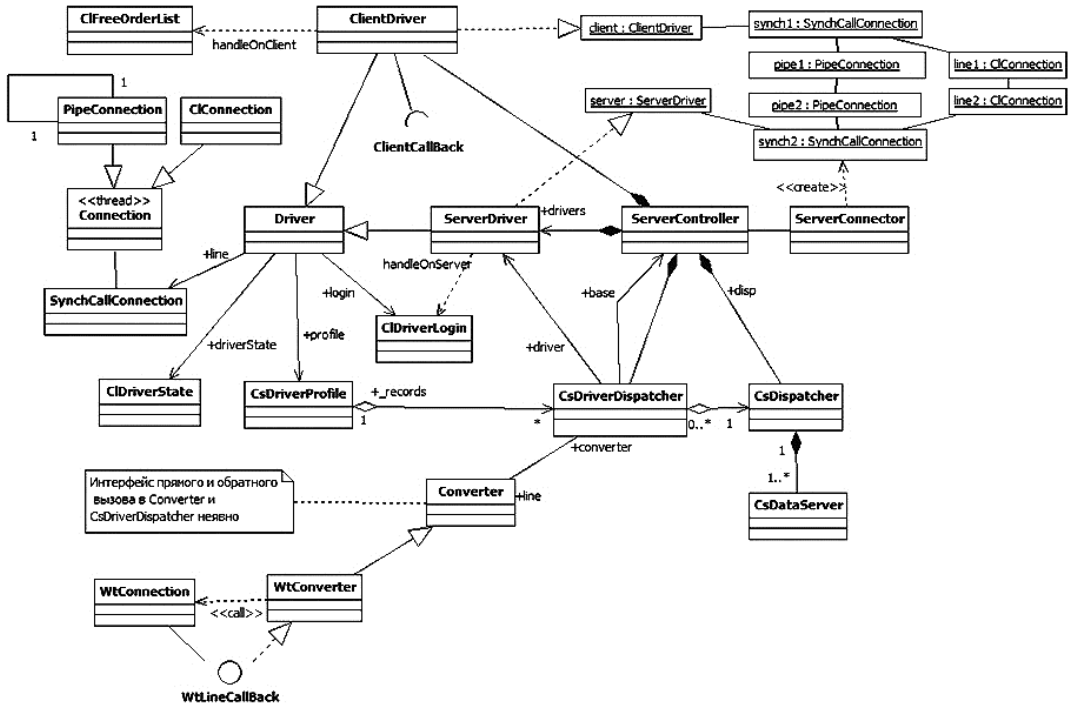


Рис. 1.3. Архитектурные классы – UML-диаграмма сервера и ее набросок в UML

- **средство коммуникации**: действительно ли все участники разработки имеют в виду одно и то же;
- **справочник**: текущая структура системы, компоненты, связи;
- **визуализация**: графическое описание информативней и компактней текстового.

Проектная документация: чертеж или эскиз?

Еще один камешек в огород проектной документации добавляет сама специфика разработки ПО. В любой проектной деятельности между этапами проектирования и производства проектная документация выступает в виде *чертежей*, в соответствии с которыми объект воспроизводится «в металле». Чертеж одновременно разграничивает ответственность между проектировщиком и изготовителем, по нему можно установить вину в случае аварии или непредвиденной ситуации.

Чертеж – описание, позволяющее воспроизвести (изготовить) предмет с заданными свойствами и качеством.
Эскиз – набросок, не обладающий таким свойством.

Ничего подобного в разработке ПО нет. «…Проект архитектуры не является чертежом. Все подробности программной системы описываются только кодом на языке высокого уровня, который, таким образом, является чертежом программы. А поскольку все операции, ведущие к созданию чертежа, являются проектированием, то и вся разработка ПО должна считаться проектированием» [4].

Отсюда следует, что любая проектная документация для ПО является *эскизом,* изготовленный по нему продукт не имеет 100 %-ной гарантии работоспособности и других менее важных потребительских качеств. То же самое можно сказать о предварительных версиях, прототипах и макетах программного продукта – это тоже своего рода эскизы. Именно поэтому особая роль отводится *тестированию* как программного продукта, так и проектной документации (см. раздел 8.1).

Качество кода и его развитие

Выше уже был определен термин «качество кода» и отмечен его главный изъян: никакой формальный набор требований и метрик не гарантирует того, что код действительно будет работоспособным и надежным. Именно с этой стороны звучит основная критика: «Главное достоинство моего кода в том, что он работает в данном месте и в данное время, а остальное не важно».

Посмотрим на этот вопрос с другой стороны. Значительная часть разрабатываемого кода является «одноразовой посудой», т. е. не рассчитана на повторное использование и развитие, причем не только со стороны, но и самим же программистом в будущем. Если же смотреть на код в исторической перспективе, то можно сформулировать понятие качественного кода в категориях вечных ценностей, например в виде следующих неформальных свойств:

- **понимаемость, внятность, эстетика** – легкость анализа и понимания логики работы и взаимодействия с кодом, его интерфейсов;
- **универсальность** – возможность применения кода для решения разных задач путем его адаптации или прямого использования;
- **модульность** – код состоит из логически завершенных модулей, имеющих внятные интерфейсы, отдельные модули могут быть использованы в других проектах путем прямого переноса или минимальной адаптации;

- **управляемость** – возможность изменять и развивать отдельные части кода, минимально затрагивая при этом остальные участки;
- **сложность повторного использования** – необходимость внесения изменения в код при адаптации к новой задаче. Возможные варианты: реинжиниринг на уровне алгоритма, копипаст с редактированием, шаблон, библиотека и т. п.

1.2. Программная инженерия – эссе на заданную тему

Любое ремесло передается не
учебниками, а подзатыльниками.

Народная мудрость

Начинать изложение любой дисциплины всегда затруднительно. Один вопрос тянет за собой остальные, термины и определения звучат как пустой звук, поскольку не наполнены содержанием. Поэтому лучше пренебречь строгостью изложения и попытаться рассмотреть предмет со всех сторон понемногу. В результате может сложиться нужная первоначальная картина.

Технология и ремесло

Программирование и программная инженерия – это ремесленный и технологический подходы в разработке ПО, а поскольку результатом является программный код, то в производстве программного кода. В программировании-ремесле важны инструменты, приемы, навыки и опыт, в программной инженерии – технологии (нормативы, организация, контроль процесса труда и качества продукта).

Программная инженерия – технологическая составляющая, **программирование** – ремесленная составляющая в производстве программного кода.

В данном случае имеет место не противопоставление, а скорее взаимное дополнение деятельностей, в которых участвует программист. Все, что направлено на сам код, является элементами ремесла. Все, что направлено на взаимодействие с другими участниками процесса, управляющей средой, – инженерия.

Баня, музыка, кино и мост. Если взять аналогию с музыкальным коллективом, то исполнительское мастерство каждого музыканта – ремесло, а организация репетиционной и концертной деятельности, сыгранность, взаимоот-

ношения и все остальное, что имеет отношение к деятельности коллектива, – инженерия.

В основе программной инженерии лежит промышленный подход к разработке ПО. Основные его составляющие:

- качество кода: инструменты разработки, тестирование, шаблоны проектирования;
- структура производственного процесса: методологии, стандарты, свод знаний о программной инженерии Software Engineering Body of Knowledge (SWEBOK);
- документирование, моделирование, описание артефактов;
- управление процессом: методологии, фреймворки.

Для дальнейшего изложения необходимо различать близкие по смыслу термины:

- **программное обеспечение** (ПО) – набор компьютерных программ, процедур и связанной с ними документации и данных (ISO/IEC 12207);
- **программная система** (ПС) – программное обеспечение в сочетании с аппаратным, программным и пользовательским окружением;
- **программный продукт** (ПП) – программное обеспечение как товар и услуга, включающий в себя само ПО, средства и сервисы установки, обновления, сопровождения и поддержки среды функционирования. Различают «коробочные» и заказные ПП.

Еще одним важным понятием является **жизненный цикл** – период существования ПО (ПС). Различают:

- жизненный цикл разработки ПО – проектная деятельность по разработке и развертыванию ПС;
- жизненный цикл ПС – разработка, развертывание, поддержка и сопровождение.

Программная инженерия и информатика

В каких отношениях находятся программная инженерия и информатика? Здесь уместнее использовать термин *computer science* (буквальный перевод «компьютерные науки»), так как в обиходе информатика обычно ассоциируется с массовыми знаниями в области информационных технологий. Отношения достаточно специфические. Самое главное, что научные знания в этой области не содержат *фундаментальных* или *рамочных законов,* которые имеют место в естественных науках – фундаментальных и прикладных. Например, архитектура моста может быть какой угодно, но сопромат позволяет оценить запас прочности конструкции.

В программной инженерии невозможно проверить проект *на прочность* или получить его фундаментальную характеристику типа *массы* до тех пор, пока он не будет введен в эксплуатацию. Отсутствие таких законов должно компенсироваться тестированием проекта и его компонент на всех этапах разработки.

> Компьютерные науки не содержат *рамочных законов,* позволяющих проверить существенные свойства программного кода.

Но не все так плохо в Соединенном королевстве. Что касается отдельных компонент компьютерной архитектуры, организации вычислительных процессов и конкретных областей приложений, то из компьютерных наук вполне могут быть полезны:

- теория информации;
- теория алгоритмов – доказательство принципиальной невозможности существования алгоритмов и соответственно разработки программ для решения определенного рода задач (алгоритмическая неразрешимость);
- структуры данных и алгоритмы, дискретная математика, теория графов – типовые способы организации данных, эффективные способы работы с ними, решение частных задач путем сведения их к стандартным математическим представлениям и решениям;
- трудоемкость и эффективность алгоритмов – оценка эффективности алгоритмов и программ «в перспективе» в зависимости от роста объемов входных данных, масштабирование;
- математические основы предметной области (например, обработка сигналов, криптография) – методы и алгоритмы решения задач в конкретной предметной области, наукоемкое ПО;
- формальные языки и трансляторы – эффективное использование языков и средств разработки на основе понимания принципов их работы: синтаксис и семантика формальных языков, формальные модели (автоматы, стековые автоматы), описания, «зашитые» в код, и метасистемы.

Знания компьютерных наук обеспечивают качество ПО, но не гарантируют успех его разработки. Периодически на программистских форумах закипает дискуссия на тему «Нужны ли программисту фундаментальные знания компьютерных наук?» Сказанное выше является основанием для взвешенной позиции в этом вопросе. Вот навскидку набор популярных мнений:

- любое решение можно «нагуглить» или сверстать из готовых компонент или готовых библиотек;

- фундаментальные или рамочные знания компьютерных наук – структуры данных, трудоемкость алгоритмов, базовые алгоритмы не важны, если в совершенстве владеть инструментами и современными средствами разработки;
- значительная часть задач, кодируемых программистами, является типовыми, и проявлять здесь креативность как минимум неуместно;
- изобретать велосипед в виде собственного решения вместо того, чтобы использовать опробованное известное, – признак консерватизма и замшелости.

В то же время опыт разработки свидетельствует о следующем:

- любую программу с заданным функционалом можно реализовать так, что она будет работать сколь угодно медленно и использовать сколь угодно много памяти;
- неэффективный, но простой и надежный алгоритм, удовлетворяющий по производительности для текущих размерностей данных, лучше сложного, но требующего больших затрат в реализации и доказательств работоспособности.

Программная инженерия как проектная деятельность

Баня, музыка, кино и мост. К разработке программного проекта наиболее близка по духу проектная деятельность в искусстве или кино, например съемка фильма или постановка спектакля. В меньшей степени сходство имеется с проектной деятельностью в материальном производстве. Тем не менее к программной инженерии применимы принципы *управления проектами* (PM – Project Management) с учетом как общих сторон, так и специфики отрасли. Общее в проектной деятельности всех видов:

- получение продукта с гарантированным функционалом, качеством и другими свойствами, с известными сроками исполнения и в рамках отработанного технологического процесса;
- общие принципы управления проектом (менеджмент), изложенные в своде знаний по управлению проектами (PMBOK) [37].

Специфика программной инженерии:

- основанием программной инженерии является информатика, а не естественные науки;
- основной акцент делается на дискретной, а не на непрерывной математике;
- концентрация на абстрактных / логических объектах вместо конкретных / физических артефактов;

- отсутствие производственной фазы в традиционном смысле в форме «проектирование – изготовление»;
- сопровождение программного обеспечения в основном связано с продолжающейся разработкой или эволюцией, а не с традиционным физическим износом;
- нематериальный характер продукта, нулевые затраты на тиражирование, аналогии с интеллектуальной собственностью;
- отсутствие аналогов изделия в окружающем мире. На ранних стадиях разработки – создание программных прототипов как материальных аналогов проекта;
- отсутствие охватывающих объективных законов и невозможность теоретического расчета и проверки. Работоспособность проекта проверяется тестированием готового продукта или верификацией артефактов разработки;
- затраты на непосредственное производство «материальных ценностей», т. е. конструирование кода составляют 20–25 % стоимости проекта;
- материальная основа проекта – программный код является гибкой субстанцией, допускающей в любой момент внешнее качественное усовершенствование и коренную перестройку – рефакторинг и реинжиниринг.

Жизненный цикл программной системы и его модели

Жизненный цикл (ЖЦ) – период существования ПО (ПС), включающий разработку, эксплуатацию и сопровождение программного продукта, охватывающий жизнь системы от установления требований к ней до прекращения ее использования. **Модель жизненного цикла –** описание структуры и ключевых идей организации жизненного цикла. Модель ЖЦ определяет характерные черты, которые объединяют родственные методологии.

Каскадная модель ЖЦ

Каскадная модель предполагает однократное последовательное выполнение всех технологических процессов (рис. 1.4). Принятое решение на определенном этапе не может быть отменено по результатам последующих этапов, все непринципиальные ошибки исправляются тестированием.

Несмотря на свою архаичность, каскадная модель в целом или на отдельных этапах применяется в крупных проектах, а также при передаче отдельных компонент на стороннее исполнение – аутсорсинг. Например, конструирование и тестирование отдельного приложения могут быть переданы после завершения этапа анализа и проектирования системы на стороннюю разработку.

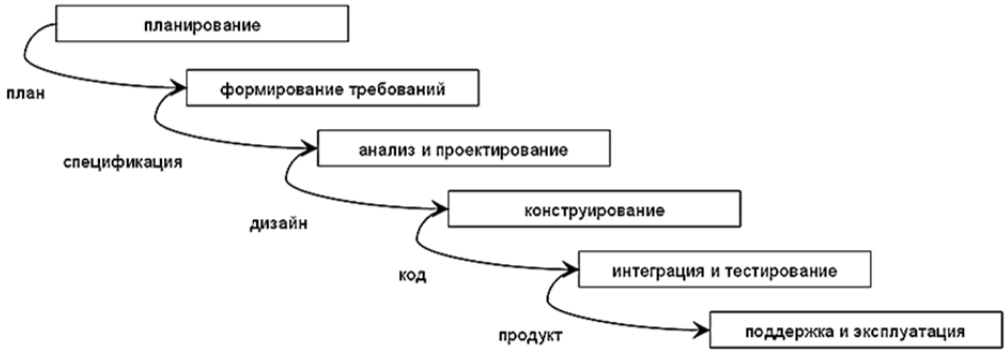


Рис. 1.4. Каскадная модель ЖЦ

Основанием для передачи и приемки является развернутое техническое задание, включающее артефакты предыдущих фаз: структуру базы данных, протоколы, спецификации графического интерфейса, сценарии, системные требования.

Итеративная модель ЖЦ

Итерация – шаг разработки, завершающийся получением согласованных артефактов системы – программного прототипа, релиза, моделей и документов. Важным здесь являются логическая завершенность и согласованность артефактов, создающих целостность – модель или работающий проект.

Инкремент – добавление функциональности, не сопровождаемое качественными изменениями свойств системы, чистая «дельта» функционала.

Итеративная модель предполагает, что разработка проекта идет в виде последовательности итераций. Итерация связана с понятием **эволюции** проекта. Под эволюцией понимается качественное изменение свойств проекта – модели или реализации. Итерация с точки зрения эволюции может быть двух видов:

- чистый инкремент: детализация требований, функциональности, архитектуры и их реализация в коде;
- эволюция требований, функциональности, архитектуры, которая сопровождается реинжинирингом кода и других артефактов, т. е. качественной перестройкой системы. При этом ревизия может распространяться от функциональности отдельных компонент до общих принципов организации системы вплоть до бизнес-требований и архитектуры.

Окончание итерации согласованными артефактами позволяет приступить к их тестированию и верификации. Это, в свою очередь, позволяет снизить неопределенность в трудоемкости и сроках исполнения и оценить возникающие риски (рис. 1.5).

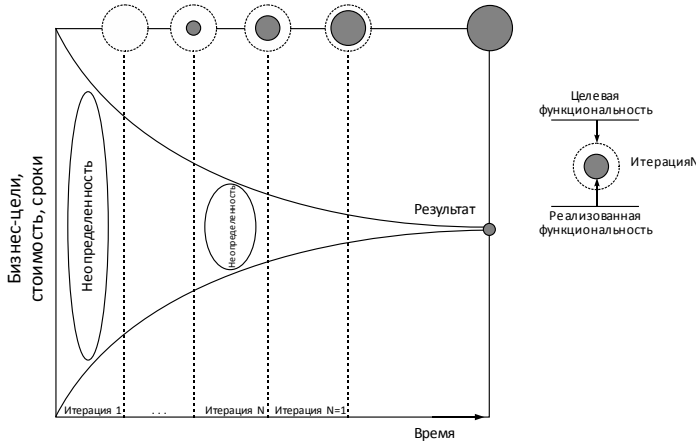


Рис. 1.5. Снижение рисков и накопление функциональности в итерационной модели [5]

Спиральная модель

Спиральная модель [5] является расширением итеративной: итерация является эволюционной и предполагает глубокую ревизию всех артефактов и свойств системы (рис. 1.6).

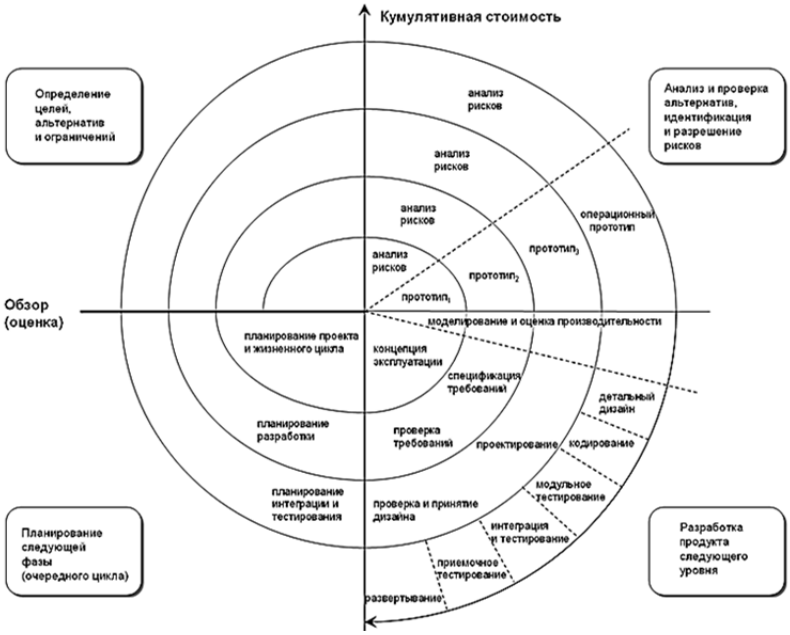


Рис. 1.6. Спиральная модель ЖЦ [5]

Итерация состоит в повторении фаз разработки *с нуля* на основе оценки результатов предыдущей итерации и рисков, таких как нереалистичные сроки и бюджет, реализация несоответствующей функциональности, разработка неправильного пользовательского интерфейса, недостаточная производительность получаемой системы и пр.

Методологии программной инженерии

Модель ЖЦ описывает только внешний вид ЖЦ, не наполняя его деятельным содержанием, чем, в свою очередь, занимается методология. **Методология** – учение о структуре, логической организации, методах и средствах деятельности. Не стоит бояться «научности» этого слова.

Все методологии и связанные с ними артефакты (модели, стандарты, фреймворки) можно позиционировать по двум качественным осям (рис. 1.7):

- используемая модель жизненного цикла: каскадная или итерационная;
- степень тяжеловесности и формализации: легкие (неформальные) и тяжеловесные (формальные).

Рис. 1.7. Методологии программной инженерии и связанные с ней понятия

Вторая ось касается нескольких аспектов разработки:

- *степень охвата жизненного цикла*: легковесные – тяжеловесные процессы. Некоторые деятельности могут отсутствовать в методологии, если они в малой степени присутствуют в проектах, например, в связи с его масштабом. Они пускаются на самотек или просто на оговариваются в методологии;

• *уровень формализации документов*: низкий – высокий. Используются канонические, стандартные формы артефактов и документов, либо их формат и содержание не оговариваются;

• *степень формализации взаимоотношений* исполнителей: низкая (гибкие) – высокая (традиционные). Практикуются формальные документированные взаимоотношения либо стимулируются неформальные коммуникации и создание доверительной атмосферы.

Коротко охарактеризуем артефакты, связанные с методологией:

• ГОСТы серии 19, 34 – классическая водопадная модель;

• ГОСТ Р ИСО/МЭК 12207–99 – описание процессов ЖЦ программных средств (методология не оговаривается);

• структурные методы – наследие эпохи *структурного программирования*, каскадный одномерный функциональный подход;

• унифицированный процесс (UP – Unified Process) – тяжеловесная итеративная, объектно-ориентированная методология, использующая UML, поддерживается фреймворком Rational Rose (см. раздел 5.1);

• Crystal Clear – легковесная гибкая методология, набор из семи практик, ориентированных на коммуникации, а не на процессы (рис. 1.8);

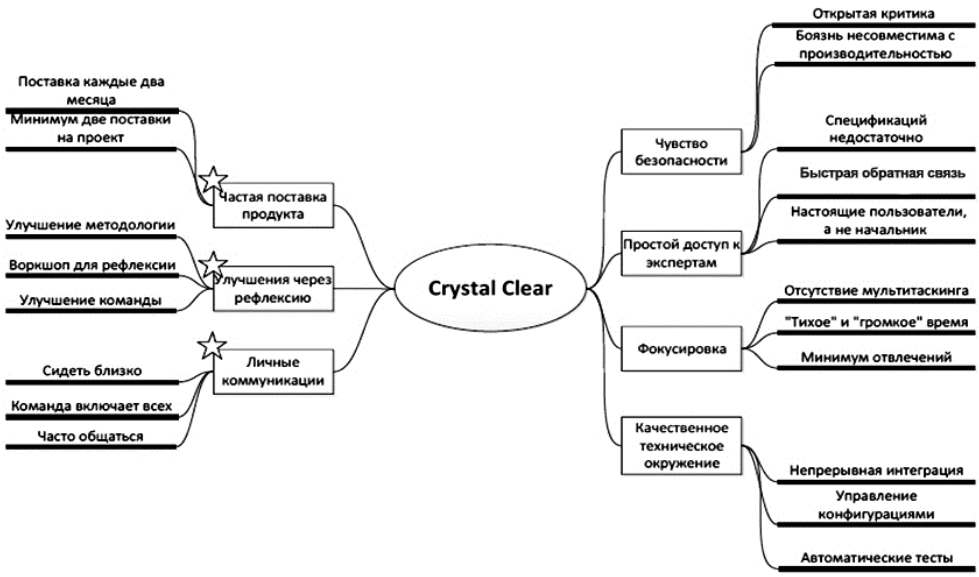


Рис. 1.8. Набор практик Cristal Clear

- экстремальное программирование (XP – Extreme Programming) – набор практик и рекомендаций в русле легковесного и гибкого проектирования (см. раздел 5.2);
- Scrum – популярный фреймворк на основе гибкой итеративной методологии с довольно глубоким охватом процессов ЖЦ и оригинальными решениями по планированию и управлению проектом (см. раздел 5.2);
- быстрая разработка приложений (RAD – Rapid Application Development) – разработка ПО осуществляется небольшой командой разработчиков за три–четыре месяца путем использования инкрементального прототипирования с применением инструментальных средств визуального моделирования и разработки. Большинство современных интегрированных сред разработки (IDE) по существу поддерживает RAD. Предельный случай RAD – верстка приложения из готовых компонент;
- AgileUP – авторская методология Скотта Амблера [23], попытка соединить строгость UP с гибким подходом к проектированию за счет снижения объемов тяжеловесной методологии и использования неканонических моделей и диаграмм (рис. 1.9);

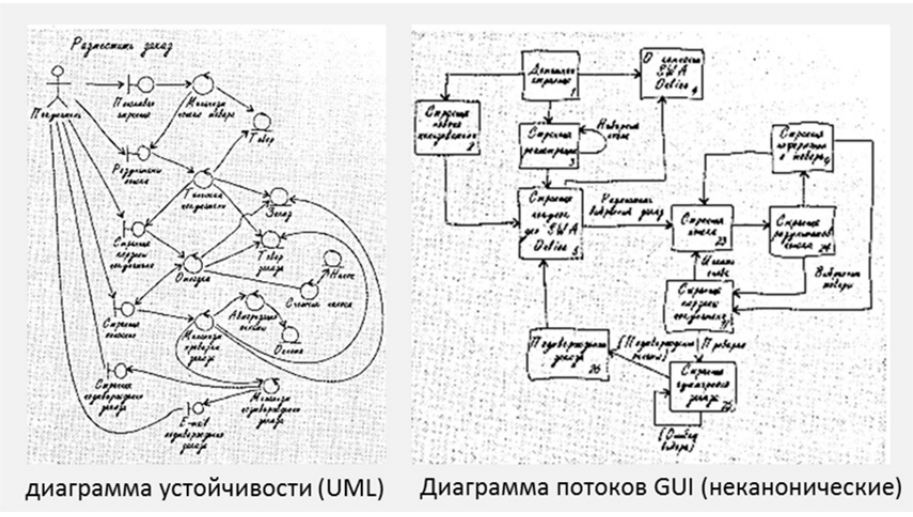


Рис. 1.9. Диаграммы AgileUP [23]

- ICONIX – облегченная версия UP с минимизацией видов моделей, артефактов и процессов (рис. 1.10). Использует только четыре вида диаграмм UML: классов, устойчивости, вариантов использования и последовательности.

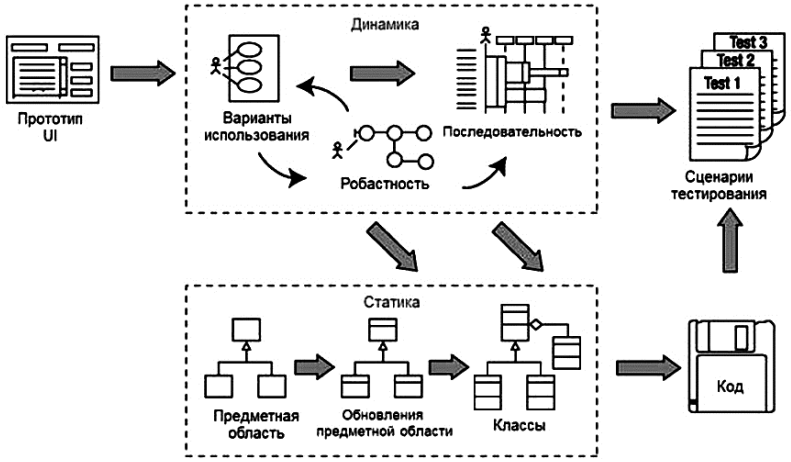


Рис. 1.10. Артефакты ICONIX

Исторический экскурс: структурные модели проектирования ПО

Структурные модели представляют не только исторический интерес. Они показывают, как текущая парадигма программирования находит свое отражение во всех сопутствующих процессах и моделях проектирования. **Структурные модели** основаны на идеях технологии структурного программирования – модульность, иерархия, функциональная декомпозиция, проектирование «от функции к функции», структуры данных, нисходящее проектирование и тестирование. Они включают:

- SADT (Structured Analysis and Design Technique) – модели и функциональные диаграммы;
- DFD (Data Flow Diagrams) – диаграммы потоков данных;
- ERD (Entity-Relationship Diagrams) – диаграммы «сущность–связь».

Недостатки структурных моделей следуют из ограниченности функционального подхода. Это невозможность эволюции, функциональная одномерность, отсутствие органической связи «функциональность–данные».

Модель SADT

Базовый элемент модели – функциональный модуль (функция), определяет процесс, в который включены данные, условия их обработки и требуемые ресурсы (рис. 1.11). Соответственно модуль-функция имеет управляющие связи четырех видов:

- вход – данные и объекты, потребляемые или изменяемые процессом;

- выход – основной результат процесса, конечный продукт;
- управление – условия, которыми регулируется процесс (стандарты, правила, время, бюджет);
- механизм – ресурсы и исполнители.

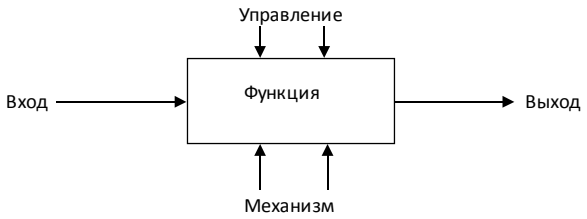


Рис. 1.11. Базовый элемент SADT

Функциональные диаграммы полезны для описания бизнес-процессов. При использовании их в качестве спецификации программного кода можно провести параллели между условиями потоком управления (вызовом функции), а также между механизмами средой исполнения кода (процессор / память).

Собранная из таких элементов модель позволяет отобразить структуру потоков данных и потоков управления и связать их в единое целое. На рис. 1.12 изображены тривиальные, с точки зрения программиста, варианты связывания модулей:

- *коммуникативность по входу* – набор данных с выхода модуля является входным для двух и более;

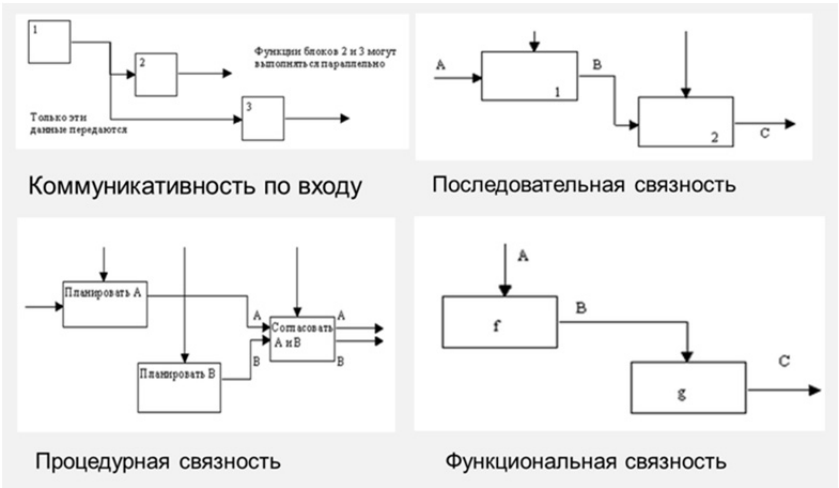


Рис. 1.12. Примеры моделей SADT

- *последовательная связность* – выход одного модуля является входом другого;
- *процедурная связность* – выходы двух модулей являются входами одного;
- *функциональная связность* – при исполнении кода одного модуля происходит вызов другого.

Модель потоков данных (DFD)

Основными элементами модели являются *внешние сущности, системы / подсистемы, процессы, накопители данных, потоки данных* (рис. 1.13). Как и в моделях потоков данных, используемых при описании вычислительных процессов, граф связей отражает не последовательность изменения состояний или действий «процессора», а перемещение объекта данных по технологической цепочке его хранения и обработки.

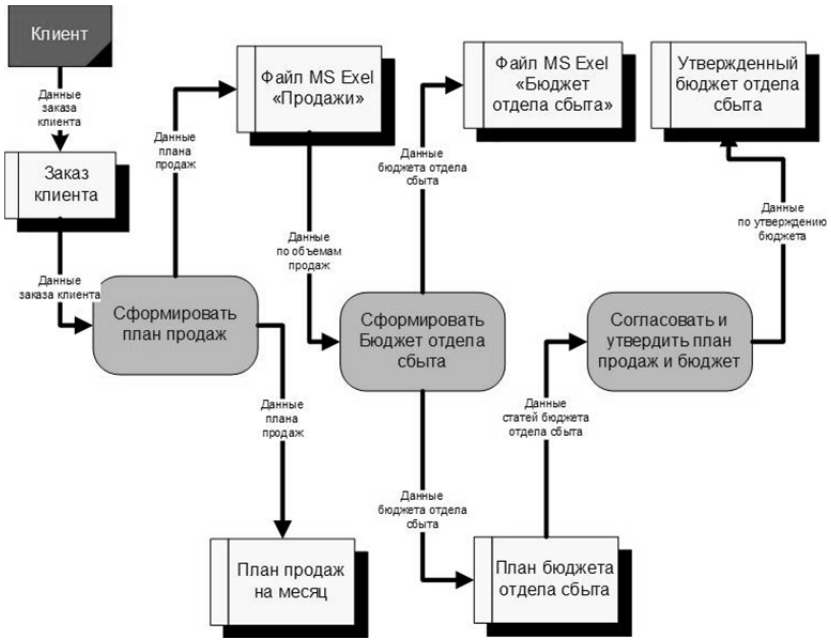


Рис. 1.13. Пример модели DFD

Модели «сущность–связь» (ERD)

Модель «сущность–связь» имеет много общего с сетевыми моделями баз данных и с диаграммами классов в UML (рис. 1.14).

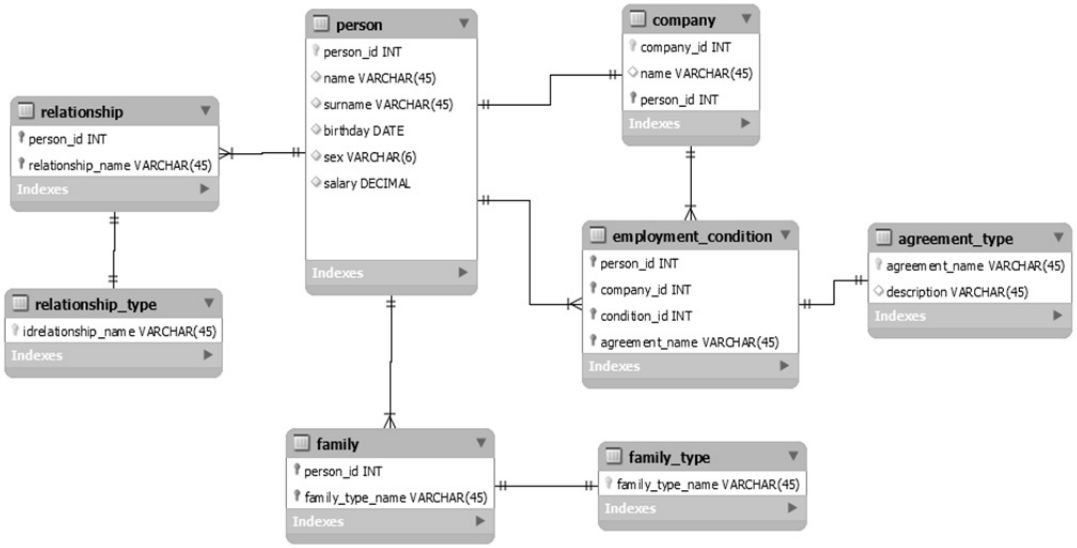


Рис. 1.14. Пример модели ERD

Основные элементы – нотация модели:

- сущность – аналог класса, таблицы;
- экземпляр сущности – аналог объекта, записи;
- атрибут – аналог свойства объекта, поля записи с характеристиками, относящимися к БД: простой, составной, однозначный / многозначный, необязательный, производный вычисляемый;
- ключ – уникальный идентификатор объекта с характеристиками: простой / составной, первичный / альтернативный (несколько ключей), абсолютный (все атрибуты принадлежат сущности) / относительный (зависимая сущность);
- связь – аналог отношения с характеристиками:
  - идентифицирующая (обязательная, родитель–потомок) / неидентифицирующая;
  - мощность связи – количество экземпляров, соответствующих одному экземпляру на противоположном конце – 0,1,*;
  - тип связи (1 : 1, 1 : N, N : 1, N : N).

Как любая модель, ориентированная на данные, она не отражает функциональность системы и функционально-зависимые свойства связей.

Под сводом знаний SWEBOK

Основополагающим документом по программной инженерии является «Свод знаний по программной инженерии» (SWEBOK) [2, 6], состоящий из десяти основных документов:

- *Software requirements* – программные требования;
- *Software design* – дизайн, проектирование (архитектура);
- *Software construction* – конструирование ПО;
- *Software testing* – тестирование;
- *Software configuration management* – конфигурационное управление;
- *Software engineering management* – управление в ПИ;
- *Software maintenance* – эксплуатация (поддержка) ПО;
- *Software engineering process* – процессы ПИ;
- *Software engineering tools and methods* – инструменты и методы;
- *Software quality* – качество ПО.

Бóльшая часть документов свода знаний соответствует основным технологическим процессам (дисциплинам) в разработке ПС. К сожалению, здесь применим афоризм Козьмы Пруткова: «Многие вещи нам непонятны не потому, что наши понятия слабы; но потому, что сии вещи не входят в круг наших понятий». Работа с этими документами имеет смысл только после погружения в соответствующий род деятельности, когда становится понятной содержательная сторона описанных в них предметов.

ГЛАВА 2
ИНСТРУМЕНТЫ И ДОКУМЕНТЫ. UML

Университеты марксизма-ленинизма (УМЛ) – одна из форм высшего звена системы партийного просвещения.

Большая советская энциклопедия

2.1. UML – как же он моделирует?

Бесспорное, а потому и непродуктивное определение UML выглядит следующим образом:

> **UML** – унифицированный язык моделирования для описания программной системы на всех этапах жизненного цикла с использованием парадигмы объектного моделирования.

То, что UML используется для описания ПС и может быть применен на всех этапах ЖЦ, следует из его назначения [7]. Лежащий в его основе объектно-ориентированный подход может быть как формальным общим местом, так и основой «объектно-ориентированного фундаментализма» [27], что определяет различные практики применения UML. Разберем название «по буквам».

Унифицированный. UML возник не на пустом месте: все его компоненты имеют исторические прототипы. Достоинство UML в том, что он объединил их в единое целое. Другая интерпретация термина: UML не привязан к программным системам, он может использоваться для моделирования любых сложных систем. Применительно к программной инженерии это могут быть предметная область, бизнес-процессы. Парадоксальным фактом является и то, что UML используется разработчиками и как метаязык для описания расширений UML, а также унифицированного процесса разработки ПО.

Язык. UML – формализованный графический язык. Все его элементы строго специфицированы, что дает основания назвать его *формальным графическим языком***.** Но формальный язык можно рассматривать в двух аспектах:

- язык описаний – спецификаций модели;
- язык интерпретации модели, т. е. собственно моделирования.

Со вторым дело обстоит значительно хуже. Хотя отдельные компоненты языка позволяют генерировать непосредственно программный код (например, диаграммы классов), собственно принципы интерпретации компонент модели разработчиками языка не оговорены. Да они и не могут быть оговорены, поскольку модель не рассматривается изначально как модель программного кода системы, а может описывать любые произвольные сущности. Кроме того, никто не гарантирует отсутствие пробелов в комплексном описании модели в виде нескольких компонент, т. е. ее семантической целостности.

Моделирование. Здесь больше всего вопросов, поэтому рассмотрим претензии UML на моделирование отдельно.

Что понимать под моделированием?

> Он лег на спину и заложил обе руки под голову. Илья Ильич занялся разработкою плана имения. Он быстро пробежал в уме несколько серьезных, коренных статей об оброке, о запашке, придумал новую меру, построже, против лени и бродяжничества крестьян и перешел к устройству собственного житья-бытья в деревне.
>
> *И.А. Гончаров. Обломов*

Прежде всего надо разобраться с самим термином «моделирование». Согласно определению, **модель – это система, воспроизводящая поведение оригинала**. Соответственно моделирование – анализ или исполнение модели с целью изучения свойств или поведения оригинала. Модели и способы моделирования бывают разными (рис. 2.1):

- *натурная модель* представляет собой физическую полноразмерную или масштабную копию для изучения ее поведения в той же среде, что и оригинал. Способ моделирования – прямое воспроизведение;
- *математическая модель* – описание поведения оригинала средствами того или иного математического аппарата, например, система обычных или дифференциальных уравнений, система массового обслуживания. Способы моделирования могут быть разными:

- *аналитическое решение* в виде функциональной зависимости параметров модели от времени, координат и т. п. Обычно аналитическое решение может быть получено для достаточно простых моделей, построенных в результате упрощения исходной модели или игнорирования в ней несущественных деталей;
- для сложных математических моделей можно получить *приближенное частное решение* для некоторого набора значений параметров при помощи методов вычислительной математики, которые обычно реализуются программными средствами. Решение при таком способе получается *по точкам*, т. е. в виде набора частных решений, представленных в форме таблицы или графика;

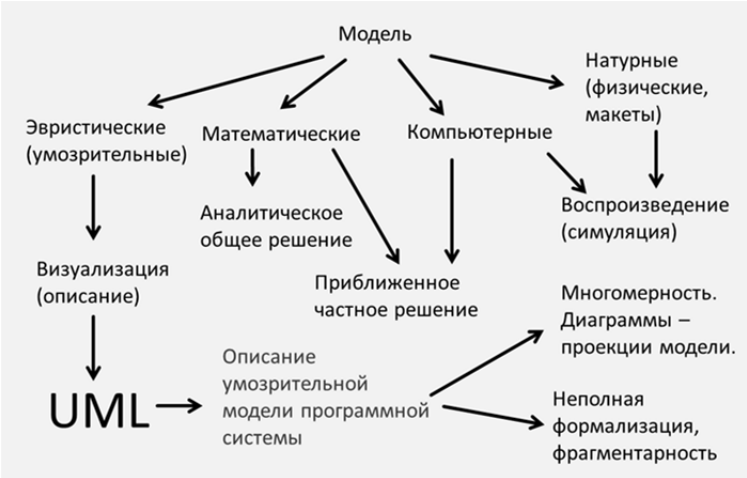


Рис. 2.1. Виды моделей

- *компьютерная модель* – программа, которая вычисляет приближенное частное решение математической модели, либо осуществляет симуляцию – представление компонент оригинала в виде программных объектов и воспроизведение в них последовательности событий или состояний оригинала;
- *эмпирическая* – умозрительная модель, представленная в виде описания или схемы, моделирование заключается в логическом анализе (верификации), умозрительном или компьютерном исполнении сценариев.

Как это ни странно, но модель в UML относится к самому слабому в смысле воспроизведения оригинала типу моделей – эмпирическому. В модели в UML нельзя воспроизвести оригинал в традиционном для моделирования понимании.

Модель в UML – умозрительное представление программной системы в «воспаленном воображении разработчиков». Документы UML – проекции (срезы) этого представления, отражающие отдельные аспекты организации или поведения системы.

Фанатизм от UML и чувство меры

Как и в случае с документированием программного проекта, в отношении UML имеются две крайности: от отрицания необходимости использования вообще до полной фиксации в документе каждого «чиха» в процессе проектирования. Несколько замечаний на этот счет:

- глубина документирования и использования UML в проекте может меняться в самых широких пределах в зависимости от принятой методологии;
- имеются полезные неканонические альтернативы: схемы (например, диаграммы экранов GUI), наброски, неформальные документы, нестандартные артефакты;
- в дополнение к основному назначению модель можно использовать как средство коммуникации, визуализации, восстановления контекста разработки или как справочник.

Есть еще пара замечаний по использованию UML как модели программной системы. В разделе 1.1 уже был сформулирован тезис о том, что только программный код является *чертежом проекта,* ибо способен гарантированно воспроизвести проектируемую программную систему. Все остальные документы и модели не обладают свойством чертежа, а являются *эскизами.*

Вторая претензия общего плана. Любой программный код имеет *комбинаторное множество вариантов исполнения.* В терминах UML код является классификатором, порождающим экземпляры. Некоторые UML-диаграммы, описывающие *поведение* системы, изображают один из возможных вариантов (экземпляров) поведения, следовательно, не обладают необходимой всеобщностью.

2.2. UML – прожиточный минимум для программиста

По большому счету любой программист, не зная UML, знаком со многими компонентами UML на практике, поскольку реальности объектно-ориентированного программирования отражены в соответствующих компонентах. Кроме того, практика проектирования программного обеспечения в среде

объектно-ориентироанного программирования (ООП) формирует соответствующие абстракции. Рассмотрим подробнее.

Технология ООП работает на логическом уровне абстрактных представлений и физическом – описании принципов работы и механизма их реализации в языке. Например, на уровне реализации в Си++ класс – это набор скомпилированных функций-методов, объекты – области статической (именованной) или динамической (адресуемой указателем) памяти, в которой код интерпретирует объявленные данные объекта. На уровне последовательности команд никакого объектно-ориентированного и структурированного кода вообще нет, ибо любой компилятор реализует компоненты ООП в рамках системы команд традиционной фон-неймановской архитектуры, т. е. в сущности, на уровне блок-схем.

На логическом уровне в процессе проектирования программист оперирует более абстрактными понятиями. Например, класс для него – описание некоторой сущности в виде набора данных и методов, реализующих элементы его поведения.

Поэтому при описании компонент UML для программиста есть постоянная возможность апеллировать к знаниям принципов ООП, почерпнутых из практики программирования. Например, отношение обобщения соответствует механизму наследования, а отношение реализации – созданию объекта класса, если класс рассматривать как порождающую сущность.

Компоненты UML

Модели UML покрывают все аспекты разработки системы, в том числе бизнес-процессы предметной области, функционал, представление данных, параллелизм и синхронизацию, архитектуру, логическую структуру ПО, физическую структуру системы и т. д. Предметом моделирования является не только программная система, но и материальное окружение, а также сам процесс ЖЦ разработки.

Описание любой системы состоит из **структуры** (статики) и **поведения** (динамики). Структура состоит из элементов (сущностей) и связей между ними. На этой основе создан набор компонент UML:

- **сущности** – «имена существительные», статические компоненты модели;
- **отношения** – взаимосвязи сущностей;
- **диаграммы** – описание некоторой части модели в виде набора компонент, диаграммы описывают структуру или поведение, т. е. статику или динамику системы;

- **общие механизмы** – семантика, смысл за пределами формальной модели, расширение UML:
  - **спецификации** (дополнения) – задний план (background), текстовое или формализованное описание деталей семантики, свойственной данному компоненту, вплоть до его реализации на формальном языке;
  - **стереотипы** – общепринятый подвид компонента UML;
  - **помеченные значения** – заданная пара «стандартное свойство–значение» для компонента. Например, пара *location* = <*имя*> обозначает метку размещения компонента;
  - **ограничения** – общепринятые свойства компонента, ограничивающие его поведение или возможности использования. Например, ограничение *complete* для обобщения (абстрактного класса) обозначает невозможность создания в модели новых производных классов помимо имеющихся;
  - **классификатор-экземпляр** – основная форма абстрагирования, основанная на описании многообразия в категориях «абстрактная сущность–экземпляр реализации»: наличие в модели описания абстрактных сущностей, порождающих в динамике множества реализаций.

Сущности

Сущности – имена существительные, основные элементы диаграмм UML:

- **структурные сущности**: классы, активные классы, интерфейсы, прецеденты и кооперации, компоненты, узлы;
- **поведенческие сущности:** взаимодействие и автомат;
- **группирующие сущности** – пакет;
- **комментирующие сущности** – аннотации. Включают текст, не привязанный к элементам модели, заметки (*Note*) с текстом, связанные с элементами модели (*NoteLink*), прямоугольники, полуовалы и овалы с текстом.

Классификатор

Основой любого абстрагирования является пара категорий «**абстракция–экземпляр реализации**». В процессе разработки программист постоянно сталкивается с этой парой:

- *код программы–исполнение*: программа как код и экземпляр исполнения с конкретными входными данными, исполняемый файл программы – процесс в операционной системе;

- *тип данных–переменная*: тип данных как описание формы представления данных и переменная как область памяти, содержащая значение определяющего ее типа;
- *класс–объект как экземпляр класса*: аналогия пары «тип данных–переменная» применительно к классам. При этом в представлении объекта как экземпляра класса есть некоторое лукавство. Память данных объекта действительно является экземпляром для типов данных его свойств, а программный код методов разделяется.

Для программиста просто необходимо распространить это понятие на другие компоненты описания системы – сигнал, прецедент, компонент, узел. В UML категории «классификатор–экземпляр» используются для следующих сущностей:

- класс–объект;
- интерфейс–реализация, т. е. присоединение интерфейса к классу;
- тип данных–переменная;
- сигнал;
- прецедент;
- компонент;
- узел.

В диаграммах модели могут быть описаны взаимосвязи и поведение как классификатора, так и отдельных его экземпляров. В качестве примера рассмотрим двусвязный список и его представление в виде диаграммы классов и диаграммы объектов (рис. 2.2).

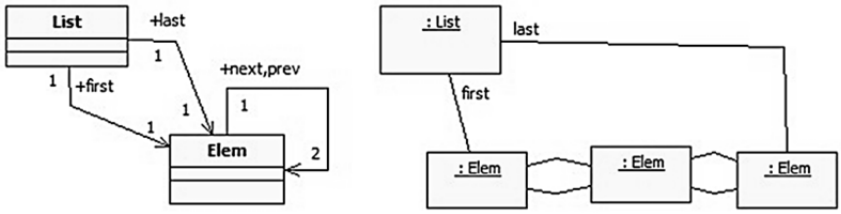


Рис. 2.2. Диаграммы классов и объектов для двусвязного списка

Конкретная линейная топология в диаграмме классов *не может быть отражена*, так как она не определяет взаимосвязи конкретных экземпляров класса *Elem*, например двоичное дерево будет иметь аналогичную диаграмму классов.

Философия и программирование. Объективный идеализм (Платон, Гегель) рассматривает *абсолютную идею* как внематериальную сущность. Любой предмет материального мира является своего рода отпечатком этой идеи

на косной материи. Вообще во многих философских, религиозных и художественных воззрениях идеальный *образ, замысел* (*промысел*) являются предтечей сотворения реальности. Ассоциации с вышеперечисленным налицо.

Стереотипы классов, активные классы

Стереотипы классов отражают реальности, с которыми сталкиваются программисты на уровне операционной системы (ОС) или среды программирования:

- **process** – полноправный процесс в ОС (активный класс);
- **thread** – поток управления внутри процесса (активный класс);
- **exception** – исключение;
- **metaclass** – класс, объекты которого являются классами (описатель классов *Class* в метамодели Java);
- **utility** – класс с открытыми данными и методами (*public*);
- **powertype** – класс, генерирующий объекты любых своих производных классов (фабрика);
- **signal** – асинхронное внешнее событие;
- **type** – абстрактный класс, не создающий объектов, используется для спецификации структуры и поведения;
- **enumeration** – перечислимый тип.

В диаграммах устойчивости (*Robustness*) используются специальные значки для стереотипов классов – компонент программной архитектуры:

- **control** – ориентированный на управление – поведение, действие;
- **boundary** – ориентированный на внешнее взаимодействие;
- **entity** – ориентированный на данные.

Активные классы – классы, объекты которых являются самостоятельными потоками управления (*Thread*). Обратите внимание на одну тонкость. Активный класс порождает поток (*Connector* и *Client* на рис. 2.3), который может вызывать методы в других классах.

Несмотря на то что метод будет вызываться *от имени* потока, содержащий его класс (*Server, Resource*) не будет являться активным.

Отношения

Между описанными выше сущностями в системе устанавливаются связи – **отношения** (рис. 2.4)**.** Связи могут быть различного типа: структурные, поведенческие, родовые (развитие и реализация). В любом отношении, даже если оно равноправно, выделяются ведущая и ведомая сущности (пара **источник–цель, причина–следствие**).

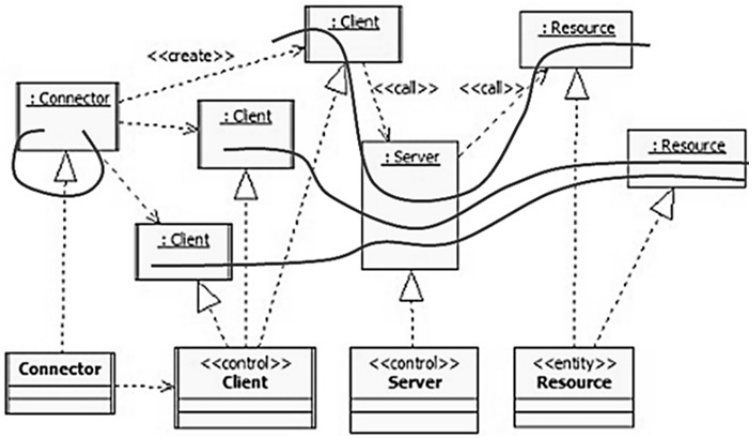


Рис. 2.3. Активные классы

Рис. 2.4. «Сущность–источник» и «сущность–цель» в отношении

Отношение вида **ассоциация** (рис. 2.5) относится к структурной (статической) части описания системы. Ассоциация обозначает *постоянную существенную связь* между сущностями, имеющую место в течение всего времени существования системы.

Если ассоциация установлена между сущностями-классификаторами, то это означает установление ассоциации между отдельными парами экземпляров разных сущностей. В этом случае определяется **кратность** ассоциации, т. е. у экземпляра на каждом конце ассоциации оговаривается допустимое количество связей с экземплярами противоположной сущности: 0,1,* – любое, а также интервалы: 0..1,1..* и т. п.

Структурный характер отношения-ассоциации предполагает, что в любой момент времени возможен *выбор или переход по ассоциации от одного экземпляра к любому из экземпляров или к их группе в противоположной сущности.* Это является методологической основой ассоциации, отличающей ее от других видов.

Рис. 2.5. Отношения: зависимость, ассоциация

Для программиста хорошей аналогией к отношению-ассоциации является взаимосвязь объектов в структуре данных – возможность одного объекта ссылаться на другой. Причем способ связи не имеет значения. Прямая ссылка, индекс, ключ в таблице с операциями линейного или двоичного поиска по ключу или хеширование – все это вписывается в UML в понятие ассоциации.

Тем не менее в UML к ассоциации не следует подходить утилитарно, как средство высокоуровневого представления структуры оно не зависит от реализации. Ассоциация обладает следующими свойствами:

- указанная выше кратность;
- имя ассоциации в сущности;
- возможность навигации (перехода) по ассоциации в заданном направлении;
- видимость – модификаторы доступа private, public, protected, package, аналогичные принятым в любой среде ООП;
- возможность привязать к каждому экземпляру ассоциации объект ассоциированного класса, т. е. нагрузить ассоциацию некоторым функциональным объектом;
- определить квалификатор – ключевое поле перехода по ассоциации.

Хотя способ реализации ассоциации в UML принципиально не оговаривается, имеются виды ассоциаций – **агрегация и композиция,** которые указывают на *целостность* пары сущностей источника и цели:

- **композиция** – экземпляр источника владеет экземплярами целевой сущности, экземпляр целевой сущности создается / уничтожается источником

и не может существовать вне связи с источником. Символ композиции – закрашенный ромбик;

- **агрегация** – экземпляры источника и цели рассматриваются как единое целое по отношению к ассоциации, но принцип владения не применим, имеется возможность независимого существования экземпляров источника и цели. Символ агрегации – прозрачный ромбик.

Наличие в системе объектов определенного класса не означает, что все они находятся в одинаковых структурных отношениях с другими классами. Например, на рис. 2.6 имеют место фактически три различных вида объектов **класса B** в зависимости от того, к какой композиции они принадлежат.

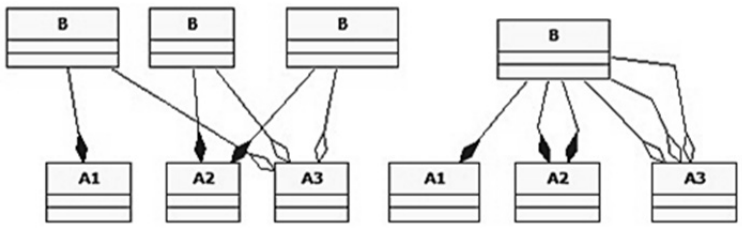


Рис. 2.6. Агрегация и композиция: ограничения структурного представления

Агрегации этих объектов с **классом A3** (левый рисунок) совпадают с соответствующими композициями в том смысле, что все объекты одной композиции могут принадлежать к одному типу агрегации. Если же объединить все группы объектов в один класс (правый рисунок), то такая взаимосвязь будет неочевидной. Можно сказать, что это свойство переходит на уровень *семантики ассоциации* и не отображается в структуре.

Отношение вида **зависимость** (рис. 2.5) означает связь или взаимодействие сущностей, относящиеся к поведению системы, т. е. имеет место ограниченное по времени взаимодействие источника и цели. Также к зависимостям относятся разнообразные причинно-следственные связи, не связанные со структурой системы. Чтобы понять, что относится к зависимости, достаточно перечислить стереотипы – общепринятые типы зависимостей. Стереотипы зависимостей для классов:

- **bind** – экземпляр шаблона с параметрами;
- **derive** – данные источника являются фиктивными и вычисляются по данным цели;
- **friend** – исключение прав доступа, источник имеет исключительные права доступа к цели;

- **instanceof** – принадлежность цели (экземпляра) источнику (классификатору). В терминологии программирования это звучит так: цель является объектом класса-источника или одного из его производных классов;
- **instantiate** – источник создает экземпляр в цели. Зависимость установлена между классами: объект класса-источника создает объект класса-цели;
- **powertype** – источник генерирует объекты любых производных классов цели (см. раздел 3.3, шаблон «фабрика»);
- **refine** – детализация сущности – источника при проектировании, используется при документировании процесса разработки и версий проекта;
- **use** – источник использует семантику цели, например класс-источник использует данные класса-цели;
- **call** – программный код источника использует код цели, т. е. вызывает в ней метод;
- **trace** – источник является историческим потомком цели, аналогично *refine*, но при этом не происходит изменения источника, например, при переименовании.

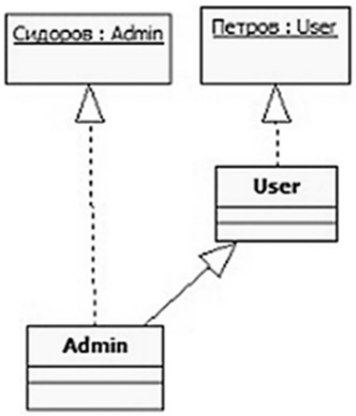


Рис. 2.7. Отношения: обобщение и реализация

Стереотипы зависимостей для объектов:

- **become** – более поздний по времени объект-источник, позволяет отображать на диаграмме классов изменение структуры связей объектов во времени;
- **call** – код источника вызывает метод в цели;
- **copy** – клонирование, цель является копией источника.

Обобщение и реализация связаны с развитием сущностей или порождением их экземпляров (рис. 2.7):

- **реализация** – целевая сущность представляет собой экземпляр исходной сущности – классификатора;
- **обобщение** – исходная сущность представляет собой производный класс, развитие целевой сущности – базового класса.

Диаграммы

Сущности и отношения являются компонентами, из которых строится описание системы. Единица описания – диаграмма. Существует значительное количество видов диаграмм, описывающих различные аспекты структуры и поведения системы. В то же время один и тот же вид диаграмм может использоваться в различных моделях на разных этапах процесса разработки. Основой классификации является деление диаграмм на диаграммы описания *структуры* и *поведения* (рис. 2.8).

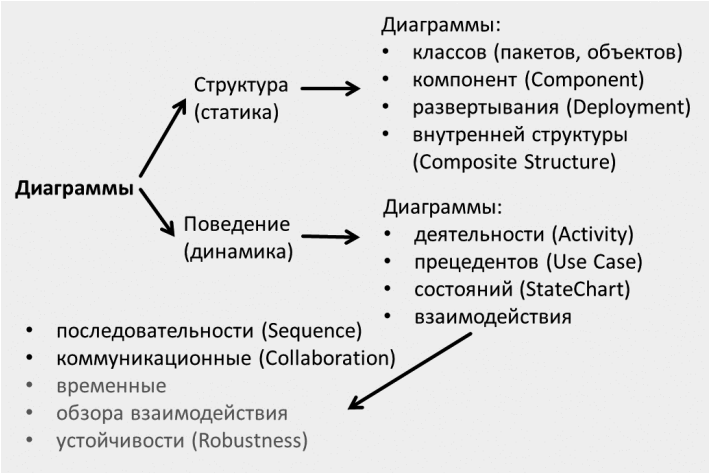


Рис. 2.8. Виды UML-диаграмм

Диаграммы классов (Class)

Диаграмма классов наиболее понятна программисту, поскольку содержит значительное количество компонент, имеющих аналоги в ООП. Однако ее не следует понимать буквально как исключительно описание структуры кода. Объектно-ориентированное представление является общепринятым для различных моделей в разных дисциплинах жизненного цикла. Среди них имеет место преемственность: модель – диаграмма классов на следующем этапе наследуется от предыдущего этапа. Виды моделей, использующих диаграмму классов:

- диаграмма классов предметной области (рис. 2.9);
- диаграмма классов анализа;
- диаграмма классов проектирования;
- диаграмма классов реализации – структура программного кода (рис. 2.10).

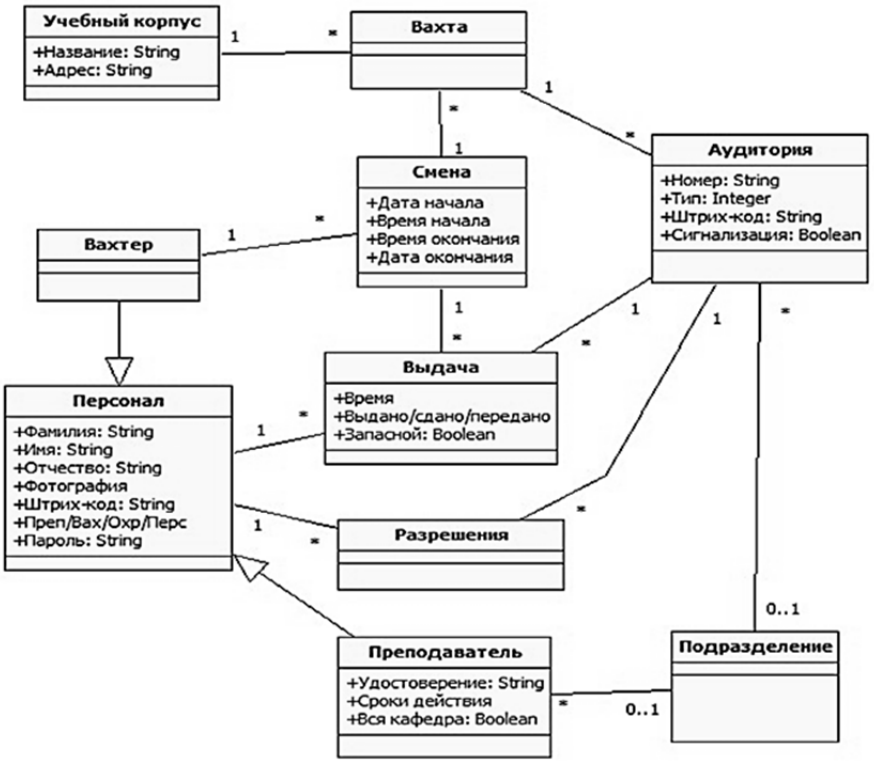


Рис. 2.9. Диаграмма классов – модель предметной области

Элементами диаграммы классов являются:

- классы, объекты, пакеты, подсистемы, стереотипы классов – исключения, сигналы;
- все виды отношений: ассоциации, агрегации, композиции, обобщения и реализации, ассоциированные классы;
- интерфейсы и классы-реализации, их присоединяющие.

Диаграммы внутренней структуры (Composite Structure)

Иногда удобно отойти от канонической объектно-ориентированной модели и рассматривать класс не как набор свойств-методов, а как набор взаимодействующих *частей,* а классы, в свою очередь, объединять в подсистемы по функциональному признаку. Тогда получается диаграмма внутренней структуры, элементами которой являются (рис. 2.11):

- подсистемы;
- входящие в них *классы*;

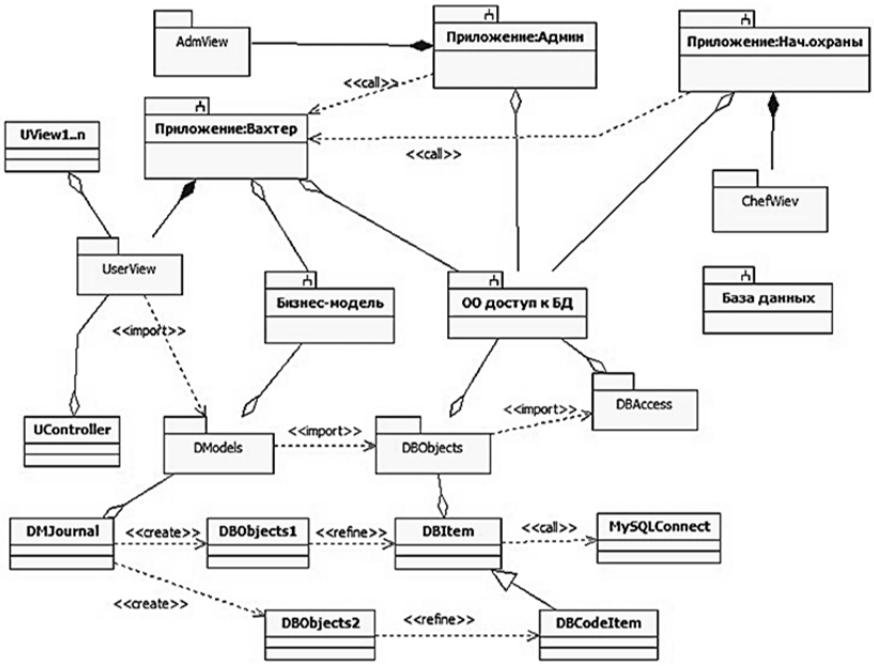


Рис. 2.10. Диаграмма классов – структура программного кода

- части классов (part);
- порты (port) – именованные функциональные входы класса и подсистемы;
- соединения (connector) – связи между частями и портами;
- интерфейсы и их реализации – классы и части классов;
- зависимости между классами и их частями.

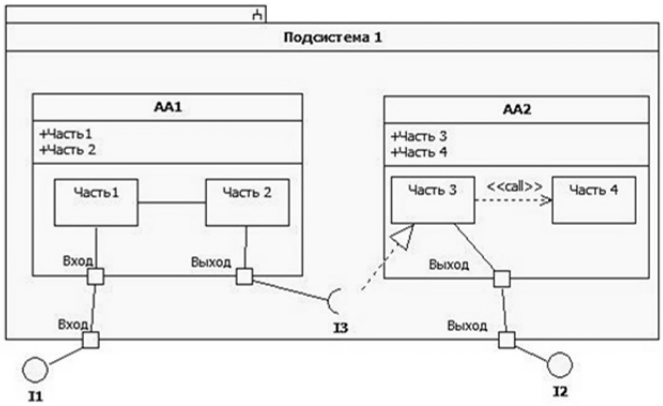


Рис. 2.11. Диаграмма внутренней структуры

Диаграмма внутренней структуры отражает логическую структуру системы в виде модульной конструкции, состоящей из подсистем, классов, частей и их соединений. Фактически это общеизвестная *структурная схема*, привязанная по необходимости к классам.

Иногда элементы диаграммы внутренней структуры включают в диаграмму классов, тогда последняя интегрирует в себя диаграмму внутренней структуры.

Диаграммы компонентов (Component) и размещения (Deployment)

Диаграмма компонентов (рис. 2.12) отображает структуру системы в виде составляющих ее программных компонент, компонент кода и используемых ими артефактов. Ее элементами являются:

- **компоненты** и **экземпляры компонентов** – физический модуль системы с содержанием, имеющим отношение к исполнению кода, имеет стереотипы *document, file, executable, library, table, source*. Специфические и важные компоненты называются *артефактами*;
- пакет программного кода;
- интерфейс, порт (*port*), соединение (*connector*), часть (*part*);
- отношения: ассоциация, зависимость, реализация.

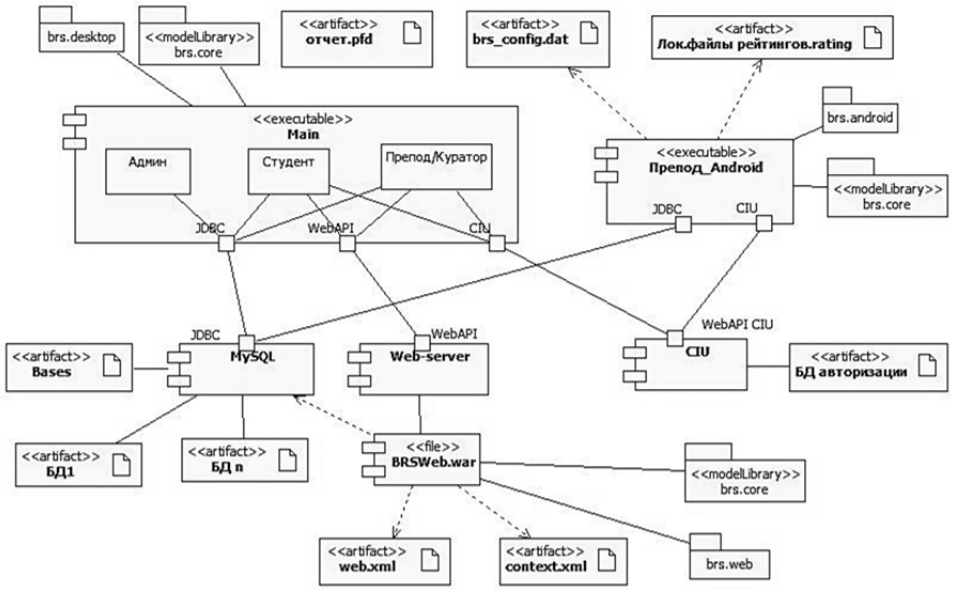


Рис. 2.12. Диаграмма компонент

Диаграмма размещения (рис. 2.13) имеет тот же самый набор выразительных средств за исключением сущности – компонент: вместо нее фигурирует **узел (node)** – физическая единица программной системы, аппаратных и программных средств. Диаграмма изображает общую структуру размещения (узел–классификатор) либо конкретный вариант размещения (узел–экземпляр).

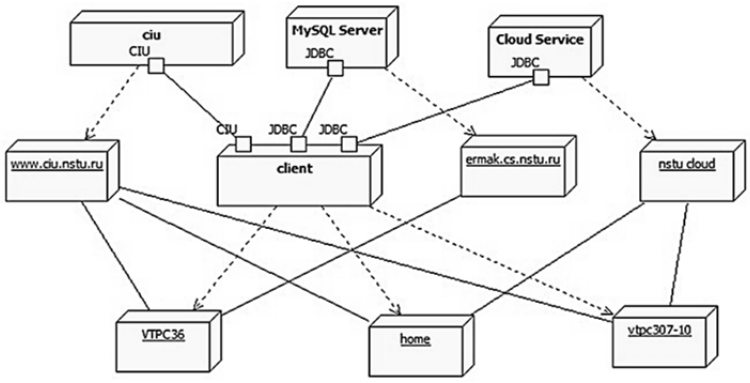


Рис. 2.13. Диаграмма размещения

Диаграммы последовательностей (Sequence)

Основной иллюстрацией поведения системы является описание развертки ее работы во времени. Но фактор времени как таковой в моделях UML отсутствует, элементом описания поведения является **взаимодействие** как направленная причинно-следственная связь сущностей или их экземпляров.

На **диаграмме последовательностей** (рис. 2.14) каждому экземпляру сущности (обычно объекту) соответствует **линия жизни**. На линии жизни отображается активность экземпляра, вызванная поступающими сообщениями. Сообщения могут интерпретироваться двояко:

- как передаваемые данные, в ожидании которых принимающий экземпляр не активен (ожидает или блокирован), т. е. в виде событийной модели;
- как переход потока управления из вызывающего экземпляра в вызываемый.

В соответствии с этим имеют место стереотипы сообщений:

- вызов, переход потока управления – **call**;
- возврат потока управления – **return**;
- передача сообщения – **send;**
- создание экземпляра – **create;**
- уничтожение экземпляра – **destroy.**

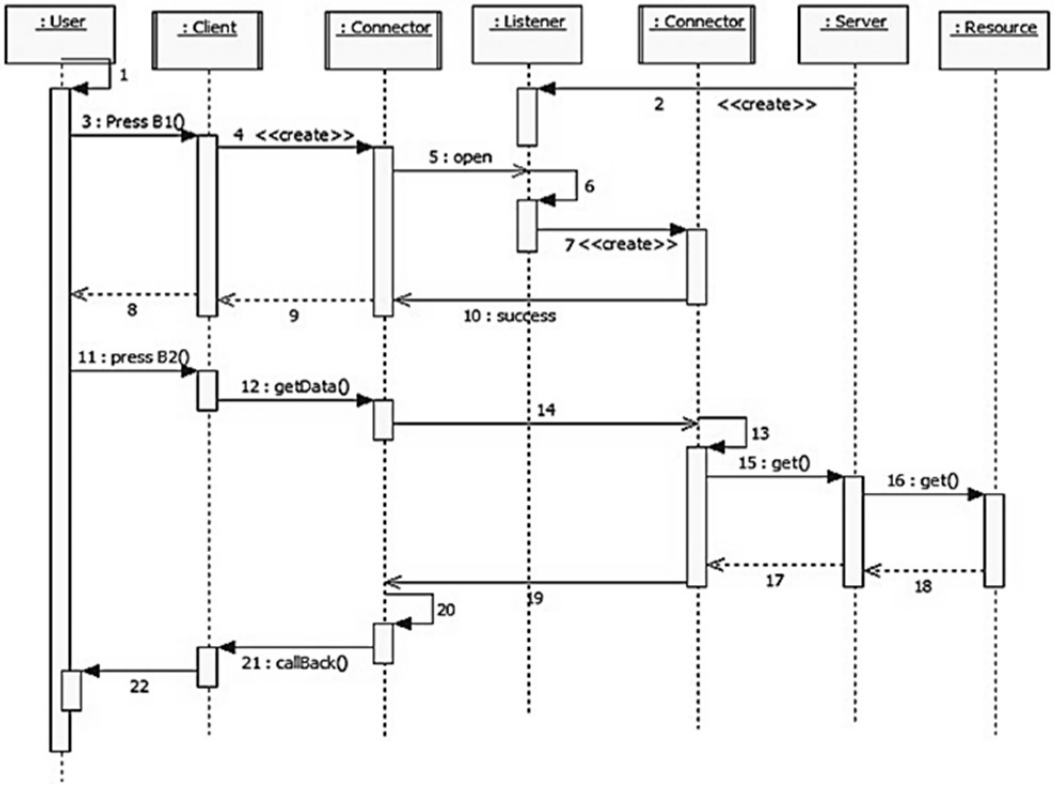


Рис. 2.14. Диаграмма последовательностей

Диаграмма последовательностей может использовать сущности «роль классификатора» (*ClassifierRole*) (см. диаграммы устойчивости) со стереотипами *caseWorker, worker, internalWorker, boundary, control, entity, process* (рис. 2.15). Такие диаграммы применяются для описания системы на уровне бизнес-процессов, функционала и архитектуры.

Часть диаграммы взаимодействия можно оформить в виде **оператора взаимодействия**, состоящего из частей – **операндов.** Это используется для отображения распространенных примитивов программирования. Перечислим некоторые из них:

- **alt** – ветвление, несколько альтернативных фрагментов, выбираемых по значению ключевого условия;
- **break** – досрочное завершение взаимодействия при выполнении условия;
- **critical / region** – критическая секция, синхронизируемый блок, аналогичный *synchronized* в Java;

- **ignore** – игнорирование любых сообщений, указанных в заголовках операндов;
- **consider** – обработка сообщений, указанных в заголовках операндов;
- **opt** – фрагмент, исполняемый при ключевом условии, ветвление без *else*;
- **par** – все фрагменты выполняются параллельно;
- **loop** – цикл, фрагмент представляет собой тело цикла;
- **neg** – вызывается при обнаружении ошибки в процессе взаимодействия;
- **seq** – слабое следование, порядок исполнения операндов может быть любым, внутри операнда последовательность сохраняется;
- **strict** – строгая последовательность, все взаимодействия внутри каждого операнда для всех линий жизни не должны выходить за рамки операнда;
- **ref** – ссылка на другую диаграмму последовательности (включение).

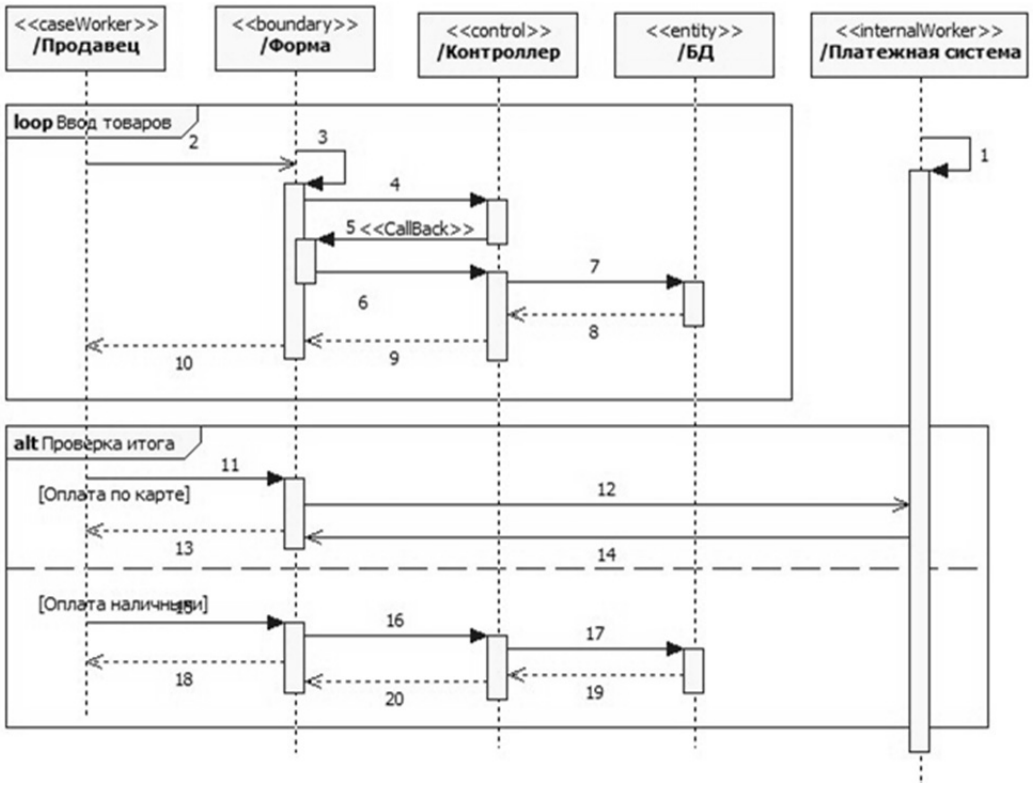


Рис. 2.15. Диаграмма для ролей с операторами взаимодействия

Вся диаграмма может быть оформлена как комбинированный фрагмент **sd,** чтобы на него можно было ссылаться из других диаграмм.

Коммуникационные диаграммы (Collaboration)

Последовательность передачи сообщений между объектами в некоторых случаях можно отобразить непосредственно на диаграмме классов, если они передаются в соответствии с ассоциациями или зависимостями. На коммуникационных диаграммах (рис. 2.16) сообщения обозначаются нумерованными стрелками, привязанными к связям между экземплярами. Предполагается, что они являются элементами отношений, но характер их в данном случае не раскрывается.

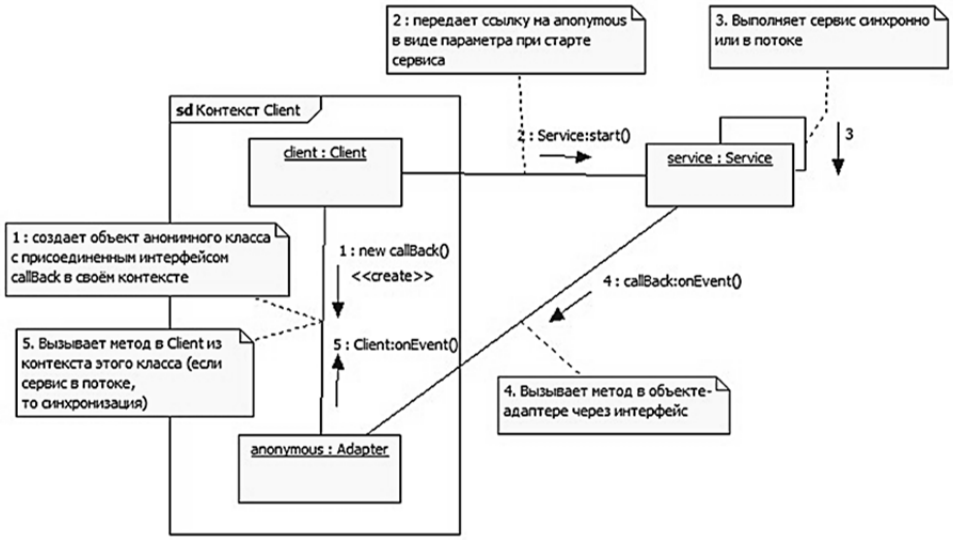


Рис. 2.16. Коммуникационная диаграмма

Коммуникационная диаграмма может использоваться для экземпляров сущностей «роль классификатора» (*ClassifierRole*) аналогично диаграммам последовательности.

Диаграммы устойчивости (Robustness)

Диаграмма устойчивости (рис. 2.17) является вариантом диаграммы классов, который применяется на этапах бизнес-анализа, функционального и архитектурного проектирования.

В качестве сущностей здесь фигурируют элементы внешнего окружения, а также элементы архитектурного описания системы, которые удается выделить на уровне бизнес-анализа и функционального анализа. Это специфицируется стереотипами **ролей классификатора (ClassifierRole)**, а на диаграмме – соответствующими значками:

- **caseWorker** – сотрудник для связи с окружением, участник бизнес-процесса, непосредственно взаимодействующий с системой;
- **worker, internalWorker** – сотрудник, участник бизнес-процесса;
- **businnesEntity** – бизнес-сущность;
- **boundary** – внутренний класс, ориентированный на взаимодействие с окружением системы;
- **control** – внутренний класс, ориентированный на управление (действие, поведение);
- **entity** – внутренний класс, ориентированный на представление данных.

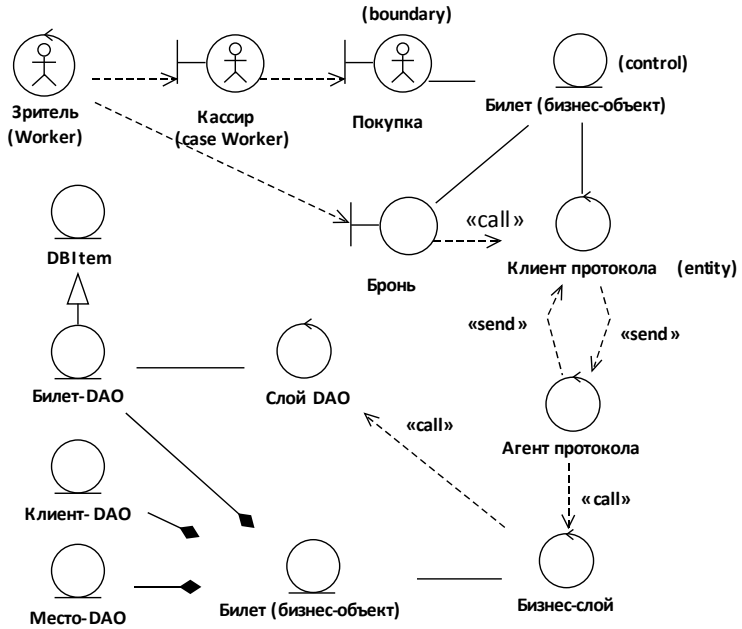


Рис. 2.17. Диаграмма устойчивости

Это позволяет создавать модели с различным «процентным соотношением» описания функционала, архитектуры и реализации.

Диаграммы деятельности (Activity)

Диаграммы последовательности больше подходят для описания последовательных взаимодействий, когда активность в одном экземпляре является причиной аналогичной активности в другом. Если же требуется описать такие вещи, как равноправный параллелизм, взаимную синхронизацию параллельных ветвей, то наиболее адекватным средством является диаграмма деятельности.

Диаграмма деятельности представляет собой своеобразное сочетание формальной системы описания параллельного поведения автоматов – **сетей Петри** и традиционных блок-схем. Последние являются средством описания алгоритма в рамках отдельного потока управления.

Семантика диаграммы деятельности сложнее, чем у других диаграмм (рис. 2.18). Постараемся ее описать в терминах, близких к ООП. Диаграмма состоит из узлов, соединенных дугами. Диаграмма одновременно работает в концепции **потока команд (управления)** и **потока данных (объектов)**. Модель потока управления используется по умолчанию, модель потока данных – только в **объектных узлах** и смежных с ними узлах и дугах. При этом дуги, смежные с объектными узлами, обозначаются пунктиром (объектные дуги).

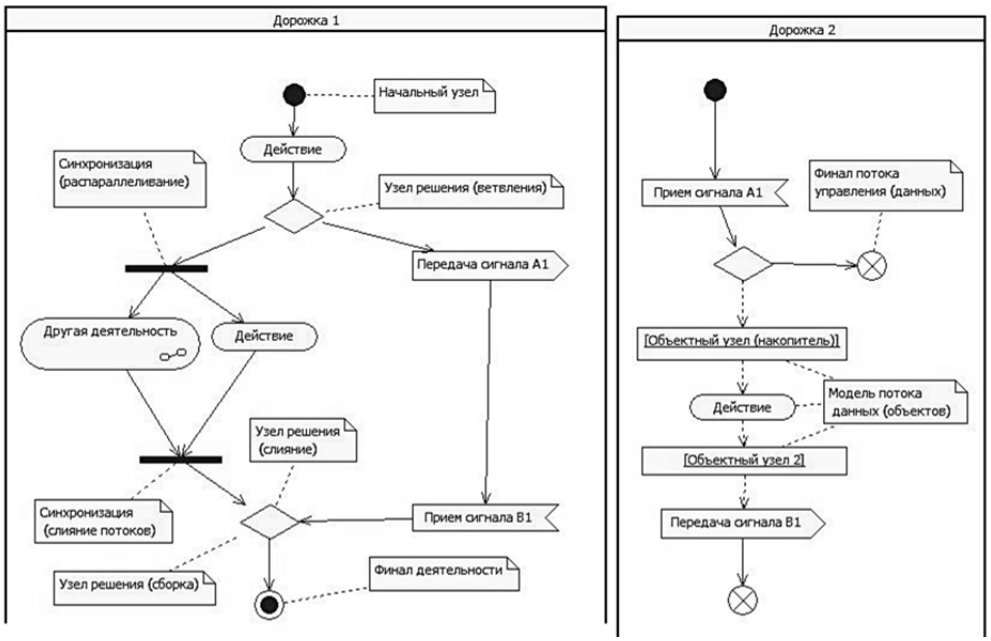


Рис. 2.18. Основные элементы диаграммы деятельности

Модель в диаграмме деятельности является по определению многопоточной и использует понятие **маркера**, заимствованное из сетей Петри. Маркер (*token*) – это текущее состояние отдельного потока управления, привязанное к дуге диаграммы. В обычной блок-схеме последовательного алгоритма маркер – единственный, в диаграмме деятельности подразумевается, что количество их может быть любым в зависимости от интенсивности активизации деятельности, т. е. поступления маркеров в начальный узел.

Узлы действия являются *синхронизированными*, т. е. последовательно обрабатывают исполняющие их потоки, перемещая маркеры из входных дуг в выходные. Таким образом, маркеры могут накапливаться на входных дугах узлов, что соответствует синхронизации потоков к единичным (неделимым) ресурсам. Справедливости ради отметим, что сами маркеры на диаграмме не отображаются, т. е. диаграмма описывает не какое-то текущее состояние деятельности, а деятельность в целом.

Все вышесказанное относится к моделям, в которых в каком-либо фрагменте создается внутренний параллелизм, связанный с разделением идентичными потоками общих ресурсов. Значительно чаще моделирование ограничивается внешним параллелизмом, когда разные участки диаграммы отображают принципиально разные потоки (функциональные виды деятельности, физические сущности).

В объектных узлах работает другое представление. Маркер – это объект (экземпляр сущности), и объектный узел способен их накапливать. Свойства и особенности объектных узлов:

- множественность потоков управления в объектном узле превращается в множественность объектов данных;
- объектные узлы используются для обозначения компонент, где происходит явное накопление данных, создаваемых различными потоками (очередей, пулов);
- иногда объектные узлы используются не для явного накопления маркеров, а для обозначения факта присутствия указанного типа объекта на данном этапе деятельности.

В диаграмме деятельности имеются следующие виды узлов потока управления:

- *действие*;
- *вызов деятельности* – исполнение указанной диаграммы деятельности как целого;

- *начальные узлы деятельности*, в каждом из них в начале цикла моделирования помещается по маркеру;
- *финальный узел потока управления* завершает поток управления (уничтожает маркер), не завершая деятельности в целом;
- *финальный узел деятельности* – достижение его маркером завершает деятельность в целом;
- *узел решения* (решающий) – выбор одного из альтернативных направлений по выходным дугам, сборка маркеров по входным дугам, т. е. количество потоков при срабатывании узла *не меняется*;
- *узел синхронизации* – распараллеливает входной поток на несколько выходных (по выходным дугам), соединяет (синхронизирует) несколько потоков по входным дугам в один;
- *узлы передачи и приема сигнала* предполагают передачу сигнала в некоторую внешнюю среду, а также ожидание его приема оттуда.

Замечание по теме. Узел действия при наличии нескольких входных или выходных дуг играет роль узла синхронизации по отношению к маркерам в этих дугах.

Еще одним элементом диаграммы является **дорожка.** Она соответствует физической сущности, роли, классу, в рамках которого осуществляется включенная в него деятельность.

В диаграммах деятельности, особенно если это касается моделирования бизнес-процессов и функционала, нельзя отождествлять модель с приведенными аналогиями из области программирования и реальностями реализации. Дуга, соответствующая потоку управления, легко может переходить с дорожки на дорожку, например от дорожки заказчика к дорожке интернет-магазина. На функциональном уровне моделирования это вполне допустимо, поток управления в модели на разных шагах может иметь различную физическую природу.

В моделях бизнес-процессов и описания функционала диаграммы деятельности могут быть удобным средством спецификации сценариев и моделирования предметной области. Что же касается моделей проектирования и конструирования, то иллюстрация шаблонов параллелизма с помощью диаграмм деятельности является не только тяжеловесной, но иногда и некорректной с точки зрения формального описания поведения модели в диаграмме. Например, при моделировании передачи сообщения с тайм-аутом (рис. 2.19) в первом приближении все правильно, имеются два потока: обмена и тайм-аута.

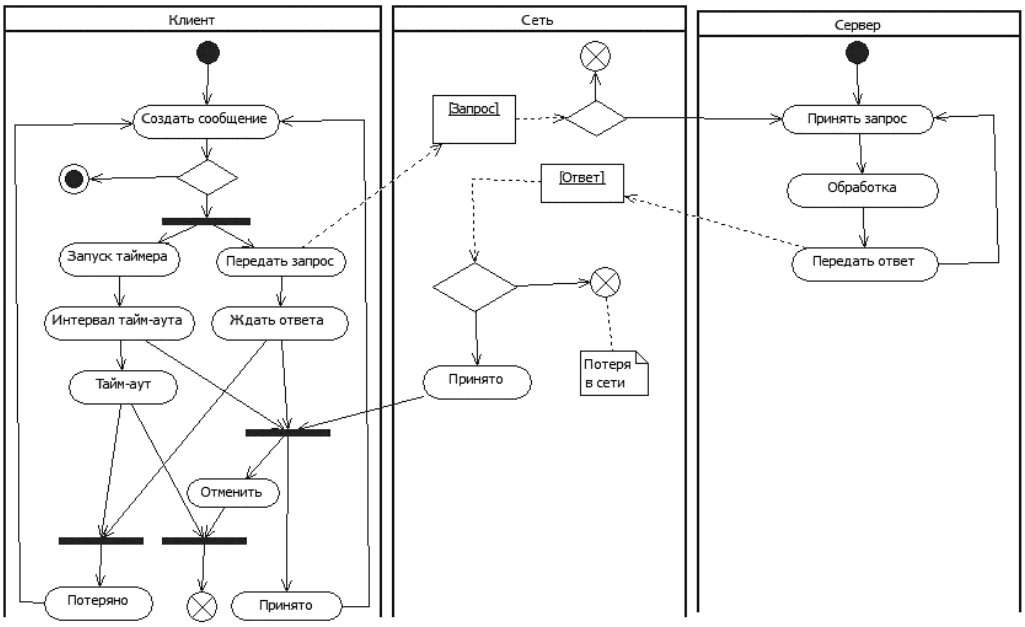


Рис. 2.19. Диаграмма деятельности: передача сообщения с тайм-аутом

При приеме ответного сообщения тайм-аут отменяется. Однако это происходит в модели по истечении тайм-аута, *но не сразу*. Это связано с тем, что в данной модели не отражен процесс прерывания действия – *интервал тайм-аута*. Добавление соответствующих средств (область прерывания в UML2) еще более утяжелит ее.

Диаграммы состояний (Statechart)

В основе диаграммы состояний лежит модель *конечного автомата* (КА). Распространенной графической интерпретацией автомата является система состояний / переходов. Переход из состояния в состояние происходит по некоторому условию (символу) и сопровождается генерацией выходного сигнала (символа).

Фактически конечный автомат представляет собой программу, лишенную данных. Единственный элемент памяти автомата – его текущее состояние. Именно поэтому его диаграмма состояний более полно визуализирует его поведение. Неявная логика работы программы может определяться также текущим состоянием переменных, чего лишен автомат.

Конечные автоматы используются в программировании для описания поведения сущностей (классов), управляемых событиями от нескольких источников. Объект класса имеет набор состояний, относительно которых определяется его поведение и производится манипулирование им при обработке событий со стороны других сущностей.

Естественно, что КА в диаграмме состояний отличается от канонической модели большей технологичностью. Основной набор элементов (рис. 2.20) включает:

- начальное и конечное состояния;
- простое промежуточное состояние с наборами действий при входе в состояние (*EntryActions*), выходе из состояния (*ExitAction*) и нахождении в состоянии (*DoActions*);
- переход с набором условий срабатывания (*Triggers*) и действий (*Effects*).

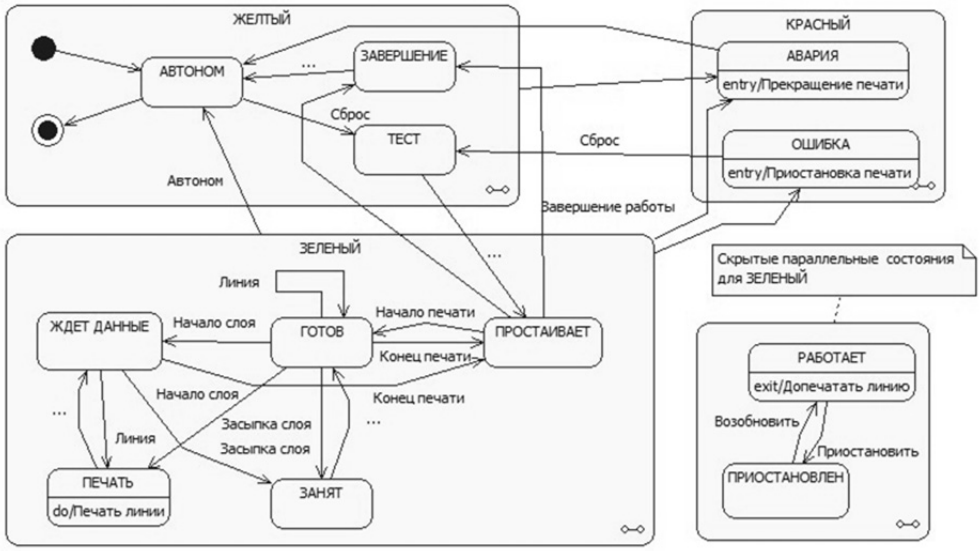


Рис. 2.20. Диаграмма состояний

Дополнительные элементы диаграммы состояний связаны с введением в КА иерархии – составных состояний (рис. 2.21). Составные состояния представляют собой на верхнем уровне одно состояние, которое раскрывается в виде отдельной диаграммы. При этом возможно создание групп параллельных состояний, т. е. текущее состояние подавтомата может быть двойным, тройным и т. д.

Выход из составного состояния может быть выполнен как через финальное состояние, так и помимо него. В последнем случае это выглядит как прерывание нормального поведения подавтомата. В этом случае при последующем входе автомат оказывается в состояниях *поверхностной* (*ShallowHistory*) или *глубокой истории* (*DeepHistory*).

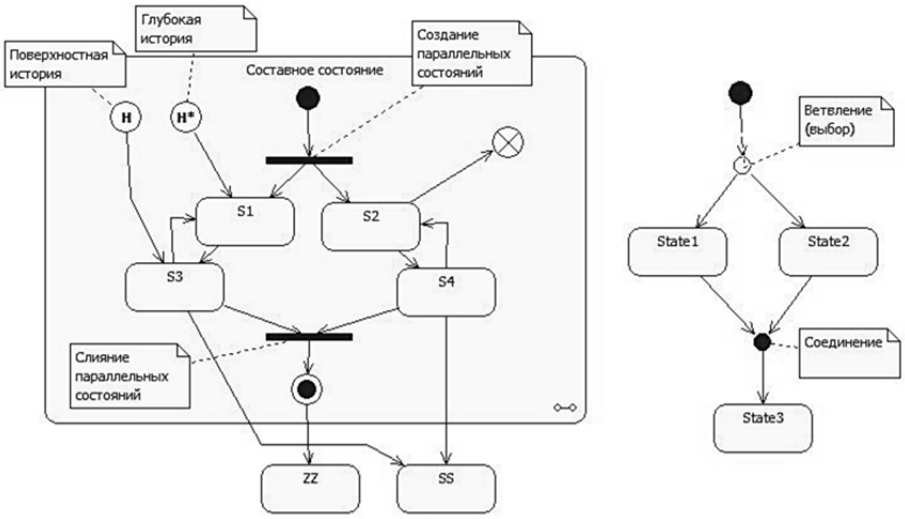


Рис. 2.21. Дополнительные элементы диаграммы состояний

Технологическим элементом являются *точки ветвления* (*ChoicePoint*) и *соединения* (*JunctionPoint*) состояний, которые позволяют сэкономить на одинаковых условиях и действиях в переходах.

Диаграммы прецедентов

Диаграмма прецедентов является фактически перечнем атомарных сценариев взаимодействия – *прецедентов* и связанных с ними пользователей – *актеров* (рис. 2.22). Между овалами прецедентов и фигурками актеров устанавливаются связи – ассоциации. Между двумя прецедентами может быть установлена зависимость одного из видов:

- *включение* (*include*) – целевой прецедент является частью прецедента-источника;
- *расширение* (*extend*) – целевой прецедент выполняется при определенных условиях исполнения прецедента-источника.

Между двумя актерами или двумя прецедентами может быть определено отношение обобщения (наследования) – целевой прецедент является более общим, а источник – его расширением.

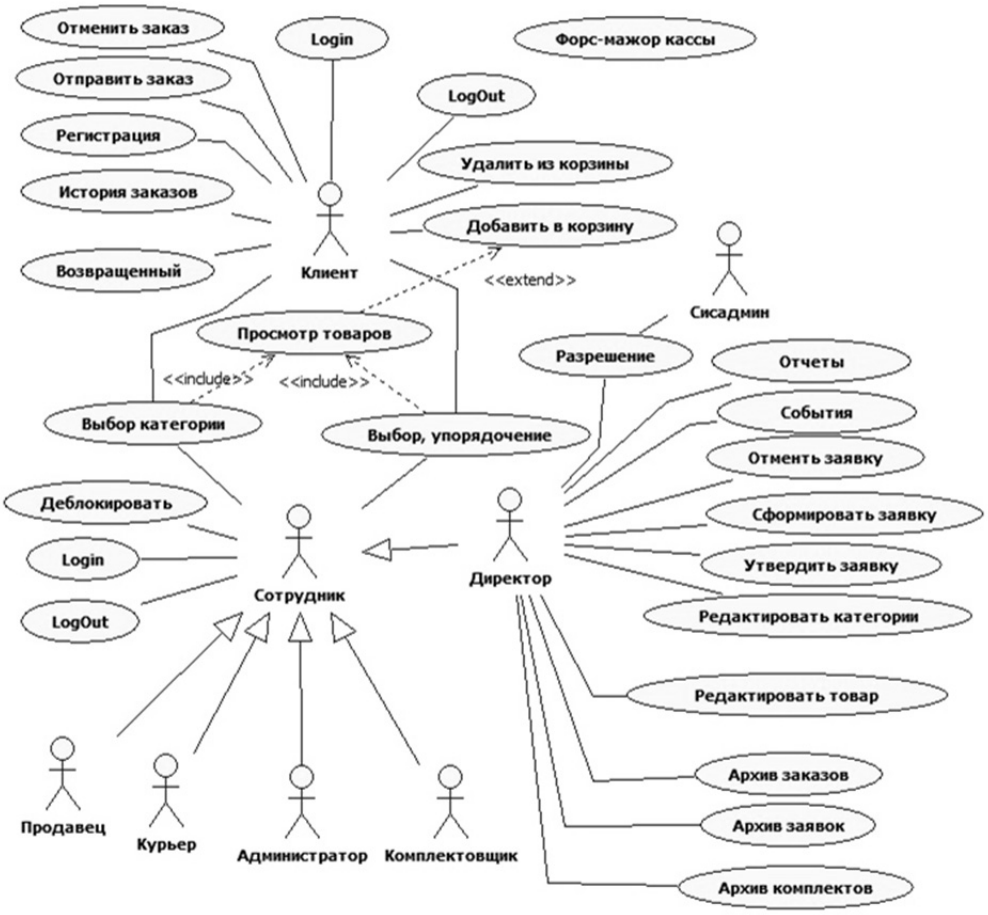


Рис. 2.22. Диаграмма прецедентов

Замечание по теме. Актером может быть какая-либо внутренняя сущность или компонента системы, которая выступает как виртуальный пользователь, инициирующий внятные сценарии. Например, периодически выполняемое архивирование данных может быть представлено актером-таймером, инициирующим соответствующий сценарий.

ГЛАВА 3
ГРАМОТНОЕ КОНСТРУИРОВАНИЕ И КОДИРОВАНИЕ

3.1. Алгоритмы и структуры данных. О пользе общего образования

Базовое образование в программировании кроме знаний собственно инструментов (языка, среды разработки) подразумевает также рамочные знания по структурам данных и алгоритмам (см. раздел 1.2). На практике знания нужны именно *рамочные:* понимание сущности, основные артефакты, границы применения, взаимосвязь с другими формальными системами. Но не для того, чтобы сэкономить несколько процентов ресурса, а чтобы видеть возможные варианты решения задачи с качественно разными результатами.

Трудоемкость алгоритмов

Трудоемкость алгоритма имеет прямое отношение к производительности и масштабированию.

> Трудоемкость программы (алгоритма) – это зависимость количества массовых операций (сравнения, обмены, сдвиги, повторения цикла и т. п.) от размерностей обрабатываемых данных (*N*).

Напомним основные аспекты понятия трудоемкости:

- трудоемкость имеет размерность количества, а не времени и не привязана к аппаратному и программному окружению. Она характеризует затратность программы не для случая «здесь и сейчас», а по отношению к масштабированию ее количественных параметров: объемов данных, количеству пользователей, транзакций и т. п.;

- трудоемкость определяется отдельно для каждого вида операций и характеризует не столько программу, сколько сам алгоритм;
- трудоемкость может зависеть от входных данных. Поэтому оценка трудоемкости дается для лучшего и худшего случаев, а также в среднем. Свойство программы – иметь различную трудоемкость для разных наборов данных, называется чувствительностью к данным;
- на практике обычно используется оценочная форма трудоемкости, основанная на понятии скорости (степени) роста функции, так называемая O-нотация;
- для характеристик времени исполнения программы используется термин «производительность».

O-нотация определяется как асимптотическое приближении функции трудоемкости при росте N с точностью до произвольного коэффициента пропорциональности. Обычно нас интересует *вид* функции трудоемкости, для чего есть свои основания. Дело в том, что размерности исходных данных меняются в программах в широких пределах – на несколько порядков при отладке программы, ее работе в реальных условиях и масштабировании. Поэтому для функции трудоемкости важно ее асимптотическое поведение при достаточно больших N. Само значение *достаточно большой* размерности N, при котором это приближение оказывается приемлемым, на уровне оценки определить невозможно.

Основные виды трудоемкости порождаются соответствующими программными решениями:

- единичная трудоемкость $O(1)$ – количество операций постоянно и не зависит от размерности. Например, поиск или размещение данных, если их местоположение вычисляется;
- линейная трудоемкость $O(N)$ – обычный линейный цикл;
- степенная трудоемкость $O(N^m)$ – m линейных вложенных циклов;
- логарифмическая трудоемкость $O(\log_2(N))$ – цикл, в котором размерность на каждом шаге удваивается или делится пополам (половинное деление);
- экспоненциальная трудоемкость $O(m^N)$ – рекурсия, порождающая на каждом шаге для каждой задачи m аналогичных задач.

Для задачи диапазон трудоемкости в разных решениях может говорить о многом. Например, простейшая задача поиска по ключу для N элементов имеет несколько решений, различных по трудоемкости:

- линейный поиск с линейной трудоемкостью $O(N)$ – примитивное, но безошибочное решение задачи «в лоб» для любых данных;

- двоичный поиск с логарифмической трудоемкостью $O(\log_2(n))$ – идеальное решение, основанное на делении пополам с гарантированной трудоемкостью для упорядоченных данных;
- хеширование – размещение и поиск с вычислением адреса – единичная трудоемкость $O(1)$ в лучшем случае с возможным вырождением до линейной $O(N)$ в худшем.

Здесь можно перефразировать известную пословицу: *«выше логарифма не прыгнешь»* – разные хитрые способы организации данных и алгоритмы поиска упираются в гарантированную логарифмическую трудоемкость, будь то упорядоченные массивы или двоичные деревья. Аналогично алгоритмы сортировки по своей природе дают $O(N^2)$ для очевидных способов перебора пар и линейно-логарифмическую $O(N\log_2(N))$ для всех изощренных: пирамидальная, рекурсивное разделение/слияние, циклическое слияние. Сказанное не касается фактора использования дополнительной памяти для качественного изменения производительности.

Перечисленные варианты трудоемкости основаны на примитивном анализе поведения программы и представления ее структуры. Необходимо учитывать не только формальное наличие компонент (рекурсия, циклы), но и условия, которые влияют на частоту их выполнения и сопутствующее им количество операций. Например, для различных случаев рекурсивных алгоритмов, когда задача размерности n сводится к задаче меньшей размерности (обычно $n/2$ или $n - 1$) и при этом содержит определенное количество операций, кратное текущей размерности (1 или n), результирующие значения трудоемкости могут иметь практически любой вид, за исключением единичной.

Таблица 3.1

Трудоемкости рекурсивных алгоритмов

	Размерность	Изменение размерности задачи	Трудоемкость
1	$T_N = NT_{N-1} + N$	На каждом шаге рекурсии возникает N задач размерности, меньшей на 1, на каждом шаге число выполняемых операций пропорционально размерности задачи	$T = N!$
1	$T_N = T_{N-1} + N$	С каждым шагом рекурсии размерность задачи уменьшается на 1, на каждом шаге число выполняемых операций пропорционально размерности задачи	$T = N^2/2$

Окончание табл. 3.1

	Размерность	Изменение размерности задачи	Трудоемкость
2	$T_N = T_{N/2} + 1$	С каждым шагом рекурсии размерность задачи уменьшается в два раза при выполнении единственной на этом шаге операции	$T = \log_2 N$
3	$T_N = T_{N/2} + N$	С каждым шагом рекурсии размерность задачи уменьшается в два раза, число операций на каждом шаге пропорционально размерности задачи	$T = 2N$
4	$T_N = 2T_{N/2} + N$	С каждым шагом рекурсии задача разбивается на две, размерность которых в два раза меньше исходной, число операций на каждом шаге пропорционально размерности задачи	$T = N \log_2 N$
5	$T_N = 2T_{N/2} + 1$	С каждым шагом рекурсии задача разбивается на две, размерность которых в два раза меньше исходной, при выполнении единственной на этом шаге операции	$T = 2N$

И последнее. Трудоемкость обычно касается не всей программы, а участков, наиболее критичных по производительности.

Алгоритм и данные – время и пространство программного кода

Крылатая фраза *проигрывая в пространстве, выигрываешь во времени* имеет прямое отношение к программированию: качественное изменение производительности и трудоемкости может быть достигнуто за счет использования дополнительной памяти. Способы такого использования могут быть разными в зависимости от поставленной задачи:

- динамическое программирование – идея состоит в сохранении (кэшировании) промежуточных решений для повторного использования результата при появлении задачи с такими же параметрами. Часто встречается в поисковых алгоритмах, основанных на рекурсивном комбинаторном переборе. Аналогичная идея заключается в запоминании параметров уже найденного оптимального решения, для того чтобы ограничить заведомо неэффективный перебор;

• использование избыточности данных или их удачное размещение. Например, распределяющие сортировки позволяют получить линейную трудоемкость процесса за счет распределения элементов в дополнительной памяти по разрядам ключа (лексикографическая сортировка) или с использованием счетчиков значений (распределяющий подсчет). Причем при использовании лексикографической сортировки на списках дополнительной памяти по отношению к исходной форме представления данных не требуется.

Жадность против тупости

Жадность фраера сгубила.
Пословица

Любая задача имеет различные по эффективности решения. Более эффективные являются и более сложными по структуре кода, следовательно, имеют потенциально бóльшую вероятность появления ошибок. Примитивное решение «в лоб», построенное на простых принципах, например полный линейный перебор в пространстве решений, имеет в этом плане преимущество. Оно также имеет право на существование, если укладывается в текущие требования по производительности и будет удовлетворительным *при масштабировании.*

Оптимизация алгоритмов часто заключается в ограничении числа просматриваемых решений. В рекурсивных алгоритмах это непосредственно видно в структуре кода: если функция порождает несколько вызовов, то их ограничение состоит в определении дополнительных условий, при которых они происходят. Здесь ключевым является понятие «**жадный алгоритм**» – алгоритм, который на каждом шаге из нескольких возможных вариантов продолжения для достижения результата выбирает единственный.

Жадные алгоритмы понижают трудоемкость комбинаторного рекурсивного перебора сразу с экспоненциальной до линейной. В этом их основное преимущество и основная опасность: сущность жадности заключается в том, что последовательность локальных оптимальных решений должна дать в итоге оптимальный результат. На самом деле результат жадного алгоритма может быть разным:

- решение является верным – оптимальным;
- решение является субоптимальным, существуют лучшие варианты;
- решение не найдено, хотя оно существует.

Поэтому для жадных алгоритмов необходима дополнительная верификация (доказательство) их работоспособности либо обеспечение условий, при которых их логика становится очевидной с точки зрения доказательности. Иначе алгоритм может «проскочить» мимо существующего решения.

Пример. Задача синтаксического анализа состоит в построении дерева из набора правил формальной грамматики (элементарных деревьев) по входной строке символов (терминальных вершин). Она может быть решена в общем виде полным рекурсивным перебором всех правил для очередной вершины, что дает экспоненциальную трудоемкость. Методы трансляции на основе анализа множества правил грамматики позволяют получить дополнительные данные (селекторы). На их основе работает жадный алгоритм однозначного выбора. Если селекторы содержат непересекающиеся множества символов, то жадный алгоритм работоспособен. Иначе возможен неоднозначный выбор, и грамматика непригодна для использования в данном методе трансляции.

Иерархия, группировка, выбор системы координат

Идеи общего порядка также могут быть полезны. Одна из них – иерархия в организации данных – может давать очевидные преимущества:

- *локальность изменений* – большинство изменений размерности не выходит за границы элемента нижнего уровня;
- *отсутствие фрагментации* – исключение перераспределения памяти блоками переменной размерности за счет распределения блоками фиксированной;
- *эффект производительности* – разделение на части, независимая обработка и последующая интеграция. Например, в сортировке однократным слиянием независимая сортировка частей и их последующее слияние дают оценку трудоемкости $O(N^{3/2})$ вместо $O(N^2)$ исключительно за счет того, что манипулирование данными не производится «в общей куче».

Аналогично группировка множества мелких запросов и транзакций позволяет снизить сопутствующие ресурсные расходы

Эффекты производительности может давать простое изменение системы координат или порядка следования операций. Примером является простая смена порядка проверки условий, если одно из них обеспечивает большую частоту отсечения вариантов. Подробности и примеры можно посмотреть в [9].

3.2. В ООП-среде как рыба в воде

Далее речь пойдет не о принципах ООП как таковых, а о типовых инструментах объектно-ориентированной разработки, с которыми приходится сталкиваться при создании приложений. Приводимые ниже технологические решения и примеры реализованы на языке и в среде программирования Java.

Рефлексия

Основным требованием к коду является его *универсальность*, т. е. применимость к объектам различных классов. Это может быть реализовано, если классы имеют общего предка, присоединяют один и тот же интерфейс либо путем использования шаблонов.

Java позволяет использовать более общий подход, основанный на рефлексии. **Рефлексия** – возможность программы получить данные о структуре любого класса в виде объекта-описателя, и через это описание выполнять операции над классом, синтаксически не используя его имени: порождать объекты, вызывать методы и т. п. [84]. Проще говоря, рефлексия – *возможность программы анализировать собственную структуру и управлять ею*.

В Java рефлексия поддерживается естественным образом благодаря наличию *метауровня* описания загруженной программы:

- каждый класс хранится в памяти в виде динамического объекта, содержащего описание интерфейса класса и собственно его байт-код. Объекты-описатели классов имеют имя *Class*;
- методам класса соответствуют точки входа в таблице имен класса, содержащие начальные адреса методов в байт-коде;
- загрузка программы осуществляется динамически, откомпилированный двоичный файл класса с расширением *Class* загружается в соответствующий объект класса *Class*;
- все классы являются прямыми или косвенными наследниками класса *Object*, что обеспечивает виртуальной машине Java единый интерфейс доступа ко всем объектам;
- каждая ссылка в Java содержит две компоненты: адрес области памяти, где размещается объект, и ссылку на объект-описатель класса – *Class*, к которому принадлежит этот объект.

Напомним основные синтаксические элементы Java и методы классов *Class* и *Object*, имеющие отношение к рефлексии:

- метод *Class Object.getClass()* возвращает ссылку на объект-описатель класса;
- *<Имя класса>.class* является константной ссылкой на объект-описатель класса;
- Статический метод *Class Class.forName(String name)* возвращает ссылку на объект-описатель класса по его имени *name* либо *null*, если класс отсутствует;
- Метод *Object Class.newInstance()* является аналогом виртуального конструктора – создает объект класса по ссылке на объект-описатель;

- Метод *Field []Class.getDeclaredFields()* позволяет получить массив объектов *Field*, которые являются описателями собственных полей класса. В классе *Field* можно получить имя поля методом *getName,* тип его данных методом *getType,* существует набор методов для извлечения из него данных всех примитивных типов;
- Метод *Method []Class.getMethods()* возвращает массив объектов *Method*, которые являются описателями методов класса;
- Метод *Object Method.invoke(Object THIS, Object par1,…)* позволяет вызвать метод не по имени, а через его описатель, первым параметром является объект, для которого метод вызывается.

Полиморфизм внешний и внутренний. Абстрактные классы и интерфейсы

А под этой личиной был простой уголовник.
Ах, какой был мужчина – настоящий полковник.
Из песни А. Пугачевой «Настоящий полковник»

Полиморфизм – фундаментальный механизм ООП, лежащий в основе многих технологических решений. Термин «*полиморфизм*» наиболее адекватно переводится как *многоликость*, но имеется в виду не только возможность определить метод с общим именем в нескольких родственных классах, но и реализовать их вызов, который может меняться в зависимости от реального типа объекта, расположенного под ссылкой.

Основой полиморфизма являются операции *расширения* и *сужения*, образная сущность которых звучит как «быть» и «казаться». Под ссылкой на базовый класс может скрываться объект любого производного класса, в этом случае ссылка указывает на его внутреннюю базовую компоненту.

Преобразование типа ссылки от производного класса к базовому называется *расширением*. Свойства расширения:

- безопасное;
- может быть выполнено неявно;
- обозначает переход от конкретной сущности объекта к более общей, абстрактной.

Обратное преобразование называется *сужением*. Свойства сужения:

- всегда делается явно, с помощью операции явного приведения типа ссылки;
- может порождать исключения – динамическую ошибку приведения типа, когда приводимый тип не совпадает с реальным типом объекта под ссылкой. Метафора к этому случаю – эпиграф в заголовке;

- обозначает переход от общей сущности объекта к его конкретному виду;
- допустимость сужения может быть проверена с помощью операции *instanceof.*

В терминах Java полиморфизм – это автоматическое сужение при вызове метода по ссылке на базовый класс к производному, в котором метод переопределен в последний раз.

Полиморфизм – это динамический механизм, т. е. механизм *времени исполнения программы* (*runtime*), так как синтаксически вызов полиморфного метода выглядит всегда одинаково, а в зависимости от предыстории работы программы вызванный метод будет одноименным, но различным.

Полиморфизм может быть *внешним*, когда вызов метода по ссылке на объект базового класса выполняется вне классов, реализующих полиморфизм, и *внутренним*, когда вызов производится из самого базового класса.

Основные приемы, в которых работает полиморфизм:

- *внешний полиморфизм на основе общего базового класса* – группа родственных классов, имеющих общий базовый, использует его для создания общей для всех функциональности. Сама эта функциональность может быть реализована в базовом классе;
- *внешний полиморфизм на основе интерфейса* – аналогично предыдущему, только общая функциональность объявляется в интерфейсе, а реализуется в каждом классе, присоединившим интерфейс, по-своему. Реализующие интерфейс классы не имеют общего кода и данных, а только соглашение о функционале;
- *внутренний полиморфизм – отложенное программирование базового класса*. Если некоторый функционал базового класса предполагает различные варианты реализации, которые откладываются или изменяются в будущем, то следует определить метод, вызываемый базовым классом, который будет переопределен при наследовании. Таким образом, он будет вызываться в производном классе из кода базового. Если функциональность базового класса нужно сохранить, то первой строкой переопределенного метода является вызов этого же метода в базовом классе – *void F(){super.F();…}*;
- *внутренний полиморфизм – обработка событий в производном классе.* Имеется система, генерирующая асинхронные события, при наступлении которых вызываются известные методы базового класса. Сама система может быть частью базового класса, а может быть и вне его. Обработчики событий могут быть записаны как переопределенные методы в классе-наследнике;
- *асинхронный обратный вызов обработки событий.* Если в процессе исполнения кода инициализируется действие, которое должно возвращать

результат, но является продолжительным по времени, то обычно запускается поток, в котором оно выполняется. Но тогда результат этого действия не может быть обычным образом возвращен в основной поток выполнения кода оператором *return.* Для этих целей используется интерфейс обратного вызова, а исполняющий метод получает объект-адаптер, определенный в контексте основного класса с присоединенным интерфейсом (см. раздел 3.3).

Замечание по теме. Java является тотально полиморфной средой, все методы в ней полиморфны, хотя в большинстве ситуаций они работают как обычные, поскольку отсутствуют расширение и сужение. В то же время есть единственное *исключение из полиморфизма* – вызов метода базового класса в первой строке метода из переопределенного метода в производном классе. Если бы такого исключения из правил не было, то метод в базовом классе вообще невозможно было бы вызвать из производного, так как полиморфизм всегда бы выталкивал нас обратно – *void F(){super.F();…}*.

Вложенные и анонимные классы

Исполняемый программный код всегда имеет неявное окружение – **контекст**, в котором ему доступны классы, переменные и методы *по прямому имени* без каких-либо дополнительных операций. Контекст является понятием, обратным *области видимости*. **Область видимости** – область программного кода, где переменная может быть использована по прямому имени (рис. 3.1). Программный код имеет одновременно несколько контекстов (перечислены в порядке их перекрытия):

- глобальные данные и функции или текущее пространство имен (Си++);
- имена импортированных классов;
- статические методы и данные текущего класса;
- методы и данные текущего класса – текущий объект;
- незатененные данные и неперекрытые методы базового класса;
- для вложенных и анонимных классов – данные и методы родительского класса;
- для анонимных классов, создаваемых внутри методов, – локальные переменные и формальные параметры с модификатором *final*;
- формальные параметры текущего метода;
- локальные переменные текущего метода.

Программный код может использовать классы, объекты и методы, если они *достижимы из текущего контекста*. Поэтому добавление новых контекстов в программный код позволяет сделать его более компактным.

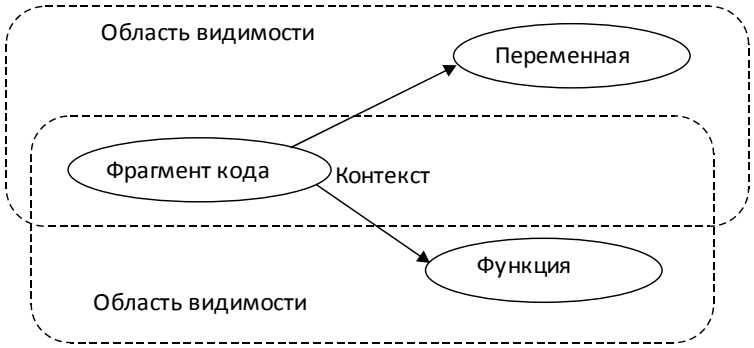


Рис. 3.1. Область видимости переменной и контекст кода

В Java имеется два способа, которые позволяют создавать классы – потомки, в которых доступен контекст исходного класса. Это вложенные и анонимные классы. Они имеют важные отличия от простых производных классов и друг от друга.

В производном классе объект базового класса является составной частью производного и создается одновременно с ним. Во вложенных и анонимных классах объект производного класса создается объектом *родительского класса* при исполнении кода этого класса, таких объектов может быть создано сколь угодно много и каждый из них «*видит*» в собственном контексте своего родителя. Отношение «родитель–потомок» является отношением вида *один ко многим* и реализуется не через вложенность, а по ссылке.

Вложенный класс – класс, синтаксически определенный обычным образом внутри тела основного класса.

Анонимный класс – вложенный класс, производный от абстрактного базового класса или интерфейса. Его тело в виде блока, заключенного в операторные скобки, записывается непосредственно вслед за операцией создания объекта *new*. Синтаксически это выглядит как *new A(){…}*. Такой класс является «одноразовым стаканчиком» – он создается для реализации уникального фрагмента кода, который по требованиям синтаксиса должен быть классом**.** Анонимный класс может быть определен внутри любого выражения, где допустим *new*. Особенно удобен анонимный класс для создания обработчиков событий на основе интерфейсов обратного вызова (см. раздел 3.3) и вспомогательных потоков, *видящих* данные родительского класса.

С точки зрения транслятора анонимный класс является обычным производным классом, который генерируется «на лету» и получает имя вида *<имя родительского класса>$<номер создаваемого анонимного класса в родителе>*, в чем можно удостовериться, посмотрев папку с классами сгенерированного кода.

Во вложенных и анонимных классах возникает затенение ссылки на текущий объект *this* – она теперь имеет отношение к объекту вложенного или анонимного класса. Объект класса-родителя доступен по имени вида *<имя родительского класса>.this*.

Замечание по теме. Объект вложенного класса видит не только родителя. Иногда ему необходимы данные локального контекста кода, в котором этот класс определяется – формальных параметров или локальных переменных. Для этих целей они помечаются как *final*. Тогда при создании объекта вложенного или анонимного класса *копии значений final-переменных* включаются в контекст объекта вложенного или анонимного класса.

Лямбда-выражения. Анонимные функции

Лямбда-выражения в Java заимствованы из языков и систем программирования, не являющихся тотально объектно-ориентированными. В них имеется возможность передавать в качестве формального параметра функцию с определенным прототипом. Фактическим параметром является имя функции, удовлетворяющей соответствующему прототипу, т. е. с такой же структурой заголовка. В языке Си это может быть сделано напрямую, с помощью типа данных – указатель на функцию [9].

Анонимные функции используют такую же схему определения, как анонимные классы. Вместо фактического параметра пишется тело функции, предваренное списком имен формальных параметров и некоторым синтаксическим элементом, обозначающим анонимную функцию. Важны только условные имена параметров, так как их типы определены в заголовке метода, использующего параметр-функцию.

Интерфейс с единственным методом, передаваемый в качестве параметра, рассматривается как параметр-функция. Фактический параметр – анонимный класс на основе этого интерфейса интерпретируется как анонимная функция, соответствующая этому методу, и имеет особый синтаксис (рис. 3.2): список имен параметров или пустые круглые скобки, символ «–>» и тело функции (метода) в виде отдельного оператора или блока.

Лямбда-выражения являются синонимом анонимных функций, передаваемых в качестве параметра.

Внутренний параллелизм. Потоки

Внутренний параллелизм, реализуемый потоками (*Thread*) – основа многих технологических решений, неотъемлемая часть языков программирования и средств разработки, обязательный сервис операционных систем.

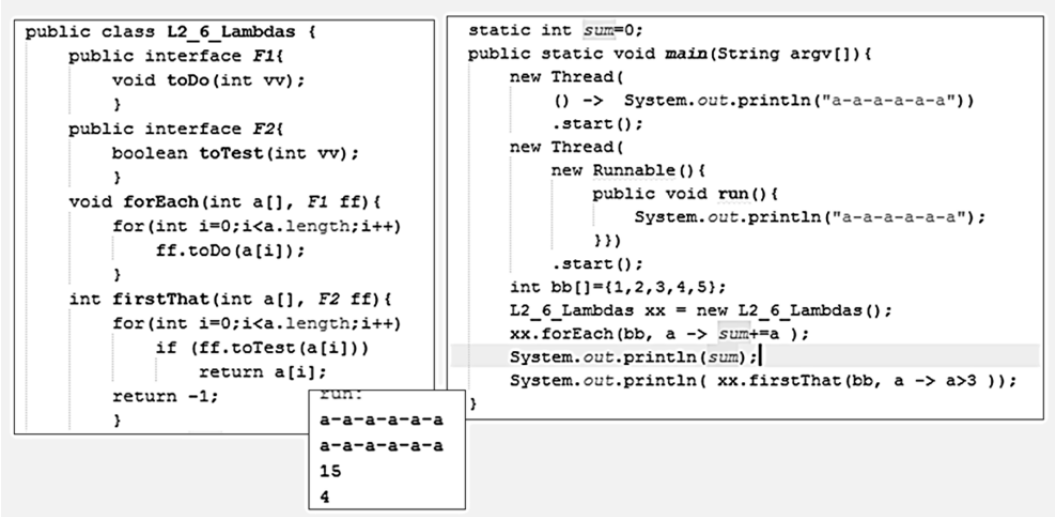


Рис. 3.2. Примеры лямбда-выражений

Поток (поток управления) – это последовательность действий программного кода, исполняемая параллельно с другими потоками в едином адресном пространстве процесса. **Процесс** – единица управления и планирования в ОС. Характеризуется адресным пространством, загруженным в него кодом и связанными с ним ресурсами. Для потоков и процессов обеспечивается параллелизм в процессе исполнения. Он может быть логическим (за счет переключения процессора и контекста процесса в ОС) и физическим (за счет исполнения на разных процессорах или ядрах с общей физическом памятью).

Основными характеристиками потока в некоторый момент времени являются:

- *состояние* процессора в точке исполнения программного кода;
- *локальный контекст* – стек потока, содержащий историю вызовов функций в потоке, их локальные данные и формальные (фактические) параметры вызовов.

Упрощенно поток можно образно определить *как функцию, исполняемую параллельно главной функции main*, а процесс – как экземпляр программы, загруженный в память.

Здесь отметим наиболее важные синтаксические и технологические принципы программирования потоков в Java:

- любой поток состоит из двух компонент – это компонента исполняемого кода потока и управляющий объект, через который JVM и программа воздействуют на поток;

- поскольку Java является системой тотального ООП и термин «функция» применим только к статическому методу, то программный код потока – это метод *run* в объекте с присоединенным интерфейсом *Runnable.* Интерфейс *Runnable* используется не только в потоках, но и во всех классах, где требуется передать исполняемый код. Например, это может быть таймер, запускающий по истечении установленного интервала времени некоторый код, передаваемый *Runnable*;
- управляющий объект – это объект класса *Thread*, через который программа и JVM осуществляют прямое управление процессом исполнения потока;
- связывание объекта управления *Thread* с управляющим кодом в методе *run* возможно выполнить двумя способами:
  - путем создания производного класса от *Thread* и переопределения в нем метода *run*;
  - путем передачи в конструкторе класса *Thread* ссылки на объект с присоединенным интерфейсом *Runnanble* и с переопределенным методом *run*;
- прямое управление потоком производится по ссылке на объект *Thread* через методы:
  - *start* – запуск потока на исполнение;
  - *interrupt* – деблокирование (пробуждение) потока, если он заблокирован (заснул) на методе *Thread.sleep*;
  - не рекомендуемые к использованию методы прямого управления потоком: *suspend* – приостановить, *resume* – возобновить, *stop* – остановить;
- статический метод *Thread.sleep* приводит к «засыпанию» потока, в коде которого исполняется этот метод. Пробуждение потока происходит по истечении интервала времени – параметра метода либо по выполнению метода *interrupt* над его управляющим элементом. При этом в потоке происходит исключение типа *InterruptedException*;
- статический метод *Thread.yield* принудительно переводит исполнение с текущего потока на какой-либо другой.

Проблема завершения / уничтожения потока

Не рекомендуется завершать потоки методом *stop.* Это связано с тем, что процедура завершения потока должна сопровождаться освобождением ресурсов, выполнением завершающих процедур, сохранением данных – *shutdown*. Поэтому завершение должно носить уведомительный характер: основной поток, желающий завершить приложение, устанавливает соответствующую

переменную-признак в потоковых объектах либо в себе самом. Потоки, обнаружив это, выполняют завершающие процедуры и выходят из метода *run*.

Если поток находится в состоянии засыпания на методе *Thread.sleep* или в состоянии ожидания на методе *wait*, то вывести его из этого состояния можно методом *interrupt*, повысив тем самым реактивность системы. Если поток блокирован на ожидании ввода из потока данных, то вывести его из этого состояния можно другим способом, закрыв поток данных.

В дополнение к описанному принципу Java-приложение не завершается, пока не завершатся все его потоки. Исключение сделано для фоновых (*daemon*) потоков. Объявить поток фоновым можно соответствующим методом на объекте управления *Thread.*

Синхронизация

Синхронизация потоков – установление ограничений на порядок исполнения фрагментов кода потоков при наличии между ними причинно-следственной связи или логической связи по используемым данным [20].

Первый вид синхронизации – **синхронизация разделения ресурса.** Несколько потоков могут одновременно использовать один и тот же ресурс. Таковым может быть все, что угодно: файл, структура данных, физическое устройство. При работе с ним выделяются **критические секции** или **неделимые операции,** исполнение которых предполагает **монопольное использование ресурса** и не может быть прервано другой такой же операцией.

Отсутствие необходимой синхронизации этого вида приводит к программным ошибкам, которые трудно обнаружить, так как они проявляются случайным образом при определенных временных совпадениях. Последствия этих ошибок могут быть различными. Рассмотрим примеры:

- вывод строки текста в log-файл. При переключении потоков возможно появление в файле строки вида *<начало строки потока 1><строка потока 2><окончание строки потока 1>*;
- создание записи в таблице с очередным значением идентификатора id при использовании последовательности операций: поиск записи с максимальным id, увеличение id на 1, добавление записи с id. При переключении потоков возможно появление двух записей с одинаковым значением id;
- операции получения и возвращения буфера в буферный пул на основе односвязного списка. При переключении потоков возможны потеря элемента списка, нарушение целостности структуры данных – один и тот же элемент может одновременно быть занятым и находиться в списке свободных.

Синтаксически синхронизация выполняется в виде блоков с синтаксисом *synchronized (объект) {критическая секция}* (рис. 3.3). Для каждого синхронизирующего объекта создается очередь потоков, все потоки, подходя к *любой секции synchronized, имеющей ссылку на этот объект, входят в секции строго по одному*. При этом имеется в виду не синтаксическая ссылка на объект, т. е. не его имя, а объект как таковой, доступный в разных фрагментах кода под разными выражениями. Например, оконный класс, производный от *JFrame*, использует для синхронизации сам объект оконного класса через *this*. Класс-поток, получив ссылку на объект оконного класса в конструкторе и запомнив ее в переменной *parent*, использует это имя для синхронизации.

Сам объект синхронизации может быть любым – от специально созданного объекта *Object* до произвольного объекта программы, доступного синхронизируемым потокам. Обычно для синхронизации к объектам-ресурсам используется сам этот объект.

Синхронизируемые методы вида *synchronized void F(){...}* на самом деле синхронизируют весь свой код к текущему объекту (рис. 3.3).

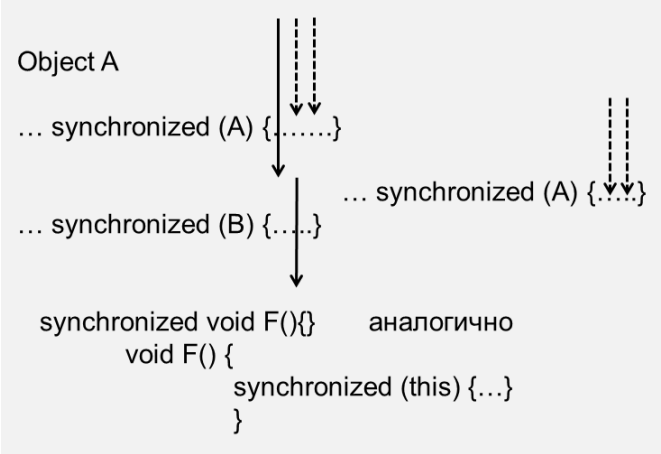


Рис. 3.3. Работа блока synchronized

Второй вид синхронизации – *причинно-следственная* – один или несколько потоков ожидают некоторого действия со стороны другого потока. Механизм синхронизации имеет название – *ожидание события на объекте.* Поток, ожидающий исполнения некоторого действия, выполняет на объекте синхронизации метод *wait* и блокируется. Другой поток при выполнении ожидаемого действия вызывает метод *notify* или *notifyAll* для того же самого объекта синхронизации – один или все ожидающие потоки деблокируются. Сами методы

wait и *notify* нужно заключать в блок *synchronized* к тому же самому объекту синхронизации. Объект синхронизации образно понимается как *экземпляр причинно-следственной связи.*

Замечание по теме

1. При выполнении *wait* перед блокировкой секция *synchronized* фактически закрывается, что естественно – ожидающий объект ресурс не занимает. После пробуждения повторная синхронизация происходит автоматически. Компилятор вставляет в код команду повторной синхронизации.

2. Описанные способы синхронизации делают реализацию некоторых примитивных взаимодействий не совсем очевидной. Например, необходима кнопка на форме, при нажатии на которую потоки приостанавливаются, а при повторном нажатии – возобновляются. Создается логическая переменная – состояние кнопки, значение которой меняется на противоположное при нажатии. Если использовать методы *suspend* / *resume*, то обработка события будет по принципу *один к одному.* Для *wait* / *notify* все выглядит несколько сложнее. При повторном нажатия кнопки нужно просто вызвать *notify* на объекте синхронизации, на котором выполняется ожидание – блокировка. При первом нажатии для приостановки потоков необходимо, чтобы они сами обнаружили изменение переменной состояния и выполнили метод *wait* – поток *нельзя усыпить принудительно.*

Синхронизация с потоком GUI

Все события графического интерфейса пользователя (GUI) в Java исполняются в отдельном потоке. Для desktop-приложений в Java потоком управляет класс *AWT-EventQueue.* Для приложений в *JavaFX* и *Android* есть свои классы. Все методы, исполняемые над объектами GUI, должны выполняться в этом потоке. Необходимые действия обеспечиваются двумя статическими методами:

- метод *invokeLater(Runnable)* планирует исполнение кода, задаваемого в методе *run* параметром *Runnable,* в потоке GUI синхронизировано с остальными событиями, но асинхронно по отношению к вызову. Сам метод не дожидается выполнения запланированного кода;
- метод *invokeAndWait(Runnable)* делает то же самое, но блокируется до завершения исполнения запланированного кода.

Так как обработка событий в потоке *GUI* происходит последовательно и они не прерывают друг друга, то указанные методы могут быть использованы для синхронизации остальных потоков между собой и с потоком *GUI.*

Символьные, двоичные и сериализуемые потоки ввода / вывода

В Java все источники внешних данных представлены классами, имеющими общие интерфейсы. Благодаря этому способы представления передаваемых данных в сетевых соединениях не отличаются, например, от аналогичного представления в файлах. Поэтому вначале обсудим общие положения, связанные с потоками.

Замечание по теме. Термин «поток» используется далее в двух контекстах – *поток данных ввода / вывода – Stream* и *поток управления – Thread.* Это создает определенную путаницу. Для потока управления есть прямой перевод – *нить,* но этот термин менее распространен. Поэтому *Thread* будем называть просто потоком, а *Stream* – потоком ввода / вывода или просто потоком, если смысл ясен из контекста.

Прежде всего потоки ввода / вывода являются последовательными. Это значит, что порядок чтения элементов данных в потоке идентичен порядку их записи и не может быть изменен. Наряду с ними в файловых потоках имеет место прямой (произвольный) доступ, основанный на *позиционировании* к любой единице данных в файле по ее физическому адресу. Естественно, в сетевых соединениях это не реализуемо.

Формы представления данных. Двоичные и символьные потоки

Передаваемые элементарные данные могут иметь *текстовую* (символьную) или *двоичную* форму представления. Поскольку все они имеют единую основу представления – двоичный разряд, байт, машинное слово, то на физическом уровне любой поток можно рассматривать как последовательность байтов, из которых составляются разные формы представления данных. На уровне этого перехода от физического представления есть важные моменты:

- машинное слово, которое складывается из байтов, может размещаться в памяти, а также передаваться в потоке, начиная как с младшего, так и со старшего байта:
  - **little endian (LE)** – последовательность размещения в памяти или передачи по каналу связи байтов машинного слова, начиная с младшего байта, является стандартным для архитектуры *Intel x86*. Например, целое *int a = 1256* во внутреннем представлении в виде 32-разрядного машинного слова выглядит как 0x000004E8, а хранится и передается побайтно как E8 04 00 00;

- **big endian (BE)** – последовательность размещения в памяти или передачи по каналу связи байтов машинного слова, начиная со старшего байта. Этот порядок байтов используется процессорами *IBM 360/370/390, Motorola 68000, SPARC*, а также в двоичных потоках ввода / вывода в Java;

• при передаче данных по последовательным каналам связи байт может разворачиваться в битовую последовательность, начиная либо со старшего, либо с младшего разряда. Это касается в первую очередь совместимости аппаратных средств.

Двоичная (внутренняя) форма представления данных есть не что иное, как представление этих данных в оперативной памяти компьютера в тех форматах, в каких они обрабатываются процессором. В языках программирования этим формам соответствуют *базовые* (примитивные) типы данных, в Java это – *byte, char, short, int, long, float, double, boolean*. Таким образом, в двоичном потоке примитивные типы данных пишутся «байт в байт». Термин «**образ памяти**» также используется для обозначения способа хранения любой области памяти в двоичном файле по принципу прямого отображения «байт в байт».

Символьная (внешняя) форма представления данных базируется на понятиях «символ текста» и «строка». Для символа определяющим понятием является *код символа* – уникальное значение машинного слова как беззнаковой переменной, которое с ним соотносится, а также *способ кодирования.* Исторически сложилось несколько способов кодирования символов:

• однобайтная кодировка по принципу *символ-байт,* сложившаяся в 1970-е гг., ограничивает множество представляемых символов значением $2^8 = 256$. Поскольку эта величина очень мала, для представления символов различных алфавитов вводятся *кодовые таблицы* – таблицы соответствия кода символа и его написания. Вся проблема заключается в том, что кодовая таблица, а следовательно, и вид символов в потоке никоим образом стандартно не оговариваются. Они могут устанавливаться программами по умолчанию, взаимной договоренности, наименование кодовой таблицы может содержаться в начальном теге передаваемого текста. Первая половина всех кодовых таблиц, содержащая наиболее популярные символы, цифры и латинские буквы, одинакова для всех кодовых таблиц;

• двухбайтная кодировка **Unicode**, сложившаяся в 1990-е гг., является универсальной системой кодирования для всех алфавитов, первые 128 символов Unicode совпадают с аналогичными в байтной кодировке;

• смешанная кодировка **UTF-8 (UTF-16)** использует односимвольную кодировку для первых 128 символов Unicode (семь разрядов), двухбайтную – для

последующих 2048 символов (11 разрядов) и т. д. Поскольку размер символа переменный, используется *саморазворачивающийся битовый формат*.

Существующее положение таково:

- под термином «**текстовый файл**» по умолчанию понимается файл символов в байтной кодировке, большинство файлов исходных текстов программ, настроечных, командных файлов и являются текстовыми, с этим представлением работают стандартные библиотеки ввода / вывода языков программирования;
- для Java внутреннее представление символов – *Unicode*, этому соответствуют тип данных *char* и строковый тип *String*. Поэтому текстовые потоки в Java выполняют перекодировку в соответствии с кодовой таблицей, имя которой задается при конструировании объекта. При этом тип *char* фактически идентичен типу *short*, с ним можно работать как с коротким целым. Кроме того, строковый тип имеет методы извлечения и загрузки из массива символов с байтной кодировкой, т. е. *byte[]*. Естественно, что при этом также производится перекодировка;
- в двоичных потоках также можно использовать символьные данные, запись в поток переменной типа *char* соответствует записи символа *Unicode*, а строка в байтовой кодировке может быть записана массивом *byte[]*;
- в двоичных потоках для передачи строки в кодировке UTF используются методы *readUTF* / *writeUTF*, но они уже используют формат передачи: количество байт в последовательности в виде *short*-переменной, за которой идет сама последовательность символов в UTF-кодировке.

Форматы передаваемых данных

При передаче фиксированной последовательности данных известных типов нет необходимости в каких-либо дополнительных данных о порядке их следования: *высокие договаривающиеся стороны – методы ввода и вывода* просто соблюдают этот порядок. Если же последовательность варьируется, то для ее описания необходимо задать формат.

Формат – описание порядка следования данных в последовательном потоке. В формате очередной элемент является либо элементом данных, либо управляющим элементом, содержащим параметры следующих за ним данных. Основными управляющими элементами формата являются:

- счетчик количества последующих элементов;
- физическая длина последующего поля данных в байтах;
- символ или значение – ограничитель последовательности данных;
- тег, идентификатор – обозначение вида последующего формата или его свойства;

- форматная строка – строка, содержащая специальные символы, на место которых подставляются значения в порядке их следования за самой строкой.

Как правило, формат является *саморазворачивающимся*, т. е. алгоритм его чтения не требует дополнительных данных и извлекает все необходимые параметры из самой последовательности – принцип «скатерти-самобранки».

Формат может быть *иерархическим* или даже *рекурсивным*, например, последовательность данных, предваряемая счетчиком, может, в свою очередь, иметь собственный формат. Рекурсивно сохраняются древовидные структуры.

Одним из вариантов реализации формата является его представление в структуре кода *один к одному*:

- последовательность операций записи в поток соответствует фиксированному порядку их следования в потоке;
- операция записи данных в цикле соответствует формату, в котором последовательность переменной длины либо предваряется счетчиком, либо заканчивается значением-ограничителем;
- перед конструкцией выбора, в которой в каждой ветке записывается свой формат, должна быть произведена запись тега, указывающая на тип формата;
- вызов метода, который записывает данные в каком-либо формате, соответствует вложенности данных одного формата в другой.

Сериализация

В технологии ООП под **сериализацией** понимается передача в потоке данных значений полей объекта и, возможно, его типа или описания. При чтении сериализованного потока происходит создание объектов тех классов, данные которых находятся в последовательности, и загрузка их значений. Это позволяет имитировать сохранение и восстановление объектов в файлах, а также передачу объектов от одного потока или процесса другому, в том числе и через сеть.

Существует множество разных способов сериализации:

- ручная сериализация собственных данных классами программы. Каждый класс сохраняет свои данные в поток в своем собственном формате, который зашит в коде этого класса;
- универсальная сериализация объектов произвольных классов в двоичный поток средствами JVM – Java-сериализация;
- универсальная сериализация объектов произвольных классов в тестовый поток в формате XML / JSON с использованием сторонних библиотек.

Общие принципы сериализации, которые следуют из положений ООП:

- каждый класс сам сериализует собственные данные при помощи пары методов с общепринятым именем, например *loadText* / *saveText*, для сериализации в текстовый поток;
- если класс является производным, то он также вызывает одноименные методы в базовом классе *super.loadText* / *super.saveText*;
- если класс содержит массивы, векторы или списки, то он в цикле сериализует в поток объекты массива, предваряя их счетчиком, либо вызывает аналогичный метод для класса вектора или списка;
- если класс содержит ссылку на объект другого класса, то он вызывает в объекте одноименный метод сериализации по ссылке. Если ссылка может быть нулевой, то в поток пишется логическая переменная со значением *false* при отсутствии объекта. При наличии объекта пишется *true,* а затем вызывается метод сериализации в самом объекте;
- при чтении данных из потока сериализации класс должен создавать объекты и структуры данных, а затем вызывать для них соответствующие методы чтения;
- ссылка на поток сериализации передается по цепочке вызовов метода во всех перечисленных случаях;
- если сериализация является универсальной, то кроме собственно данных объекта в поток необходимо передавать метаданные о структуре объекта или имя его класса. Это может быть сделано однократно в начале потока (Java-сериализация) либо в процессе сериализации каждого объекта путем записи пары *имя компоненты – значение* (JSON-, XML-сериализация). Перед сериализацией самого объекта в поток может записываться имя его класса или идентификатор класса;
- если при сериализации объектов по ссылкам возможно появление ссылок на повторно сериализуемые объекты, то процедура сериализации усложняется. При сериализации необходимо помечать и нумеровать сериализованные объекты, а также передавать эти номера в поток. Если встречается ссылка на отмеченный объект, то в поток передается только его номер. Также каждую ссылку нужно предварять записью в поток логической переменной, обозначающей факт первоначальной или повторной сериализации;
- при универсальной сериализации необходимо использовать средства рефлексии, для того чтобы по имени класса, получаемого из потока, создавать объект этого класса.

Классификация потоков данных в Java

В Java классы для работы с потоками ввода / вывода находятся в пакете *java.io*:

- группы классов строятся на основе общих базовых классов или интерфейсов, что позволяет создать разнообразие потоков с единым функционалом;
- объекты классов связаны по принципу конвейера, текущий функционал реализуется через объект, который ссылается на объект другого класса для использования его функционала.

Также имеются классы, предназначенные для работы с файлами локальной файловой системы, например *File* – для операций над файлами и каталогами или *RandomAccessFile* – для двоичных файлов произвольного доступа. Можно также открывать потоки ввода / вывода на файлах, используя объект класса *File* в качестве параметра.

Для работы с потоками данных имеются три группы классов, образующих конвейер (рис. 3.4).

Классы – источники данных основаны на абстрактных классах *InputStream* и *OutputStream*. Они реализуют модель физического потока байтов. Классы позволяют читать / записывать отдельные байты и их массивы, проверять количество готовых байтов в потоке. На основе этого класса создаются классы для следующих источников данных:

- *FileOutputStream, FileInputStream* – файлы в локальной файловой системе;
- *ByteArrayInputStream, ByteArrayOutputStream* – массив байтов в памяти. Размерность массива увеличивается в процессе перераспределения памяти при записи. Используется для буферизации данных перед их передачей в другие потоки. После записи в поток *ByteArrayOutputStream* результирующий массив может быть извлечен из объекта методом *toByteArray*, при создании объекта *ByteArrayInputStream* ему передается массив байтов с данными;
- *PipedInputStream, PipedOutputStream* – поток, который передает данные через канал (*Pipe*), имеющий внутреннюю буферизацию. При конструировании объекта-потока ввода / вывода он получает в качестве параметра объект противоположного класса, они поддерживают внутренний буфер, через который обмениваются данными. Это позволяет развести по времени процессы записи и чтения;
- *SequenceInputStream* – поток, соединяющий в себе последовательно данные из нескольких потоков;
- сетевое соединение на сокете, реализуемое классом *Socket,* может создавать потоки – производные классы от *InputStream* и *OutputStream*

с помощью методов *getInputStream* и *getOutputStream*. Они имеют в качестве источника данных породившее их сетевое соединение. Само сетевое соединение поддерживает надежный двунаправленный поток данных между сокетами с использованием циклической буферизации, поэтому на одном сокете могут быть получены одновременно потоки для ввода и вывода.

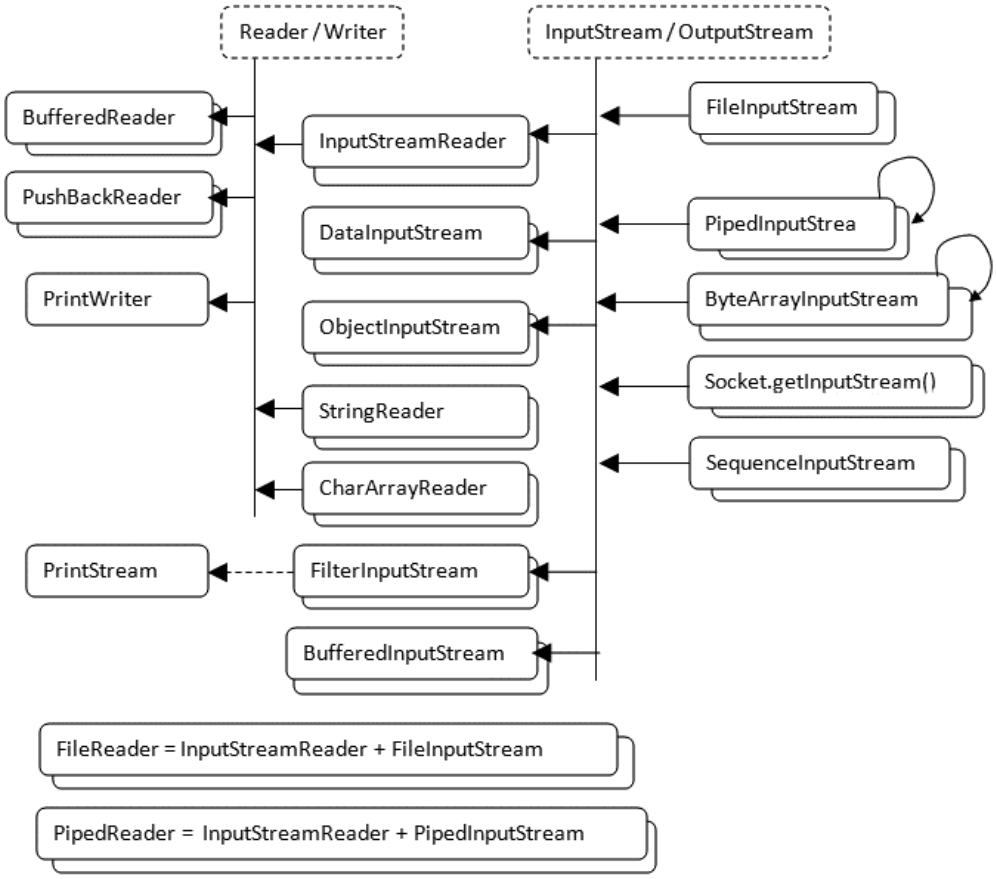


Рис. 3.4. Классы ввода / вывода в Java

Классы следующего уровня создают на основе физического потока различные **формы представления** данных в потоке:

- *DataInputStream, DataOutputStream* – потоки двоичных данных;
- *InputStreamReader, OutputStreamWriter* – потоки символьных данных в *Unicode* с перекодировкой из байтного представления, получаемого из потоков-источников;

- *ObjectInputStream, ObjectOutputStream* – потоки для универсальной двоичной сериализации средствами JVM.

Последний уровень выполняет вспомогательные функции по буферизации данных и их форматированию:

- *BufferedReader, BufferedWriter* – построчное чтение и запись в символьном потоке на основе буферизации;
- *PushbackReader* – символьный поток с возможностью возврата символов и повторного перечитывания;
- *PrintStream* – символьный поток с возможностью форматного вывода данных примитивных типов, аналогично функции *printf* в языке Си.

Кроме основного конвейера «*источник–представление данных–сервис*» существует еще ряд классов и интерфейсов:

- символьные потоки также имеют свой интерфейс – абстрактный класс *Reader* / *Writer*, с которым связаны свои внутренние потоки – источники данных:
- *StringReader, StringWriter* – внутренний строковый буфер (*StringBuffer*);
- *CharArrayReader, CharArrayWriter* – внутренний массив символов *char[]*;
- средства, аналогичные позиционированию в файлах произвольного доступа, предлагает *FilterInputStream, FilterOutputStream*. Они позволяют отмечать текущую позицию потока, а затем возвращаться к ней для повторного чтения или записи. Также возможен и пропуск заданного количества байтов. Аналогичные действия в совокупности с буферизацией имеются в *BufferedInputStream, BufferedOutputStream*;
- имеются классы, объединяющие уже существующие, например *FileReader – InputStreamReader* и *FileInputStream*, *PipedReader – InputStreamReader* и *PipedInputStream*.

В заключение рассмотрим подробности некоторых способов сериализации.

Универсальная сериализация с помощью JVM. Поток сериализации является одновременно и двоичным потоком. Для классов, объекты которых подлежат сериализации, необходимо подключить маркерный интерфейс *Serializable*. Ссылки на объекты, которые не нужно сериализовать, помечаются служебным словом *transient.* Это ссылки на объекты окружения, которые при загрузке надо также восстанавливать вручную.

Для сериализации объекта в потоке существует единственный метод *writeObject,* которому может быть передан объект любого класса и присоединенным интерфейсом *Serializable.* При десериализации объекта метод *readObject* возвращает ссылку на объект типа *Object*, которая должна быть сужена до ожидаемого класса.

Универсальная XML-сериализация с помощью библиотеки XStream. Библиотека *XStream* [86] позволяет сериализовать объекты любого класса без использования методов установки / получения значений полей *set* и *get*. С помощью рефлексии класс сам добирается до полей объектов. Формат сериализации достаточно прост:

- главный тег сериализуемого объекта соответствует имени класса, т. е. *class AAA{}* сериализуется в *<AAA>…</AAA>*;
- имена тегов элементов данных класса совпадают с их именами в классе, т. е. *Color cl* сериализуется в *<cl>… теги элементов… </cl>*;
- если сериализуется вектор или массив, то создается внешняя пара тегов по имени вектора или массива, а элементам массива (вектора) соответствуют теги с именем типа массива, т. е. *Color cc[]* сериализуется в *<cc><color>…</color> <color>…</color> …</cc>*.

Для работы достаточно создать объект класса *Xstream*. Но формат лучше всего настроить хотя бы для уменьшения размера текстовых данных:

- метод *alias* устанавливает синоним для имени класса, если он отсутствует, то библиотека использует полное имя класса;
- метод *aliasAttribute* устанавливает синоним для имени элемента данных в классе;
- метод *useAttributeFor* включает имя элемента данных и его значение в открывающийся тег класса.

Для сериализации и десериализации объектов используются методы *toXML* и *fromXML*, при этом в качестве источника / приемника данных может выступать либо текстовый поток типа *OutputStreamReader*, либо строковый объект.

JSON-сериализация с помощью библиотеки google-gson. Библиотека [87] позволяет сериализовать объекты любого класса без использования методов установки / получения значений полей. С помощью рефлексии класс сам добирается до полей данных объектов.

Для работы достаточно создать объект класса *Gson*. Однако алгоритм сериализации зависит от особенностей формата JSON. Так как в нем отсутствуют имена тегов, соответствующие именам классов, а есть только связки «имя–значение», то возникают следующие ограничения:

- при десериализации объекта методу *fromJson* необходимо передавать описатель того класса, объект которого загружается, например, в виде константной ссылки вида *Zwezda.class*;

- ссылки на другие объекты, а также массивы ссылок не должны быть полиморфными, если ссылка соответствует базовому классу, то она может ссылаться только на этот класс, а не на производный. При отсутствии имени реального класса в формате десериализатор создает объект по синтаксическому имени, т. е. объект базового класса, что является неустранимой ошибкой.

В связи с упомянутыми ограничениями сериализация в формате JSON производится аналогично ручной, т. е. в поток пишутся счетчики объектов и имена классов, а сами объекты сериализуются классом *Gson*, причем каждый объект в отдельную строку.

Исключения и их обработка

Исключения связаны с обработкой ошибок и их механизм приблизительно одинаков во всех языках программирования. Классическая схема обработки ошибок заключается в возвращении функцией недопустимого значения, которое анализируется в точке вызова. Исключение позволяет синтаксически отделить код для нормального поведения программы и код обработки ошибок, выведя последний в отдельный поток управления (рис. 3.5).

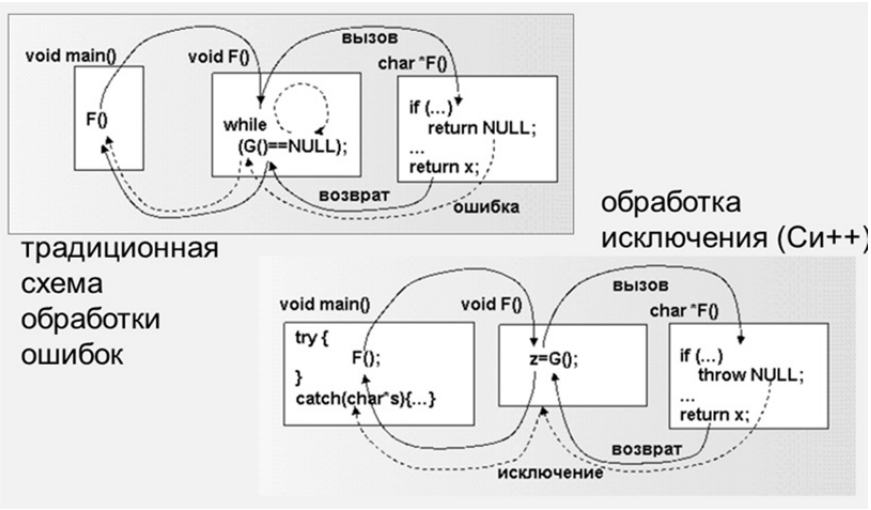


Рис. 3.5. Обработка ошибок, возвращаемых функцией, и исключения

Исключение – альтернативный поток управления при обнаружении ошибки в программе. Его основные принципы:

- альтернативный поток управления – при генерации исключения выполняется цепочка операторов *return,* происходит *разматывание* стека с разруше-

нием локальной среды исполнения методов до обнаружения первого *catch*, способного обработать это исключение;

- источником исключения является оператор *throw*;
- данные об исключении передаются в объекте класса *Exception* или его наследниках. Функционал объекта-исключения описан интерфейсом *Throwable*;
- исключения, генерируемые внутри защищенной секции *try {} catch…catch…finally*, пропускаются через обработчики *catch*;
- *catch(Exception)* – анонимная inline-функция с параметром – типом исключения. Обрабатывает исключения указанного класса и его производных.

Для контроля за полнотой обработки исключений вводятся синтаксические правила делегирования. Когда в некотором коде присутствует источник исключения *throw* класса *XException* или производных ему классов либо вызываемые методы делегируют ему такое исключение, то компилятором проверяется следующая последовательность условий:

- если источник заключен в секцию *try-catch(XException ee){}*, т. е. существует соответствующий обработчик, то исключение обрабатывается;
- иначе исключение должно быть делегировано вызывающему этот метод коду, для чего в заголовке метода прописывается *void F() throws XException {}*. В этом случае метод является потенциальным источником исключений;
- неисполнение предыдущих условий рассматривается как синтаксическая ошибка.

Делегирование исключений не распространяется на исключения, вызванные программными ошибками и сбоями самой JVM – их можно перехватывать по желанию:

- *RunTimeException* – программные ошибки, классы, производные от Exception: *NullPointerException*, *ArrayOfBoundsException*;
- *Error* – сбои JVM: *StackOverFlowError*, *OutOfMemoryError*.

Блок *finally* содержит код, который гарантированно выполняется при наличии или отсутствии исключений во всех вариантах их обработки секцией *try-catch*. Он выполняется при выходе из секции и не меняет ее результата – наличия или отсутствия исключения.

Механизм обработки исключений рассчитан на обработку одиночных сбоев. Если в блоке *finally* или при обработке исключения в *catch* возникает новое исключение, то оно перекрывает старое, и метод становится источником этого исключения. Обработка таких ситуаций не производится корректно механизмом исключений.

Основные архитектурные претензии к исключению состоят в том, что оно является достаточно сильным средством, разрушающим поток управления.

Кроме того, многообразие типов исключений существенно усложняет структуру кода. Возможен вариант замены исключений на другие технологические приемы и паттерны (см. раздел 3.4):

- использование интерфейса обратного вызова с методами для нормального, аварийного завершения и различных его вариаций;
- определение валидного и нескольких недопустимых состояний возвращаемого объекта. Например, для GPS-координат возможны следующие состояния: *координаты отсутствуют* и *координаты устарели*. Принятие решения, как поступать с таким объектом, делегируется вызывающему методу;
- использование шаблона-контейнера *Option*, вводящего для класса-параметра обобщенное значение валидности и имеющего набор методов от возвращения ссылки с возможным значением *null* до генерации исключения;
- создание универсального класса исключений для приложения, которое классифицирует и унифицирует различные типы исключений, а также содержит средства их обработки. В этом случае перехватываются исключения всех типов при помощи *catch(Throwable)* и приводятся к единому виду.

3.3. Шаблоны проектирования и конструирования

Пить Анна Петровна не хотела, но, повинуясь моему нескрываемому желанию ее напоить, чтобы она развязала мне руки для шаблонных мужских поступков, она велела заморозить две бутылки шампанского.

А.С. Бухов. Шаблонный мужчина

Марксизм – не догма, а руководство к действию.

В.И. Ленин

Шаблон проектирования (паттерн) – общепринятое решение часто встречающейся проблемы проектирования, повторимая архитектурная конструкция.

Как всегда сущность заключается в деталях:

- уровень решения в шаблоне – решение может быть в виде определенной программной конструкции из классов и объектов – шаблон проектирования либо состоять из более крупных аппаратно-программных компонент – архитектурный шаблон;

- шаблон проектирования может быть проиллюстрирован UML-диаграммой классов (объектов);
- шаблон не следует понимать буквально как некоторую заготовку, воспроизводящую предложенную в образце структуру. В основе шаблона лежит идея, которая может по-разному преломляться для разных задач. И даже сама идея тоже может быть подвергнута критике и ревизии;
- несмотря на разнообразие шаблонов проектирования, в их основе лежит один и тот же инструментарий ООП: классы, объекты, интерфейсы, наследование, полиморфизм. Именно поэтому при поверхностном взгляде шаблоны смотрятся довольно однообразно: сущность шаблона заключена в идее и ее реализации, скрытой в поведении шаблона.

Базовые шаблоны

Делегирование

Некоторая функциональность основного класса выносится в отдельный класс, через ссылку на объект этого класса осуществляется *делегирование* функционала. Это технологическое решение, позволяющее разгрузить основной класс и сделать программу более модульной. В делегируемом коде могут остаться обращения к данным ведущего класса, тогда делегатам придется передавать обратную ссылку на ведущий класс и в нем создавать методы для доступа к нужным данным.

Абстрактный суперкласс

Абстрактный класс создает интерфейс для группы производных классов как основа для многообразия последующих реализаций и содержит общий функционал и структуры данных. Общий функционал использует полиморфные вызовы интерфейсных методов для *отложенного программирования* функционала в производном классе.

Маркерный интерфейс

Интерфейс, не содержащий методов, используется для обозначения наличия у класса какого-либо свойства, например, *Serializable* – стандартная Java-сериализация. Наличие интерфейса у класса может быть проверено операцией *instanceof*.

Заместитель (Proxy)

Заместитель имеет тот же интерфейс, что и замещаемый класс, а также ссылку или возможность передачи сообщений замещаемому объекту (рис. 3.6).

Рис. 3.6. Шаблон «заместитель» (proxy)

Заместитель изменяет поведение замещаемого объекта в различных аспектах:

- безопасность, ограничение доступа, буферизация и фильтрация данных;
- удаленный доступ по сети. Класс-заместитель передает сообщение серверу, который создает замещаемый объект и выполняет с ним указанные действия;
- метод замещаемого объекта выполняется асинхронно в потоке, объект-заместитель завершает метод, не дожидаясь его исполнения в заместителе;
- шаблон *virtual proxy*. Заместитель не создает замещаемого объекта до тех пор, пока в нем не возникнет реальная необходимость – *ленивая инициализация*.

Принципиальным для *Proxy* является идентичность интерфейса оригинального класса и его заместителя, в связи с чем заместитель и может выступать в роли оригинала везде, где используется оригинал.

Обратный вызов (Callback)

Шаблон обратного вызова является альтернативой такой простой базовой сущности, как вызов метода в объекте. У обычного вызова есть несколько особенностей, которые не всегда приемлемы или удобны в использовании:

- вызов является синхронным и выполняется в едином потоке управления;
- результат вне зависимости от варианта завершения вызова передается через один и тот же тип результата либо фиксируется в формальных параметрах вызова. Например, при отрицательном результате поиска вместо ссылки на объект возвращается *null*;
- сбой или ошибка при выполнении метода сопровождается исключением, которое требует отдельного кода обработки.

Такая жесткая схема бывает неудобна по следующим причинам:

- момент появления результата вызова может быть асинхронным: метод запускает поток и завершается, фактические результаты вызова являются результатом работы потока, т. е. имеет место *внутренний параллелизм метода*;
- метод возвращает результат в виде множества объектов *или параметров*. Можно, конечно, создавать специальный класс, интегрирующий все

результаты либо использовать несколько объектов-параметров, что не совсем удобно;

- исключения, порождаемые методом, можно обрабатывать внутри метода, создавая дополнительный вариант результата – ошибки метода;
- метод возвращает результат в виде множества однотипных объектов, в самом методе происходит их накопление, а затем они возвращаются как одно целое, что приводит к большим промежуточным издержкам памяти;
- метод имеет множество вариантов завершения, которые вызывающий код должен обрабатывать.

Во всех случаях прямой вызов метода и обработку результата в контексте вызывающего класса можно реализовать с использованием одного или нескольких методов **обратного вызова,** реализуемых через интерфейс.

Перечислим технологические решения, для которых может использоваться обратный вызов.

1. При обращении к сети непосредственно из GUI могут иметь место существенные задержки, блокирующие клиента, которые внешне воспринимаются как временное зависание программы. При использовании обратного вызова метод, работающий с сетью, запускает поток, в котором выполняется необходимое взаимодействие и по окончании которого производится асинхронный обратный вызов.

Необходима как минимум синхронизация обратного вызова, выполняемого в отдельном потоке, с основным потоком управления. Если основной поток является потоком GUI, то эта синхронизация выполняется стандартными статическими методами. Кроме того, необходимо заблокировать возможность повторного прямого вызова этого же метода, пока не завершился текущий, а также разнести данные, с которыми работает основной поток и поток в вызове, либо синхронизировать работу с ними.

2. Вызываемый метод обращается к БД, от которой в цикле получает выбранные записи, которые должны обрабатываться вызывающим объектом. В этом случае *каждую запись можно возвращать в виде обратного вызова*, а по завершении цикла выполнять еще один обратный вызов специального метода завершения в том же интерфейсе, если обратные вызовы являются асинхронными.

3. При вызове метода возможны различные варианты ответа, которые можно реализовать в виде группы обратных вызовов в общем интерфейсе. Тогда вызывающему коду не потребуется их повторная селекция.

4. При использовании обратного вызова обработка исключений может быть реализована в два этапа – в вызываемом методе при помощи обычного перехвата исключения, а затем в дополнительном методе обратного вызова.

5. Если метод переопределен в группе классов, и в некоторых из них требуется использование дополнительных параметров, то эти параметры можно не передавать в общем списке, а запрашивать в нужный момент методами обратного вызова из контекста вызывающего объекта. Обратный интерфейс с методами запроса параметров можно сделать производным от базового интерфейса. Классы, принимающие параметры, будут выполнять у себя явное сужение до требуемого интерфейса, а уже затем забирать через него нужные параметры.

Компоненты шаблона изображены на рис. 3.7, их поведение – на рис. 3.8. Класс-клиент создает объект-адаптер с интерфейсом обратного вызова и передает его классу-сервису до начала взаимодействия. Объект-адаптер работает в контексте класса-клиента, т. е. ему доступны его данные и методы, при необходимости он должен выполняться в том же потоке, что и класс-клиент. При исполнении метода в классе-сервисе и возникновении событий, требующих участия клиента, через интерфейс обратного вызова вызываются методы в объекте-адаптере.

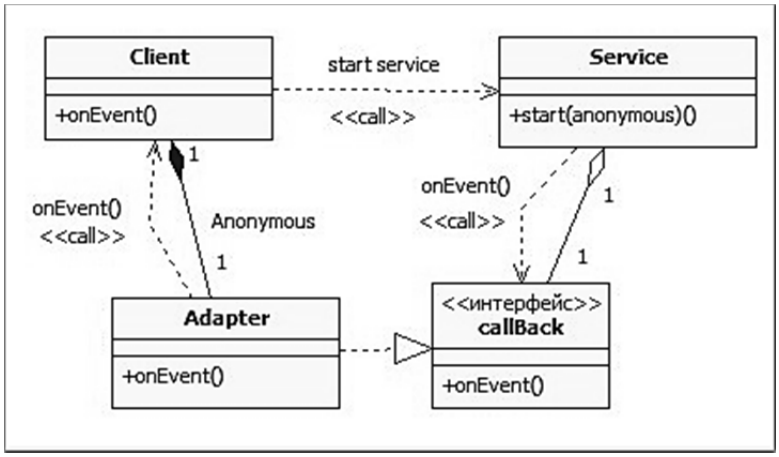


Рис. 3.7. Диаграмма классов обратного вызова

С помощью UML-диаграммы взаимодействия (рис. 3.9) проиллюстрируем, как выглядит возврат последовательности сгенерированных данных через интерфейс обратного вызова:

- создается интерфейс, в котором объявляются методы *onBeginSequence*, *onEndSequence* и *onDataElement* с параметром – очередным элементом данных;
- объект класса, в котором реализуется метод генерации, получает в качестве параметра ссылку на объект-слушатель с присоединенным интерфейсом обратного вызова;

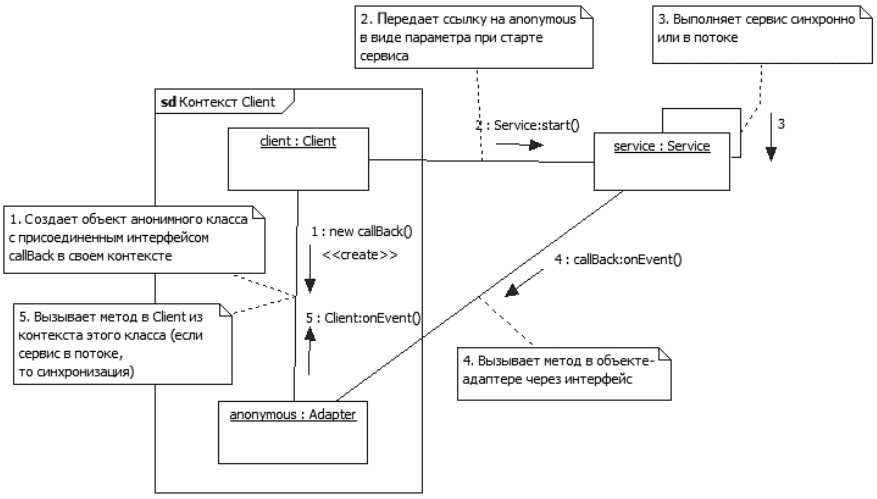


Рис. 3.8. Коммуникационная диаграмма обратного вызова

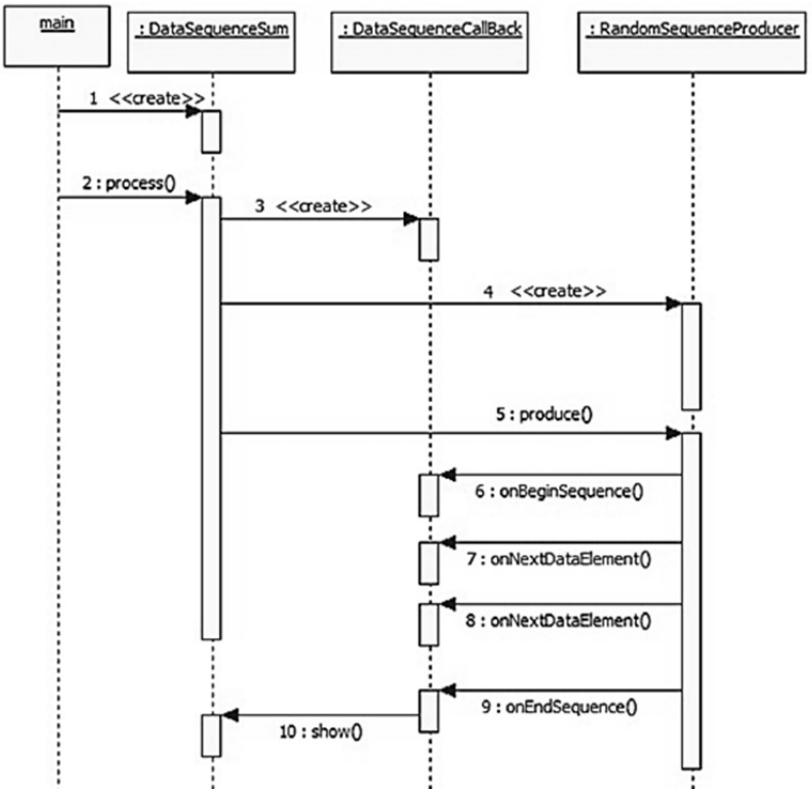


Рис. 3.9. Диаграмма взаимодействий для группы обратных вызовов

• класс-генератор последовательности вызывает в обратном интерфейсе методы *onBeginSequence*, *onEndSequence* в начале и конце последовательности, а также метод *onDataElement* для каждого сгенерированного объекта данных.

Порождающие шаблоны

Фабрика (Factory)

Универсальность программы определяется в том числе и возможностью создавать объекты, принадлежащие к одной абстракции, но чтобы конкретный класс объекта определялся динамически, т. е. в зависимости от обстоятельств. Для этой цели и существуют классы-фабрики.

Метод «фабрика» (Factory Method)

Существует группа родственных классов на основе абстрактного базового класса или интерфейса. Метод «фабрика» создает и возвращает объект одного из классов, руководствуясь своими параметрами, например по типу файла, передаваемого в имени, либо имеет набор статических методов, возвращающих объекты разных классов.

Абстрактная фабрика (Abstract Factory)

Иногда требуется создать не отдельный объект, а группу объектов, каждый из которых является конкретизацией отдельной абстракции. Например, при создании отдельного стиля оформления оконного приложения классы окна и кнопки базируются на абстракциях *IWindow* и *IButton* (рис. 3.10). Интерфейс фабрики *AbstractFactory* содержит методы для порождения объектов класса окна и кнопки. Конкретные фабрики для разных стилей оконного приложения создают каждая свой набор объектов, соответствующих стилю.

Иногда фабрика обеспечивает возможность создания объектов из некоторого разнообразия, на данный момент «зашитого» в программе. В этом случае классы, порождаемые фабрикой, должны быть потомками класса, указанного при создании фабрики. Фабрика при помощи средств рефлексии может обнаружить эти классы и создать на их основе вектор объектов-прототипов.

Строитель (Builder)

Шаблон аналогичен фабрике, только создает не отдельный класс, а систему из объекта основного класса и связанных с ним объектов других классов. Кроме того, в абстрактном базовом классе имеется статический

метод, который создает объект производного класса, руководствуясь параметрами вызова.

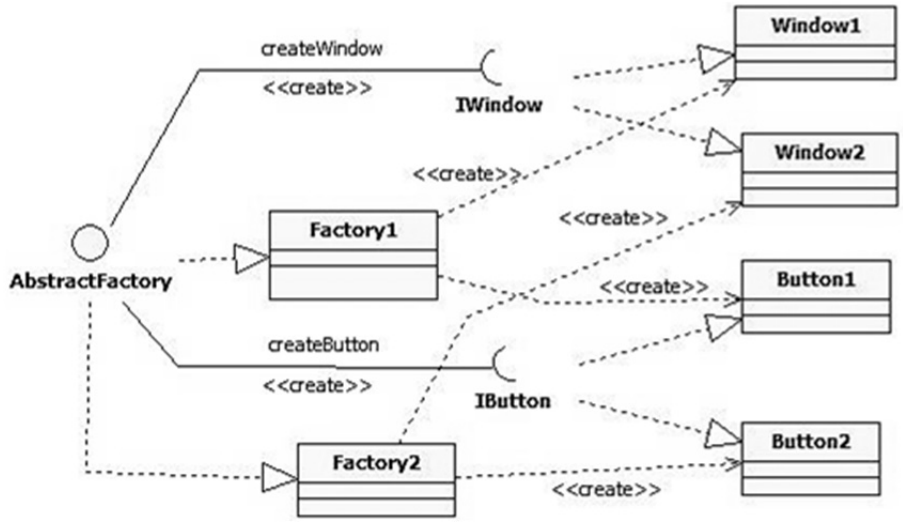


Рис. 3.10. Абстрактная фабрика для классов оконного приложения

Прототип (Prototype)

Группа родственных классов имеет в общем интерфейсе метод копирования (клонирования), программа создает копию объекта-прототипа по ссылке на объект-оригинал через указанный интерфейс. Класс объекта-прототипа и класс создаваемой копии формально программе неизвестны, т. е. могут быть произвольными. В Java на примитивном уровне можно использовать интерфейс *clone* или средства рефлексии – метод *Class.newInstance*. Описанный выше шаблон «фабрика» может использовать коллекцию объектов-прототипов для создания клонов, используемых программой.

Синглетон (*Singleton*)

Обеспечивает единственность экземпляра объекта некоторого класса. В отличие от статической ссылки на объект он создается при первом доступе к объекту и закрыт для прямого доступа по ссылке. Фактически он использует ленивую инициализацию: создает объект при первом обращении к нему через статический метод (рис. 3.11).

```
public class Singleton {
    private static Singleton one=null;
    private Singleton(){ count=0; }
    public synchronized static Singleton getInstance(){
        if (one==null){
            one=new Singleton();
            }
        return one;
        }
```

Рис. 3.11. Синглетон

Синглетон может использоваться для хранения общего контекста приложения.

Пул объектов (Object Pool)

Программа может иметь ограничение на создание объектов определенного класса, например, соединений с БД, сетевых соединений, открытых файлов. Также могут быть существенные временные и ресурсные затраты на создание и утилизацию объектов. В этих случаях используется пул объектов: свободные объекты находятся в пуле, выделяются по требованию, по окончании использования не утилизируются, а возвращаются в пул. Варианты реализации:

- предварительная генерация объектов в пуле при его создании либо создание их по требованию при отсутствии в пуле свободных объектов;
- хранение свободных и выделенных объектов, либо только свободных. Во втором случае пул не контролирует корректность работы программы с пулом;
- наличие или отсутствие ограничения на размер пула;
- действие при отсутствии свободного объекта: создание дополнительного объекта или блокировка запрашивающего потока до появления свободного (см. шаблоны параллелизма);
- действие при освобождении: помещение в буферный пул либо потеря ссылки с последующей утилизацией для сокращения размера пула при наличии ограничений.

Структурные шаблоны

Фильтр (Filter)

Класс имеет тот же самый интерфейс, что и замещаемый. Фильтр получает данные из замещаемого (делегируемого) класса и производит их преобразование, возвращая отфильтрованные данные через собственный интерфейс.

Шаблон имеет много общего с заместителем (*proxy*), но есть существенное различие: он не расширяет функциональность при том же интерфейсе, а фильтрует лишние результаты вызова методов в объекте-делегате.

Адаптер (Adapter)

Вариант 1. Требуется реализовать в классе клиента некоторый интерфейс, например обратный вызов по асинхронному событию. По технологическим соображениям этот интерфейс нельзя навесить на класс-клиент. Например, есть несколько источников событий с таким интерфейсом либо основной класс уже достаточно нагружен присоединенными интерфейсами. Тогда создается класс-адаптер, который присоединяет к себе указанный интерфейс и при этом «видит» класса-клиента. Способы «видения» могут быть разными:

- конструктор получает ссылку на объект класса-клиента;
- класс адаптера является вложенным;
- класс адаптера является анонимным;
- для функционального интерфейса с единственным методом адаптером является *lambda*-выражение – анонимная функция.

Вариант 2. Класс, который обеспечивает сопряжение интерфейсов, незначительно отличающихся друг от друга. Класс-адаптер с целевым интерфейсом получает ссылку на объект с исходным интерфейсом и преобразует обращения к целевому интерфейсу в обращения к исходному.

Композиция (Composite)

Шаблон моделирует древовидную структуру объектов. Общий интерфейс или абстрактный класс *Node* предполагает несколько частных реализаций однокомпонентных классов, а также производный от него класс *NodeGroup*, содержащий вектор ссылок на вложенные компоненты класса, имеющих тип *Node* (рис. 3.12). Рекурсивный характер древовидной структуры может быть отражен на диаграмме классов в явном виде либо условно в виде рефлексивного замыкания класса *Node* в форме композиции с произвольным количеством потомков.

Итератор (Iterator)

Класс – движок по элементам структуры данных, обеспечивающий интерфейс последовательного перемещения по ее элементам: перемещение в начало, конец, вперед и назад, доступ к текущему элементу. Класс итератора является вложенным в класс структуры данных. Этим обеспечивается «видимость»

итератором всей структуры данных, возможность позиционироваться на начало и конец.

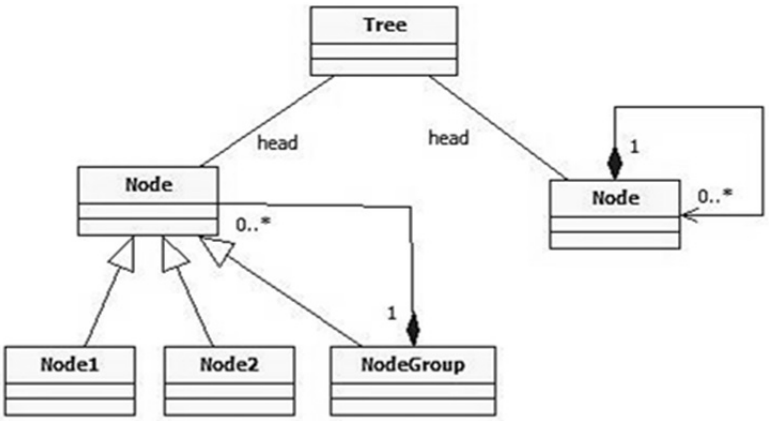


Рис. 3.12. Шаблон древовидной структуры – композиция

Мост (Bridge)

Вариант 1. Используется при наличии двух групп многообразий связываемых компонент. Например, несколько таблиц выводится в файлы нескольких форматов. Группа классов первого многообразия делегируется к объекту группы классов второго через интерфейс (рис. 3.13).

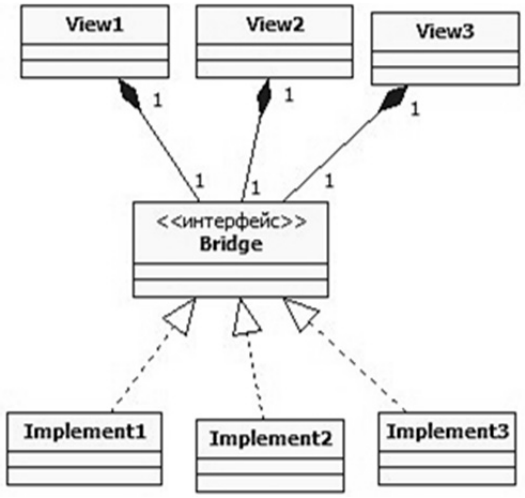


Рис. 3.13. Шаблон «мост» для компоновки двух многообразий

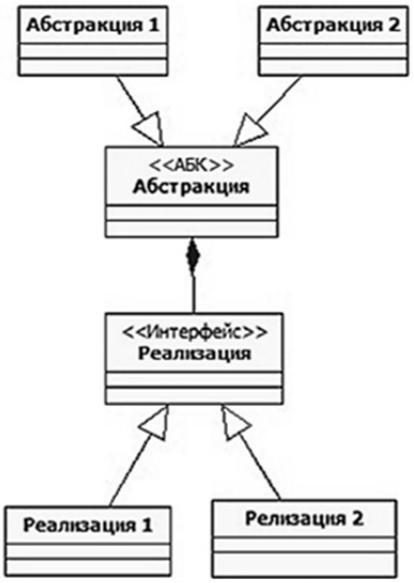


Рис. 3.14. Шаблон «мост» для независимого логического и физического уровней

Вариант 2. Имеется группа связанных классов логического представления и группа классов физического уровня – реализации. Базовый класс логического представления делегируется к объекту физической реализации через интерфейс или абстрактный класс (рис. 3.14).

Вариант 3. Группа классов логического уровня и аналогичные группы классов физического уровня – реализаций связаны по одной схеме. Связывание логического и физического уровней производится через интерфейсы (рис. 3.15).

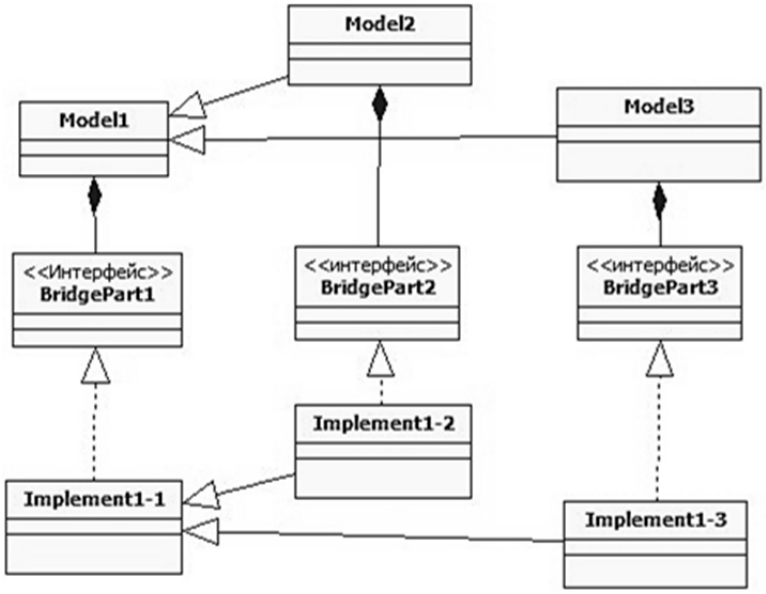


Рис. 3.15. Шаблон «мост» для связывания групп классов

Фасад (Facade)

Технологическое решение. Вместо того, чтобы хранить набор ссылок на требуемые объекты, они переносятся в промежуточный класс, обычно оставаясь открытыми. Появляется возможность использовать фасад в других компонентах как единое целое, а не передавать все ссылки. Кроме того, в фасаде можно реализовать методы, данные для которых локализованы в фасаде.

Декоратор (Decorator)

Расширение функциональности исходного класса. Декоратор присоединяет интерфейс исходного класса, а также имеет собственную функциональность. В отличие от наследования он получает или создает объект исходного класса, к которому делегирует старую функциональность, а затем дополняет ее.

Динамическое связывание (загрузка, компоновка) (Dynamic Linkage)

В Java благодаря рефлексии есть возможность динамической загрузки классов при помощи класса *ClassLoader*. Естественно, загружаемый класс должен иметь оговоренный интерфейс использования, поддерживаемый программой.

Приспособленец (Flyweight)

Вариант 1. Имеет место несколько представлений содержимого одного объекта (рис. 3.16). На объект делаются ссылки со стороны нескольких классов-приспособленцев, каждый из которых обеспечивает оригинальную интерпретацию содержимого.

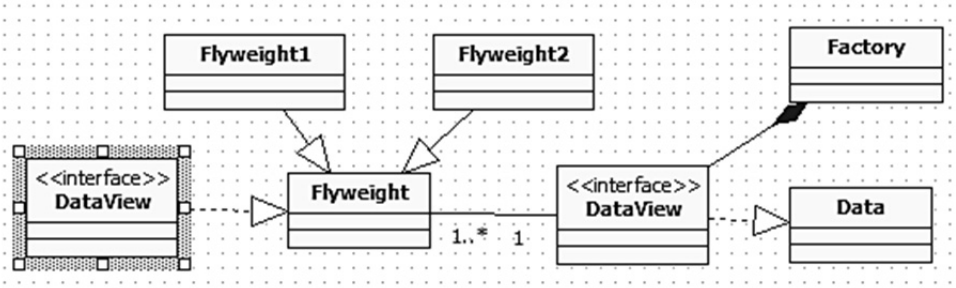


Рис. 3.16. Множественность представлений объекта через приспособленцев

Вариант 2. Обеспечивается эффективное разделение объектов данных с одинаковым содержимым путем раздачи ссылок и контроля изменения объектов (рис. 3.17).

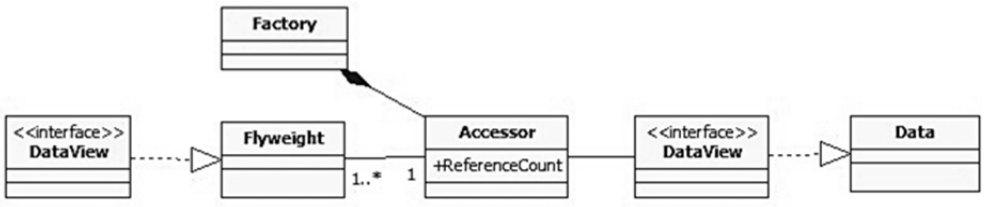


Рис. 3.17. Шаблон «приспособленец» для разделения данных объекта

Например, имеется текст или набор версий текста с повторяющимися словами. Ссылки на объекты данных делаются не прямо, а через объекты доступа *Accessor*, которые содержат счетчики ссылок. Класс фабрики *Factory* контролирует все объекты доступа и через них – объекты данных. В пользовательском коде доступ к объекту данных осуществляется через объект-приспособленец *Flyweight*. При попытке изменить значение объекта данных через приспособленца он может быть изменен только при наличии единственной ссылки. Иначе фабрика создает новый объект данных (или ищет такой же), создает объект доступа к нему и настраивает на него приспособленца. В старом объекте доступа счетчик ссылок уменьшается. При удалении объекта удаляется не сам объект, а приспособленец, а в объекте доступа счетчик ссылок уменьшается. Объект данных удаляется, если счетчик ссылок становится равным нулю.

Аналогичное решение на основе раздачи и контроля ссылок используется для контроля целостности данных и автоматической утилизации объектов. Например, в ядре *Windows* таким образом проверяется возможность безопасного удаления объекта ядра и освобождения памяти.

Управление кэшем (Cashe Management)

Класс-менеджер объектов контролирует процесс загрузки объектов и сохраняет их копии или ссылки на них в кэше. При повторном обращении к тому же объекту по идентификатору он повторно извлекается из кэша. Размер кэша ограничен. Для удаления из кэша лишних объектов при его переполнении в процессе загрузки новых известны *стратегии вытеснения*, например FIFO, LRU.

Объект-значение (Value-Object)

Шаблон обеспечивает передачу результата операции в виде объекта-копии, т. е. не в виде ссылки, а по значению. Значение оригинала при этом не меняется. В Си++ аналогом является передача объекта в метод и возвращение объекта-результата *по значению,* для чего необходим *конструктор копирования.* Паттерн «объект–значение» фактически создает копию объекта с клонированием всех его ссылок.

Поведенческие шаблоны

Конечный автомат (State)

Конечные автоматы широко используются для интеграции поведения в системах, управляемых событиями. Для реализации автомата можно воспользоваться формальным способом задания конечного автомата – *таблицей состояний-переходов,* что плохо вписывается в ООП-нотацию. Для реализации автомата на принципах ООП создается абстрактный базовый класс состояния автомата *State* (рис. 3.18), в который закладывается каркас его поведения. Каждому состоянию соответствует производный класс, который описывает поведение автомата в этом состоянии. Для вывода сгенерированных управляющих воздействий используется интерфейс обратного вызова *ActionCallBack.* Классы состояний автомата вложенные, чтобы видеть данные класса-автомата. Класс самого автомата *StateMachine* имеет ссылку на объект – текущее состояние *current*.

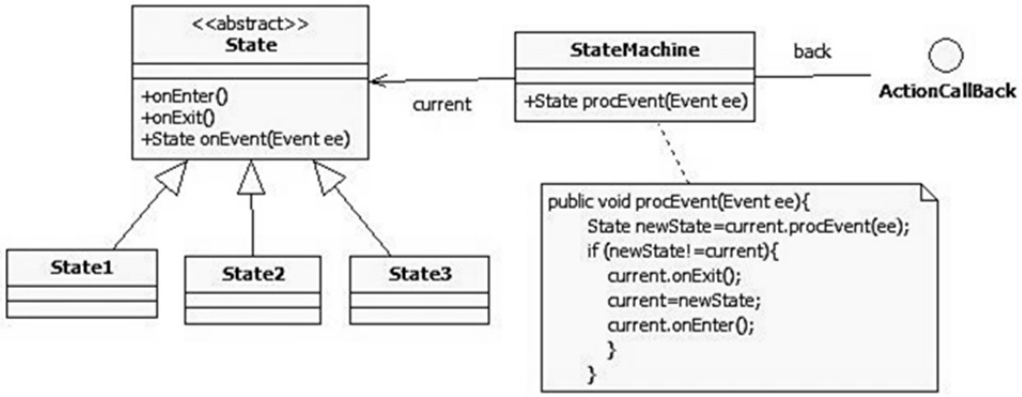


Рис. 3.18. Шаблон «конечный автомат»

Обработка входного события в классе *StateMachine* методом *procEvent* начинается с вызова одноименного метода в одном из производных классов – текущем состоянии. Он анализирует событие и создает объект класса для нового состояния. Если состояние изменилось, *StateMachine* вызывает *onExit* в старом состоянии, *onEnter* – в новом и запоминает объект для нового состояния.

Моментальный снимок (SnapShot)

Шаблон обеспечивает сохранение текущего содержимого объекта и связанных с ним данных в объекте-снимке, а также возможность последующего восстановления содержимого по снимку. В реализации шаблона может использоваться сериализация либо коллекция именованных объектов для сохранения необходимых значений данных. Например, в ОС Android используется именованная коллекция *Bundle* для сохранения приложением собственных параметров состояния при таких событиях, как поворот экрана, уход с переднего плана и т. п.

Последовательность команд (Command)

Базовый класс шаблона создает единый интерфейс для исполнения в наборе команд – производных классов в любой последовательности. Например, сервис исполнения, отмены и восстановления отмененных действий *Do/ReDo/UnDo*. Производный класс запоминает параметры, необходимые для выполнения прямой и обратной команд. Класс-менеджер команд содержит очередь ограниченной длины для команд, текущий обрабатываемый объект. Реализуя методы *ReDo/UnDo*, он выбирает команды из очереди и вызывает в них соответствующие методы. Возможны варианты запоминания состояния объекта для исполнения команд:

- моментальный снимок структуры данных;
- генерация обратной команды. Например, при удалении из строки пятого символа, которым в данный момент является «f», для команды *UnDo* генерируется команда вставки этого символа на пятую позицию.

Стратегия (Strategy)

Базовый класс реализует структуры данных и базовый функционал их обслуживания. Производные классы реализуют различные варианты решения задачи – алгоритмы, стратегии.

Метод шаблона (Template Method)

Аналог абстрактного суперкласса. Абстрактный метод является дополнением основного алгоритма в базовом классе и работает по принципу отложенного программирования при реализации производного класса.

Встраиваемый объект (Pluggable Object)

Аналогичен методу шаблона. При наличии небольших вариаций функциональности основной класс может получать при конструировании ссылку на объект-делегат, реализующий дополнительную функциональность.

Встраиваемый переключатель (Pluggable Selector)

Аналогичен методу шаблона, но при зашитых вариантах функциональности в самом классе – наборе однотипных методов, которые вызываются по имени с помощью рефлексии.

Цепочка ответственности (Chain of Responsibility)

Объекты классов с подключенным интерфейсом обработки сообщения (события) образуют линейную или древовидную структуру данных. Основной класс последовательно вызывает метод обработки сообщения во всех объектах в порядке обхода структуры данных. Если один из объектов обработал сообщение, обход прекращается. В моделях систем контроля сообщения также могут передаваться вверх по дереву иерархии от классов-источников (сенсоров) к классам элементов управления. Элемент управления реагирует на сообщение либо пересылает его выше.

Наблюдатель (Observer)

Средства *широковещательной* или *селектируемой рассылки* сообщений между объектами группы классов с общим интерфейсом *Observer*. Объект «подписывается» на прием сообщений к классу-источнику или классу-диспетчеру, передавая интерфейс *Observer*. Класс широковещательной рассылки, получая или генерируя сообщение, рассылает его всем подписавшимся объектам. Наблюдатель также может быть источником сообщений, передавая его для рассылки классу-диспетчеру.

Маленький язык, интерпретатор (Little Language, Interpreter)

Интепретатор произвольного языка с универсальной или настраиваемой лексикой и синтаксисом.

Посредник (Mediator)

Класс, согласующий состояния связанной с ним группы объектов. Классы-клиенты не взаимодействуют между собой. Вместо этого они посылают посреднику сообщения. Посредник реализует общие принципы поведения системы и рассылает клиентам соответствующие команды (рис. 3.19).

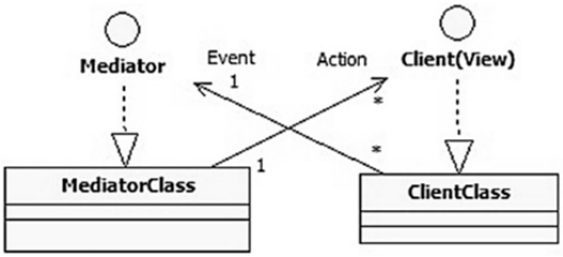


Рис. 3.19. Шаблон «посредник»

Посетитель (Visitor)

Класс, выполняющий обход структуры данных и реализующий некоторый общий алгоритм для всех ее элементов.

Null-объект (NullObject)

В группе классов на основе интерфейса создается класс с *нулевой* функциональностью, замещающий *null*-ссылку. В качестве *null*-объекта может использоваться объект базового класса с *нулевой* функциональностью (рис. 3.20).

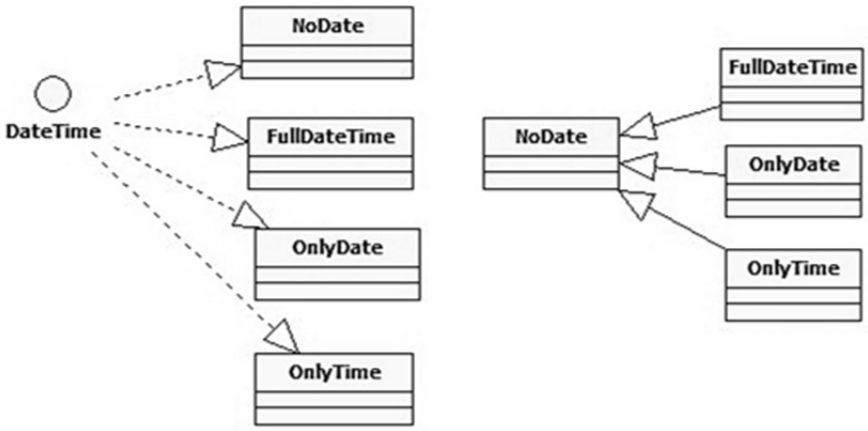


Рис. 3.20. Варианты создания null-объекта

Возможно другое общее решение проблемы – создание контейнера-шаблона, который для объекта любого типа хранит ссылку на объект и логическую переменную состояния, а также ссылку на объект-исключение, если оно связано с появлением невалидного объекта.

Программные компоненты вместо того чтобы возвращать ссылку, *null* или генерировать исключение, возвращают соответствующий вариант контейнера. Вызывающий код использует контейнер по своему усмотрению – передает дальше в существующем виде либо извлекает ссылку и проверяет на исключение.

Шаблоны параллелизма

Шаблоны параллелизма являются наиболее сложными с точки зрения логики работы и наличия возможных ошибок синхронизации.

Однопотоковое исполнение (Single Threaded Execution)

Синхронизированное исполнение группы действий друг за другом, как правило, при разделении ими общего ресурса. Возможны различные реализации в зависимости от требования – исполнять планируемые действия в одном потоке или нет. Возможные варианты реализации:

- блоки *synchronized* в различных классах над общим объектом синхронизации либо метод *synchronized*, вызываемый в одном объекте несколькими потоками. В этом случае не создается общий поток, поэтому каждое действие выполняется в текущем потоке. Имеющиеся недостатки: если синхронизируемое действие занимает много времени, то это тормозит тот поток, в котором оно выполняется;
- используется шаблон «планировщик», организующий исполнение действий последовательно в одном потоке с очередью запросов.

Объект блокировки (Lock Object)

Назначение шаблона аналогично предыдущему. Предотвращает возможные ошибки синхронизации. Создается один общий объект блокировки на всю структуру данных вместо множества независимых элементов синхронизации по отдельным данным и операциям над ними.

Блокировка (Read / Write Lock)

Обращения за чтением данных к ресурсу выполняются параллельно, без взаимной блокировки, операции записи или обновления монополизируют ресурс, синхронизируя все параллельные операции, в том числе и чтения.

Двухфазное завершение (Two-phase Termination)

Основная идея шаблона: работающий поток нельзя просто уничтожить в любой момент времени. Поток должен получить уведомление о необходимости завершиться, после чего он выполняет финальные действия и выходит из метода *run.* Уведомление производится через логическую переменную в классе потока, эта же переменная является условием продолжения его основного цикла. Главный поток устанавливает признак завершения и уведомляет потоки о начале завершения. Потоки завершаются асинхронно, обнаруживая установленный признак. Необходимо следить за количеством незавершенных потоков. Когда все потоки завершатся, главный поток должен выполнить асинхронный обратный вызов к инициатору завершения.

Асинхронная обработка (Asynchronous Task)

По умолчанию все действия приложения, инициированные событиями графического интерфейса, выполняются в потоке графического интерфейса (GUI). Если выполняется другой поток, то для взаимодействия с элементами GUI, его необходимо синхронизировать, т. е. передать исполняемый код методу, который запланирует его выполнение в общей последовательности обработки событий в потоке GUI.

Если необходимо выполнить продолжительное действие и при этом не загружать поток GUI, то необходим шаблон «асинхронное задание» (рис. 3.21). В абстрактном классе запроса на асинхронное исполнение *Task* содержится объект – возвращаемый результат *result*, объявлен метод *onExecute*, который должен быть исполнен в потоке, а также методы нормального завершения *onFinish* и завершения по перехвату исключения onException.

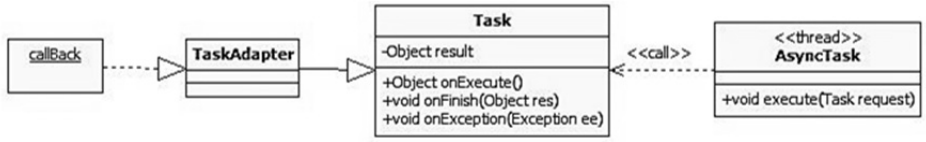


Рис. 3.21. Шаблон «асинхронная обработка»

Класс асинхронного задания *AsyncTask* получает запрос, запускает поток и вызывает метод *onExecute* в запросе, который возвращает объект-результат, сохраняемый в запросе. По окончании *AsyncTask* синхронизируется к потоку GUI и выполняет в нем код завершения *onFinish*, передавая сохраненный объект. Если при исполнении кода в потоке происходят исключения, то они перехватываются, и в потоке GUI выполняется метод *onException* из запроса.

Вызывающий код создает объект *AsyncTask* и запрос – объект анонимного класса на базе *Task* с кодами перечисленных методов.

Планировщик (Scheduler)

В шаблоне «планировщик» действия, инициируемые независимыми запросами, выполняются последовательно в отдельном потоке. Рассмотрим подробности реализации:

- класс планировщика имеет вектор или список объектов-запросов, в котором имитируется очередь;
- запрос представляет собой абстрактный класс, аналогичный описанному в шаблоне асинхронного задания. В него добавлен метод *onCancel*, вызываемый при отмене запросов, которые находятся в очереди на обслуживание при завершении работы планировщика;
- должна быть реализована стратегия планирования – извлечение из очереди очередного запроса на исполнение, в простейшем случае – это FIFO;
- сам планировщик имеет поток, который запускается при создании объекта и выполняет цикл. В начале цикла поток засыпает в ожидании события с помощью метода *wait*. При помещении запроса в очередь инициируется его пробуждение методом *notify* над объектом синхронизации. Поток извлекает из очереди запрос, исполняет запланированный код и планирует код завершения в потоке GUI;
- поток продолжает просматривать и выполнять запросы из очереди, пока она не опустеет, после чего засыпает;
- операции работы с очередью синхронизируются между собой.

Двойная буферизация (Double Buffering)

Шаблон воспроизводит стандартную схему взаимоотношений «поставщик–потребитель» при отсутствии колебаний производительности. Пока потребитель обрабатывает содержимое одного буфера, поставщик готовит другой. По окончании обработки потребитель меняет буферы и пробуждает поставщика.

В приведенной логике есть небольшой дефект. Поставщик должен работать быстрее потребителя. В противном случае потребитель может поменять буферы в момент их заполнения поставщиком. Чтобы этого не произошло, можно ввести систему исключения двух спящих потоков: если один поток уже спит, то он пробуждается и производится обмен, иначе проверяющий условие поток засыпает сам.

Поставщик-потребитель (Producer-Consumer)

Стандартная задача синхронизации потоков генератора и обработчика данных при вариациях производительности. Между поставщиком и потребителем имеется буферный пул блоков, в которые поставщик предварительно подгружает данные. Имеет место блокировка поставщика по заполнению буферного пула и блокировка потребителя по отсутствию данных в пуле, а также синхронизация при выполнении операций с данными в буферном пуле. В простейшей реализации достаточно блокировать потоки на одном объекте, так как поставщик и потребитель одновременно не могут быть блокированы. Код методов буфера используется одновременно потоками поставщика и потребителя.

Пул потоков

В программировании потоков используется шаблон «пул потоков», аналогичный структурному шаблону «пул объектов» с теми же задачами – исключить затраты на создание и утилизацию объектов (в данном случае потоков) и повысить реактивность доступа к ним. Однако простая аналогия с пулом объектов здесь неуместна, поскольку потокам из пула требуется передавать сторонний код, что требует включения в состав шаблона компонент, аналогичных планировщику.

Ограничимся подробной словесной спецификацией шаблона. Диаграмма его классов изображена на рис. 3.22:

- код, исполняемый в потоке из пула, должен быть записан в расширении абстрактного класса *Request* с методами исполнения в потоке, методами завершения без ошибок, завершения по исключению и отмене;
- пулом потоков управляет класс-менеджер *PoolManager*, который включает все структуры данных, связанные с пулом. В нем синхронизированы вызовы от всех внешних компонент;
- сам менеджер также является потоком, в котором реализован алгоритм работы пула. Поток менеджера включает цикл, ограниченный логической переменной *shutdown,* и засыпает в начале каждого шага цикла;
- потоки пула являются объектами класса *PoolThread*. Поток пула также включает цикл, ограниченный переменной *shutdown*, и аналогично засыпает в начале каждого шага цикла. Поток пула имеет ссылку на исполняемый запрос *Request* и ссылку на менеджер, для чего *PoolThread* логично сделать внутренним классом *PoolManager*;

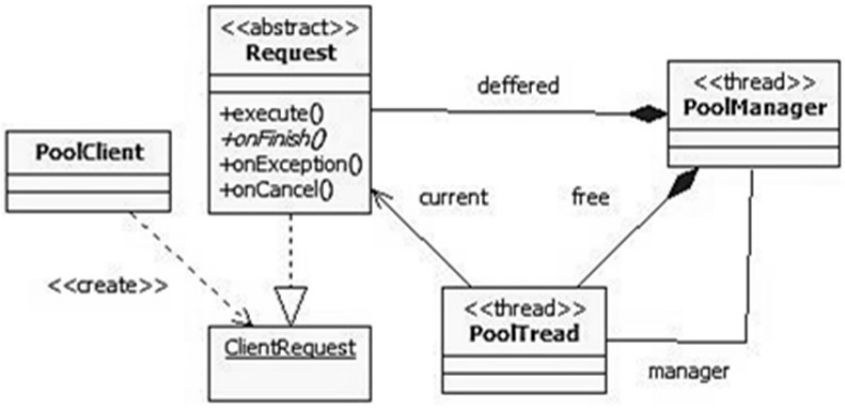


Рис. 3.22. Шаблон «пул потоков»

- менеджер содержит очереди свободных потоков и отложенных запросов;
- при инициализации класса менеджера создаются объекты класса *PoolThread*, и ссылки на них помещаются в очередь свободных потоков, потоки стартуют и тут же засыпают. Поток менеджера также стартует и засыпает;
- класс-клиент создает конкретный запрос *Request* – адаптер обратного вызова на основе анонимного класса с переопределением всех методов в контексте вызова, после чего передает его менеджеру. Метод передачи менеджеру просто включает этот запрос в вектор отложенных запросов и пробуждает поток менеджера;
- поток менеджера содержит цикл со следующей логикой шага: если векторы свободных потоков и отложенных запросов не пусты, то выбирается пара «свободный поток–запрос», в объект *PoolThread* помещается ссылка на запрос, и поток в нем пробуждается;
- пробудившийся поток в *PoolThread* выбирает ссылку на запрос и вызывает в собственном потоке управления метод исполнения в потоке. По окончании он выполняет метод завершения, после чего добавляет себя в вектор свободных потоков и пробуждает *PoolManager*. Если при выполнении кода запроса произошло исключение, то оно перехватывается, поток вызывает в запросе метод завершения по исключению, после чего добавляет себя в вектор свободных потоков и будит *PoolManager*;
- при выполнении процедуры завершения работы используется паттерн двухфазного завершения. Устанавливается переменная *shutdown* в менеджере и классах потоков. В этом режиме менеджер не принимает новые запросы, для запросов из очереди отложенных вызывается метод отмены, менеджер дожи-

дается завершения всех потоков пула, после чего выполняет асинхронный обратный вызов к коду, инициировавшему завершение.

Текущие исполняемые запросы в потоках пула не прерываются, а завершаются естественным образом.

Системные шаблоны

Модель–представление–контроллер (MVC-Model-View-Controller)

Шаблон будет подробно обсуждаться в разделе 4.2.

Сессия (Session)

Позволяет обслуживающему классу-серверу выполнять логически связанную последовательность команд с сохранением в классе промежуточных данных, например, прав, полученных при авторизации или контекста пользователя. При выполнении команды авторизации в классе-сервере создается структура данных – внутренний объект сессии, с которым ассоциируется некоторый уникальный идентификатор сессии – *handle*, возвращаемый классу-пользователю. С помощью *handle* клиент и сервер идентифицируют сессию и связанные с ней данные. Клиент может использовать класс – представитель сессии, хранящий этот *handle*. Диаграмма классов шаблона изображена на рис. 3.23.

Рис. 3.23. Шаблон «сессия»

При реализации шаблона с взаимодействием через сетевое соединение возможно восстановление соединения после аварийного разрыва без авторизации путем введения специальной команды, передающей серверу выданный *handle* для разорванного соединения. Сервер обязан в таком случае хранить в течение некоторого времени *handle* и связанные с ним данные сессии.

Транзакция (Transaction)

Выполнение последовательности методов в серверном классе единым блоком. При возникновении ошибки во время выполнения одного из методов производится откат к начальному состоянию транзакции. Кроме того, транзакция

реализуется как неделимая операция – критическая секция, она должна быть синхронизирована с другими транзакциями. Шаблон транзакции включает:

- примитивы: начать транзакцию, закончить транзакцию, откатить транзакцию – *RollBack*;
- моментальный снимок изменяемых данных в начале транзакции;
- синхронизированное изменение данных – шаблон «блокировка» – *read* / *write lock.*

```
try {              // Транзакция "от обратного" - отмена автоизменений
    dbConn.setAutoCommit (false);     // Начать транзакцию
    ss=selectOne("SELECT MAX(id) FROM "+tname+";");
    if (ss!=null) id=Integer.parseInt(ss[0])+1;
    String sql="INSERT INTO "+tname+" (id)  VALUES ("+id+");";
    execSQL(sql);
    dbConn.commit();                  // Выполнить целиком
    dbConn.setAutoCommit(true);       // Закрыть транзакцию
    } catch (Exception ee){
        dbConn.rollback ();           // Откатить транзакцию
        dbConn.setAutoCommit(true);// Закрыть транзакцию
        throw new SQLException(ee.getMessage());
        }
    return id;
```

Рис. 3.24. Пример кода использования транзакции при работе с БД через JDBC

На рис. 3.24 приведен пример программной реализации транзакции при работе с БД через интерфейс JDBC [88].

3.4. Метрика и качество кода

Взять бы много кирпичей,
штук примерно пять,
Вот бы вышел славный дом,
только где их взять?

Песенка кума Тыквы
из м/ф «Чиполлино»

Ввиду того, что в программировании отсутствуют рамочные законы, позволяющие оценить или рассчитать фундаментальные свойства программного продукта на основе его кода, приходится делать это косвенным образом. В разделе 1.1 было введено понятие *качества кода* – формального соответ-

ствия кода набору правил. Сами по себе правила ничего не гарантируют, но их соблюдение способствует повышению качества процесса разработки в целом.

Цели получения метрических характеристик качества кода:

- поддержание стандартов кода при коллективном владении;
- обнаружение потенциально опасного кода;
- обнаружение узких мест в структуре кода;
- мониторинг разрабатываемого проекта в системе контроля качества при наличии статистики, развернутой по времени.

Текущее состояние в этой области выглядит следующим образом:

- в основном преобладает прагматический подход: используются средства поддержания стандартов кодирования – единообразная стилистика кода и простые количественные метрики;
- отдельно развивается научно-исследовательский подход: разработка метрик, отражающих свойства правильного кода;
- сами по себе метрики не гарантируют эффективной реализации функционала;
- собранная метрическая статистика нуждается в обработке и интерпретации.

Таким образом, метрические характеристики кода в настоящее время достаточно разнообразны, но сами по себе они не являются показателями реального качества программного продукта и *нуждаются в анализе и интерпретации*.

В форме показателей качества могут выступать различные формальные и стилистические свойства:

- свойства стиля:
  - документированность;
  - читаемость;
  - устойчивость к потенциальным ошибкам;
- свойства структуры кода:
  - модульность;
  - сложность;
  - инкапсуляция – сокрытие свойств;
  - управляемость – автономность, независимость, связность.

Метрика – количественная оценка (мера) некоторого свойства кода, создаваемая путем введения параметра, который может его характеризовать.

Говоря о метриках, нужно помнить, что они отражают *формальное качество кода,* которое может способствовать улучшению его реальных качеств, однако это не происходит автоматически. Между реальным и формальным качеством существует определенная корреляция, но это вовсе не означает, что формально некачественный код будет обязательно плохим, а формально качественный – хорошим. Реальное качество можно рассматривать в разных аспектах:

- **развитие**: легкость понимания – внятность, эстетика, универсальность, модульность, управляемость, простота адаптации и повторного использования;
- **надежность:** потенциальное наличие ошибок, устойчивость к ошибкам, трассируемость, восстанавливаемость;
- **эффективность:** производительность, трудоемкость, использование памяти.

Очевидно, что перечисленные аспекты расположены в порядке убывания влияния на них формального качества кода. Различные виды преобразования кода по-разному связаны с его метрикой: *рефакторинг*, как правило, сопровождается улучшением метрики, а *оптимизация и реинжиниринг* меняют структуру кода таким образом, что его метрические характеристики не всегда сопоставимы.

Стилистические метрики

Стилистика кода позволяет сохранять дополнительную информацию о структуре кода в самом его оформлении [8]. Современные IDE помогают программисту ее соблюдать. Стилистику наиболее просто контролировать, можно найти множество частных рекомендаций по оформлению кода. И наконец, стилистика – это стандарт оформления, которого должна придерживаться команда разработчиков, чтобы как минимум не вызывать дискомфорта при чтении чужого кода.

К стилистике причисляются правила, которые имеют самое разное отношение к языку и практике его применения: эстетика, синтаксис и семантика языка, шаблоны конструирования:

- комментарии по сложным участкам кода, *шапки методов* – часть программной документации;
- форматирование текста программы, соответствующее синтаксической структуре кода, например, отступ по уровню вложенности позволяет визуально контролировать синтаксис операторов;

- соглашения об именах переменных, методов, интерфейсов и аббревиатуры позволяют по имени определить принадлежность к определенному функционалу, свойства и особенности именованного объекта. Например, закрытые свойства класса рекомендуется сопровождать префиксом «m» (*mFirstValue*), имена методов содержат аббревиатуры выполняемых действий, типа возвращаемого результата или его формата (*createParamList, getSelectedRecords*);
- предупреждение возможных ошибок при редактировании кода. Например, тело цикла и условного оператора рекомендуется заключать в фигурные скобки, даже если оно состоит из одного оператора на случай будущей замены оператора последовательностью;
- синтаксически допустимые конструкции, которые не согласуются с хорошим стилем программирования. Например, не рекомендуется перехватывать исключения всех типов конструкцией *catch_Throwable(ee)*, а делать это отдельно по разным источникам.

Замечания по теме

1. Любая стилистика имеет свои эргономические и эстетические обоснования. Например, иногда удобнее форматирование, обеспечивающее компактное представление кода, которое позволяет видеть одновременно на экране больший участок текста (рис. 3.25). Это противоречит общепринятым положениям стилистики, например, требованию размещения открывающейся фигурной скобки на отдельной строке.

```
private Throwable eout=null;
private DBItem[] getRecords(DBItem item, final Vector<DBField> ff, String sql)throws Throwable{
    eout=null;
    final Class cls=item.getClass();
    final Vector<DBItem> out=new Vector();
    srv.selectMany(sql, new DBRecordCallBack(){
        @Override
        public void procRecord(ResultSet rs) {
            try {
                DBItem it=(DBItem)cls.newInstance();
                it.setFields(ff);
                it.loadDBValues(rs);
                out.add(it);
                } catch(Throwable ee){ eout=ee; }
            });
    if (eout!=null) throw new SQLException(eout.getMessage());
    DBItem xx[]=new DBItem[out.size()];
    out.toArray(xx);
    return xx;
    }
```

Рис. 3.25. Компактное форматирование кода

2. К стилистике относятся также варианты выбора эквивалентных синтаксических конструкций языка. Например, тело цикла со значительным количеством *break* и *continue* будет более компактным, нежели с эквивалентными *if...else*, к тому же будет содержать меньше фигурных скобок. Однако такая «бейсик-подобная» стилистика не очень приветствуется в структурном программировании.

Количественные метрики

Следующая группа метрик, за неимением лучшего, используется для оценки объема и сложности программного кода и далее, вплоть до трудоемкости программного проекта. Общеизвестной здесь является **SLOC** (*Source Lines of Code*) – количество строк исходного кода, операторов, комментариев.

Применимость SLOC обусловлена разными факторами:

- парадокс меры состоит в том, что более примитивные решения в коде дают более объемный и, следовательно, трудоемкий с точки зрения метрики код. В то же время компактное изящное решение будет проигрывать. Поэтому при сравнении двух функционально идентичных программ мера работает с точностью до наоборот;
- SLOC может использоваться как удельная мера модульности: количество строк кода (операторов) на один модуль, файл, функцию, процент строк с комментариями. Предпочтительными являются короткие функции и модули;
- готовые закрытые решения и сторонние средства – библиотеки, фреймворки – автоматически понижают SLOC за счет реализованного ими функционала;
- визуальные средства конструирования, производящие код или структурные описания, плохо вписываются в метрику SLOC;
- в однородном технологическом процессе с линейкой однородных продуктов и стабильной командой разработчиков SLOC может использоваться как метрика процесса производства.

Программный код формально можно рассматривать как обычный текст на основе ограниченного набора слов – словаря. **Мера Холстеда** рассматривает программу с позиций теории информации как некоторое сообщение, информационная мера которого используется как метрика кода программы. Основные определения меры:

- $n1$ – словарь действий – операторов (ключевые слова, знаки операций, операторы, символы-разделители);
- $n2$ – словарь сущностей – операндов (имена типов данных, переменные, константы);

- $N1$ – количество операторов;
- $N2$ – количество операндов;
- $n1'$, $n2'$ – теоретический словарь программы – словари действий и сущностей всего языка;
- $n1 + n2$ – словарь программы;
- $N = N1 + N2$ – длина программы;
- $V = N \log_2(n)$ – объем программы;
- $V' = N' \log_2 n'$ — теоретический объем программы;
- $N' = n1 \log_2(n1) + n2 \log_2(n2)$ – теоретическая длина программы.

Основной принцип меры: символы программы учитываются в мере линейно, а их многообразие по логарифмической шкале – как степень соответствующей двойки. Пример расчета метрики для функции двоичного поиска приведен на рис. 3.26.

```
//------Двоичный поиск в упорядоченном массиве
 int binary(int c[], int n, int val){  // Возвращает индекс найденного
 int a,b,m;                            // Левая, правая границы и
      for(a=0,b=n-1; a <= b;) {        // середина
      m = (a + b)/2;                   // Середина интервала
      if (c[m] == val)                 // Значение найдено -
      return m;                        // вернуть индекс найденного
      if (c[m] > val)
           b = m-1;                    // Выбрать левую половину
      else
           a = m+1;                    // Выбрать правую половину
      }
 return -1; }                          // Значение не найдено
```

```
n1 = 18     - (,), «,» ,{,},=,<=,==,+,-,/,[.],while,if,else,return,«;»;
n2 = 11     - binary,int,c,n,m,val,a,b,1,2,0;
N1 = 51     - количество операторов;
N2 = 38     - количество операндов;
n  = 29     - словарь программы;
N = 89      - длина программы;
N' = 113    - теоретическая длина программы (словарь);
V = 433     - объем программы;
```

Рис. 3.26. Мера Холстеда для функции двоичного поиска

Замечание по теме. В любом случае в оценке возникают разные нюансы, связанные с синтаксисом языка. Например, как считать синтаксически связанные элементы – парные скобки или символ «запятая» в различных вариантах использования.

Сложность потоков управления и данных

Сложность структуры кода можно оценить, рассматривая поток управления и поток данных программы и их взаимоотношения. Каждый из них можно представить в виде графа, а уже сам граф оценивать как структуру определенной сложности. Самая простая метрика определяется для потока управления.

Граф потока управления отображает возможную связность по управлению – переходы между различными точками программы. Его проще построить на основе блок-схемы программы (рис. 3.27):

- вершины графа – уникальные дуги блок-схемы, *точки останова*;
- дуги графа соединяют вершины, если между ними имеется путь через условие (ромб) или действие (прямоугольник).

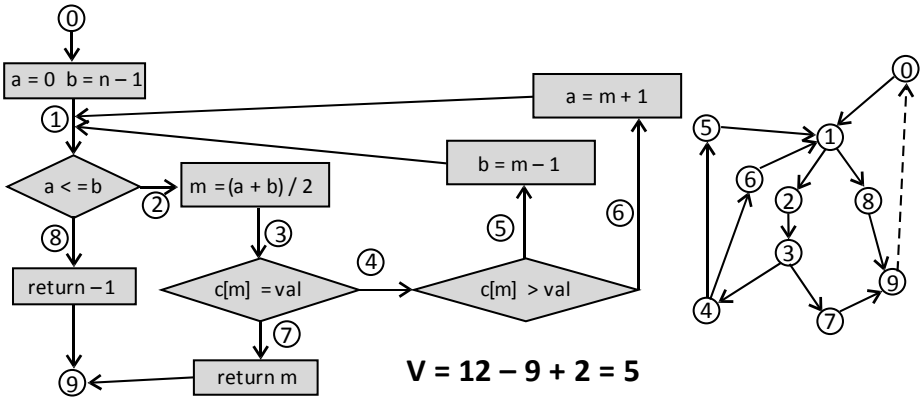


Рис. 3.27. Граф потока управления

Для ориентированного графа, каковым является граф потока управления, известна оценка его сложности – **цикломатическая сложность графа** в виде $V = E – N + 2P$, где E – количество дуг, N – количество вершин, P – число компонент связности. Число компонентов связности графа можно рассматривать как количество дуг, которые необходимо добавить для преобразования графа в сильно связный. Сильно связным называется граф, любые две вершины которого взаимно достижимы. Для графов корректных программ, т. е. графов, не имеющих недостижимых участков от точки входа и *висячих точек входа*

и выхода, сильно связный граф, как правило, получается путем замыкания дугой вершины, обозначающей конец программы, на вершину, обозначающую точку входа в эту программу. В результате получается $V = E - N + 2$. Для нашего примера $V = 12 - 10 + 2 = 4$. Свойства такой оценки:

- длина линейных участков на оценку не влияет;
- для *пустой* программы $V = 1$.

Формально эта метрика указывает превышение числа дуг над числом вершин, а поскольку количество дуг возрастает только при проверке условий, то данную метрику можно рассматривать как показатель *ветвистости* или *запутанности* кода.

Метрики связей модульного кода

Модуль, состоящий из компонент – методы, классы, библиотека функций – может быть охарактеризован с точки зрения способов взаимодействия компонент между собой – **связность (cohesion)** и способа обращения к ним извне – **сцепление (coupling)** (рис. 3.28). Характеристики являются качественными, они отражают *способ связи* компонент.

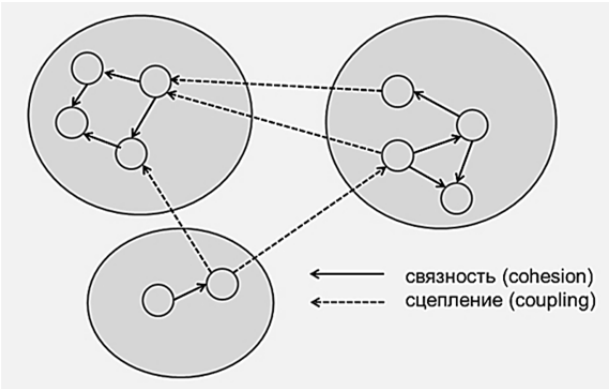


Рис. 3.28. Связность и сцепление

Виды связности обозначаются весами, отражающими степень связности компонент:

- *совпадение* (СВ = 0) – компоненты не имеют ничего общего, кроме факта нахождения в одном модуле;
- *логическая* (СВ = 1) – компоненты используются в рамках общего функционала, но не связаны между собой, например, обработка ошибок различных типов;

- *временная* (СВ = 3) – компоненты используются на одной фазе процесса, в один период времени – инициализация, завершение;
- *процедурная* (СВ = 5) – компоненты используются в рамках одного сценария, имеют определенный порядок вызова, т. е. могут быть связаны между собой по результату;
- *коммуникативная* (СВ = 7) – компоненты работают с общей структурой данных;
- *последовательная* (СВ = 9) – результат первого компонента является входом второго;
- *функциональная* (СВ = 10) – один модуль вызывает другой.

В ООП вводится *объектная связность*. В рамках класса с общими свойствами и функциональностью она может быть любой из перечисленных, но для класса естественным выглядит использование трех последних.

Виды сцепления также обозначаются весами:

- *сцепление по данным* (СЦ = 1) – вызов с параметрами – примитивными типами данных;
- *сцепление по образцу* (СЦ = 3) – вызов с параметрами-объектами, структурами данных;
- *сцепление по управлению* (СЦ = 4) – модуль устанавливает флаги в другом модуле, управляя его поведением;
- *сцепление по внешним ссылкам* (СЦ = 5) – модули используют один и тот же глобальный элемент данных или ссылку на него;
- *сцепление по общей области* (СЦ = 7) – модули разделяют одну и ту же глобальную структуру данных или используют ссылку на общий объект;
- *сцепление по содержанию* (СЦ = 9) – один модуль прямо исполняет часть кода другого модуля.

Сама по себе степень связности или сцепления не является показателем хорошего или плохого кода. Они лишь индикаторы, сигнализирующие о сложностях с *управляемостью кода*. Высокая степень связности и сцепления говорит о том, что существуют косвенные способы влияния одного компонента на поведение другого, что может быть причиной трудно обнаруживаемых и локализуемых ошибок.

Замечание по теме. Способы сцепления и связности имеют различные исторические варианты реализации в языках программирования. *Сцепление по содержанию* используется механизмом **сопрограмм**: существуют два логически независимых потока управления, оба передают управление друг другу по принципу пинг-понга: *A – B – продолжение A – продолжение B*. На уровне архитектуры это можно сделать с помощью команды вызова процедуры *call*

с аргументом – адресом возврата в стеке. В настоящее время сцепление по содержанию реализовано в шаблоне проектирования «обратный вызов» (см. раздел 3.3). Указатели на функции в Си и анонимные функции (лямбда-выражения) также используют способ исполнения фрагментов стороннего кода в текущем компоненте.

Объектно-ориентированные метрики

Понятия связности и сцепления определены в рамках структурно-функционального подхода: функция как основной компонент модуля. В ООП класс представляет собой единство данных и методов, поэтому связность компонент класса можно оценивать как со стороны данных – разделение их методами, так и со стороны методов – использование общих данных (рис. 3.29).

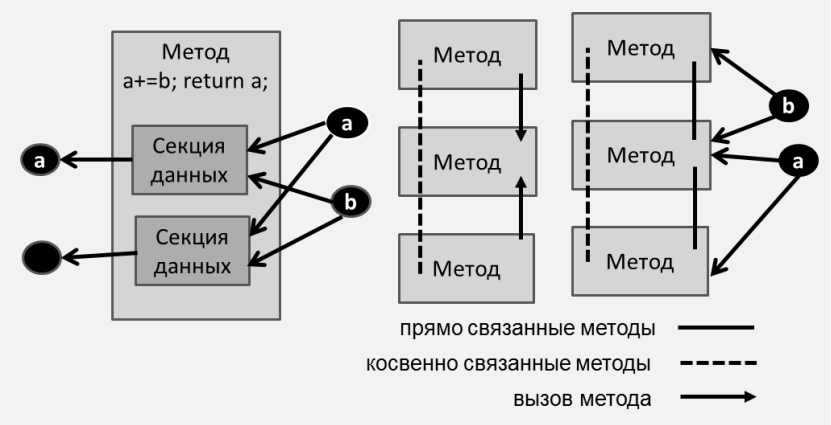


Рис. 3.29. Связность класса по данным и по методам

Метрика связности (cohesion) класса по данным

Метрика связности по данным базируется на следующих определениях:

- *токен* (*token*) – элемент данных класса – переменные, ссылки и константы;
- *секция данных* – набор токенов для вычисления одного из выходных параметров метода – результата метода или элемента данных класса. Количество секций данных метода равно количеству изменяемых им токенов плюс результат метода;
- *склеенный токен* – разделяемый токен, который используется более чем в одной секции данных;

- *сильно склеенный токен* – токен, который используется во всех секциях данных;
- *слабая связанность по данным* (WDC – Weak Data Cohesion) – доля склеенных токенов;
- *сильная связанность по данным* (SDC – Strong Data Cohesion) – доля сильно склеенных токенов.

Финальным параметром является *клейкость данных* (DA – Data Adhesiveness), определяемая как доля дуг *токен-секция* по отношению максимально возможному их количеству, т. е. $DA = N / (S \cdot T)$, где N – количество дуг *токен-секция*, S – количество секций, T – количество токенов.

Существуют технологические нюансы: конструкторы, *set*/*get*-методы в метрике не учитываются, при вызове метода A из метода B секции метода A учитываются в B.

В качестве примера рассмотрим крайние случаи минимальной и максимальной связности.

Вырожденный случай, когда методы не связаны с данными вообще, дает нулевые значения всех параметров. Минимальная связность, имеющая смысл, будет у класса, в котором каждый элемент данных связан с методом, его изменяющим и не возвращающим результата, т. е. в форме 1-1-соответствия:

- токенов – N, секций – N, методов – N;
- SDC = 0, WDC = 0, DA = $N / (N \cdot N) = 1 / N$.

Если же методы будут возвращать измененные значения, то получим:

- токенов – N, методов – N, секций – $2N$, каждый токен связан с двумя секциями данных;
- SDC = 0, WDC = $N / N = 1$, DA = $2N / (2N \cdot N) = 1 / N$.

Максимальная связность имеет место для методов, не возвращающих результат, при использовании ими единственного токена:

- токенов – 1, секций / методов – N;
- SDC = 1, WDC = 1, DA = $N / (N \cdot 1) = 1$.

Метрика связности класса по методам

Связность по методам выглядит проще и рассматривает только факт использования элемента данных класса безотносительно к его изменению или причастности к формированию результата. Выглядит это так:

- пара *прямо связанных методов* имеет общий элемент данных, NT – количество прямо связанных пар;
- пара *косвенно связанных методов* имеют общий метод, с которым они связаны прямо либо вызывают его. NL – количество косвенно связанных пар;

- $NP = N (N – 1) / 2$ – общее количество пар методов;
- TCC = NT / NP сильная связность класса (Tight Class Cohesion);
- LCC = $(NT + NL) / NP$ – слабая связность класса (Loose Class Cohesion).

Из технологических нюансов: связность может определяться как с учетом наследования, так и без него, т. е. для текущего класса.

Прагматические объектно-ориентированные метрики

В качестве метрик используются также наборы прямых структурных характеристик кода класса, которые могут применяться для комплексной оценки качества кода класса. В некоторых случаях имеют место рекомендации по их допустимым значениям.

Метрики Чидамбера и Кемерера предназначены для комплексной оценки качества класса и включают:

1) *взвешенные методы на класс* – WMC (Weighted Methods Per Class) – метрика количества и сложности методов: суммарная нормированная сложность кода методов, например, цикломатическая сложность потока управления, общее количество методов. Варианты: учет унаследованных или только собственных методов;

2) *высота дерева наследования* – DIT (Depth of Inheritance Tree) – максимальная длина пути, количество вершин – классов по дереву наследования;

3) *количество детей* – NOC (Number of children) – среднее количество прямых наследований класса;

4) *сцепление между классами* – CBO (Coupling Between Object classes) – общее количество вызовов методов и использования свойств объектов других классов в коде самого класса;

5) *отклик класса* – RFC (Response For a Class) – количество методов класса плюс количество методов других классов, вызываемых из данного класса. В отличие от CBO учитываются только связи по управлению;

6) *недостаток связности в методах* – LCOM (Lack of Cohesion in Methods) – количество пар несвязанных методов (не использующих совместно хотя бы один элемент данных класса) минус количество пар связанных методов. Если значение становится отрицательным, то оно приравнивается к нулю.

Метрики Мартина предназначены для оценки кода проекта в целом с точки зрения сцепления – внешних связей между категориями классов. Категория классов – группа функционально связанных классов, обычно оформленных в виде пакета:

1) *центростремительное сцепление Ca* – количество классов вне категории, зависящих от классов этой категории;

2) *центробежное сцепление Ce* – количество классов внутри категории, зависящих от классов вне этой категории;

3) *нестабильность I = Ce* / (*Ca* + *Ce*);

4) *абстрактность A* – доля абстрактных классов;

5) в категории сбалансированы абстрактность и нестабильность, если $I + A = 1$, тогда она относится к *главной последовательности*. На этой основе вводятся две метрики:

- $D = \left|(A + I - 1)/\sqrt{2}\right|$ – расстояние до главной последовательности;
- $Dn = |A + I - 2|$ – нормализированное расстояние до главной последовательности.

Метрики Лоренца и Кидда содержат характеристики классов и системы в целом. К тому же они выключают рекомендации для диапазонов значений:

1) размер класса – CS (Class Size) – количество методов, в том числе унаследованных и закрытых плюс количество свойств. CS ≤ 20 методов;

2) количество методов, переопределяемых подклассом – NOO (Number of Operations Overridden by a Subclass). NOO ≤ 3;

3) количество операций, добавленных подклассом – NOA (Number of Operations Added by a Subclass). Для CS = 20 и DIT = 6, NOA ≤ 4;

4) индекс специализации – SI (Specialization Index). SI = NOO · L / M, где L – уровень наследования, M – общее количество методов. SI ≤ 0,15;

5) средний размер операции AOS (Average Operation Size). AOS ≤ 9, SLOC-метрика кода;

6) сложность операции OC (Operation Complexity) – любая из метрик сложности метода (например, цикломатическая, Холстеда). Авторами предлагается собственная калькуляция с весами для различных типов операций;

7) среднее количество параметров на операцию ANP (Average Number of Parameters per operation). ANP = 0,7;

8) количество описаний сценариев NSS (Number of Scenario Scripts) – комплексный показатель для классов, управляющих поведением;

9) количество ключевых классов NKC (Number of Key Classes). NKC > 0,2 от общего количества классов системы;

10) количество подсистем NSUB (Number of SUBsystem). NSUB > 3.

Запутывающие преобразования

Запутывание, обфускация – формальные преобразования кода, не меняющие функционал, с целью защиты от повторного использования кода сторонними лицами. Обфускация производится как над исходным текстом, так и

над скомпилированным кодом – внутренним представлением. Как правило, она сопровождается снижением качества кода, ухудшением его метрик. Способы запутывания нацелены на то, чтобы усложнить анализ программного кода при его реинжиниринге как в исходном тексте, так и после декомпиляции из внутреннего представления:

- **запутывание форматирования исходного текста** (*layout obfuscation*) – стилистические преобразования – изменение имен, форматирования, удаление комментариев, оказывает влияние только на стилистику кода и на SLOC-метрики;
- **запутывание данных** (*data obfuscation*):
  - преобразования размещения переменных, изменение их классов памяти (локальные, глобальные, объектные) и формы кодирования значений вплоть до шифрования;
  - преобразования агрегации – преобразование структур данных, слияние нескольких скалярных переменных в одну, использование неочевидных представлений данных, сжатие, шифрование;
- **запутывание управления** (*control obfuscation*):
  - преобразование агрегации – вставка (*inline*) и вынос (*outline*) функций, копирование кода, раскрутка цикла в последовательность шагов, переупорядочение операторов;
  - преобразование вычислений – вставка мертвого кода, диспетчеризация – замена прямого потока управления вызовом или передачей управления фрагментам кода;
  - табличная интерпретация – автоматные модели поведения программы, переменные состояния;
- **превентивные трансформации** (*preventive transformation*) – преобразования, усложняющие декомпиляцию или деобфускацию двоичного представления.

Запутывание упоминается здесь еще и потому, что некоторые его способы используются в преобразовании кода с совершенно другими целями, например для повышения эффективности системы за счет оптимизации алгоритмов, структур данных.

Замечание по теме. Возможна ситуация, что изящное и компактное решение, которое не просматривается напрямую в коде, по своей природе является результатом запутывания очевидного решения, в котором логика *зашита в код*. Использование таблиц решений, переменных состояния, автоматных моделей вместо многоэтажных условных конструкций делает код более управляемым, но менее качественным с точки зрения его запутанности.

Средства контроля качества кода

Контроль качества кода поддерживается различными средствами:

- автономными программными продуктами;
- облачными сервисами;
- плагинами, встраиваемыми в средства разработки.

Автономные программные продукты – SonarQube

SonarQube [17] – свободно распространяемый проект с открытым кодом, позволяет получить стилистические и количественные метрики качества кода.

Особенности программной архитектуры (рис. 3.30): проект написан на Java, поставляется в виде jar-файла, запускается в режиме командной строки. В корневом каталоге проекта необходимо создать конфигурационный файл с именем, версией проекта, языком разработки и другими параметрами. Результаты анализа кода записываются в БД MySQL. Просмотр производится через web-интерфейс обычным браузером по URL http:// localhost:9000, в качестве эмулятора сайта выступает компонента wrapper.exe, которая запускается отдельным командным файлом.

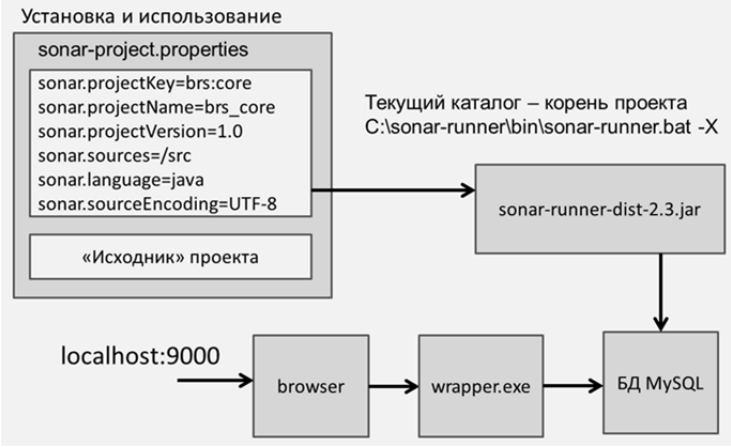


Рис. 3.30. Программная архитектура SonarQube

Функционал продукта:

- оценка объема кода по классам, пакетам, методам в SLOC, документируемость, комментарии;
- поиск дублирования кода – копипаст;
- собственная мера оценки сложности кода;

- около 900 правил кодирования: стилистика, минимизирующая ошибки, стандарты оформления кода, предупреждение технологических дефектов и ошибок;
- оценка времени исправления стилистики.

Собирается статистика по стилистике и количественным метрикам кода для всех компонент (проект, пакет, класс, метод), оценивается его качество (рис. 3.31 и 3.32).

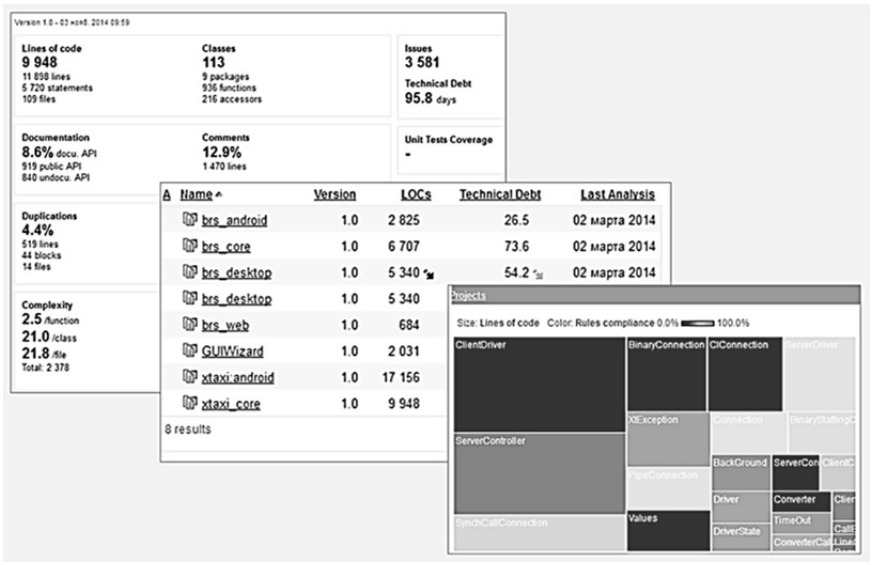


Рис. 3.31. Основные характеристики, оценка качества и статистика кода в SonarQube

Рис. 3.32. Контроль стилистики оформления кода

Средство оценки качества кода в MS Visual Studio

Встроенные средства MS Visual Studio вычисляют комплексный показатель качества кода *Maintainability Index* (рис. 3.33).

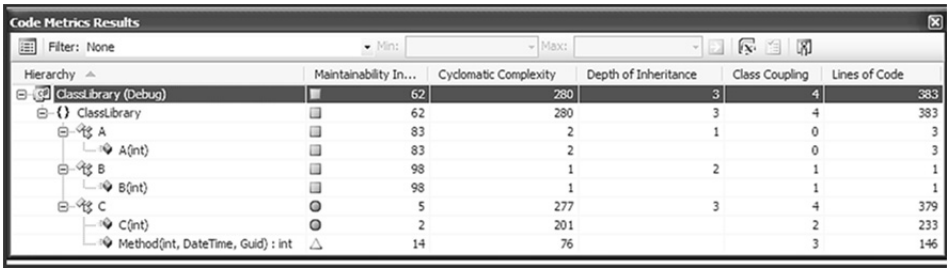


Рис. 3.33. Комплексный показатель качества кода в MS Visual Studio

Показатель вычисляется по классам и методам по формуле:

$$MI = 171 – 5{,}2 * \log (HV) – 0{,}23 * CC – 16.2 * \log (LoC)) * 100 / 171,$$

где HV (*Halstead Volume*) – вычислительная сложность метода (класса);

CC (*Cyclomatic Complexity*) – цикломатическая сложность кода;

LoC (*Lines of Code*) – количество строк кода, исключая пустые строки, комментарии, строки со скобками, объявление типов и пространств имен.

Плагины оценки кода в NetBeans и IntelliJ IDEA

Плагин, встраиваемый в Java NetBeans [18], позволяет получить 33 метрики качества кода для проекта, пакета и класса и экспортировать их в Excel (рис. 3.34).

Аналогичный плагин для IntelliJ IDEA позволяет экспортировать полученные метрики в XML-формате (рис. 3.35).

Online-оценка кода. Облачный сервис Codenforcer

Коммерческий облачный сервис *codenforcer* [19] предоставляет услуги по метрическому сопровождению проектов с исходным текстом на различных языках программирования. Проект должен быть размещен в репозитории на *github*, с которого он скачивается и анализируется. Предоставляемый сервис:

- нормализованные объектно-ориентированные метрики: семь метрик – для пакета, два – для класса;
- рекомендации по исправлению кода;
- оценка стилистики кода;
- поддержка разработки документации к проекту.

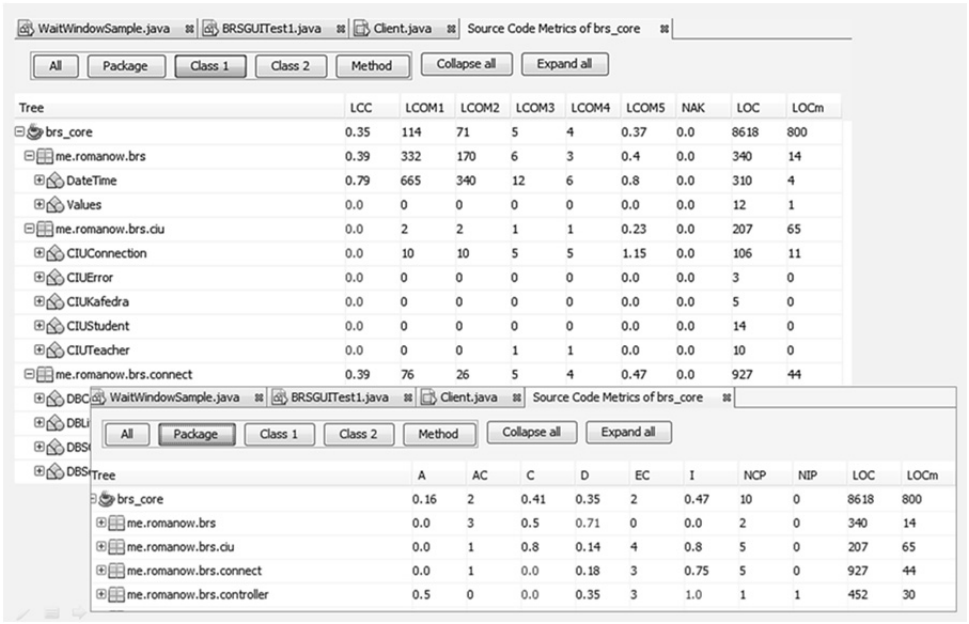


Рис. 3.34. Метрики качества кода на Java в NetBeans

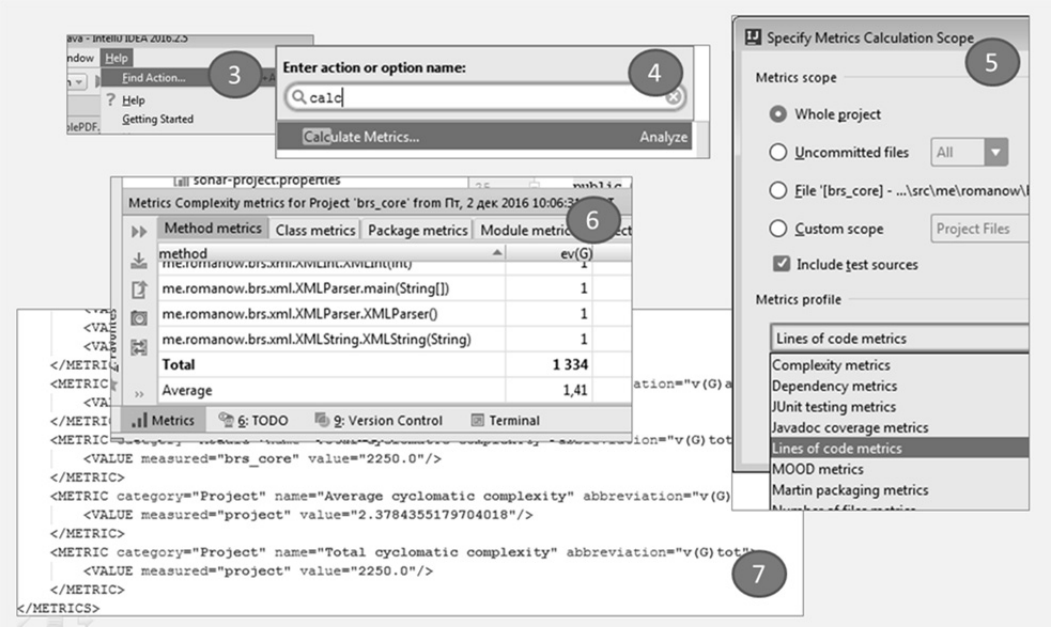


Рис. 3.35. Метрики качества кода на Java в IntelilJ IDEA

В качестве основного набора используются метрики Мартина (рис. 3.36): сцепление, центробежное и центростремительное сцепление, нестабильность, абстрактность, расстояние до главной последовательности – все они характеризуют внешние связи (сцепление) классов в пакетах. Также используется метрика Чидамбера и Кемерера – недостаток связности в методах (LCOM):

- *относительная связность* – доля автономных классов, в которых нет обращений вне пакета;
- *ассоциация между классами* – количество случаев использования данным классом элементов других классов – свойств, методов, констант, перечислений, нормированное количеством использования собственных элементов класса.

#	Metrics	Application	Assembly	Namespace	Package	Class	Interface	Structure	Enumeration
1	Coupling			✓✓✓	✓	✓	✓	✓✓	✓
2	Afferent Coupling			+++	+	✓	✓	✓✓	✓
3	Efferent Coupling			+++	+	✓	✓	✓✓	✓
4	Instability			+++	+	✓	✓	✓✓	✓
5	Relational Cohesion			+++	+				
6	Distance from the Main Sequence			+++	+				
7	Abstractness			+++	+	+	+	++	+
8	Association Between Classes			+++	+	+	+	++	+
9	Cohesion of LCOM					✓	✓	✓✓	
10	Cohesion ofLCOM HS					✓	✓	✓✓	
11	Modularity					++	++	+	
12	OOP Level For Types	++	+						
13	OOP Level For Methods	++	+	++					
14	OOP Level For Fields	++	+	++					

+ - available in codEnforcer metrics ✓ - metrics under development
+✓ - metric applicable for Java, C++, C# and PHP +✓ - specific for Java
+✓ - specific for C# +✓ - specific for PHP +✓ - specific for C++

Рис. 3.36. Виды поддерживаемых метрик в codenforcer

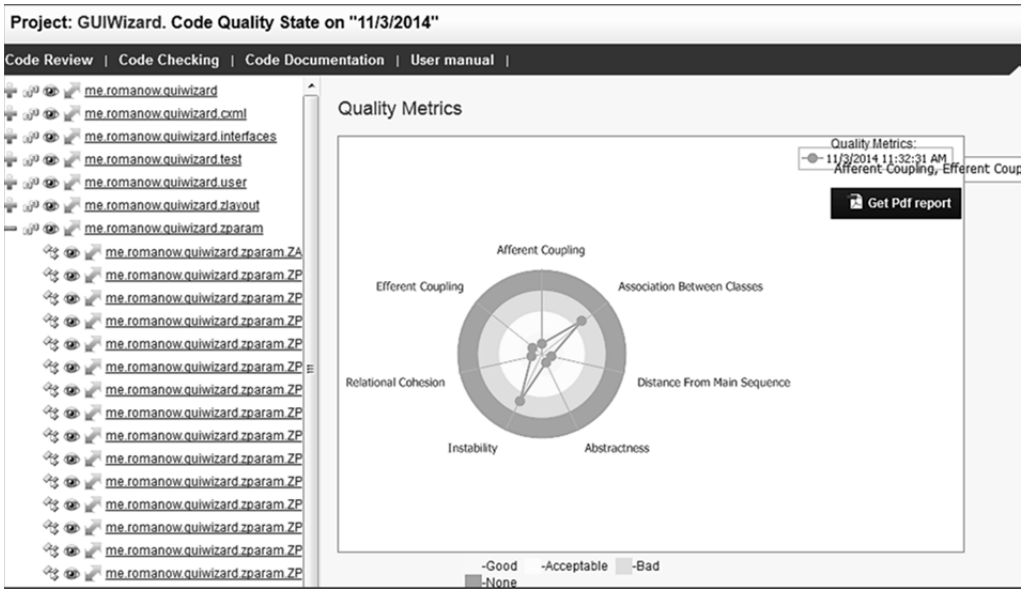


Рис. 3.37. Радиальная диаграмма для нормализованных метрик пакета

Собранные нормализованные метрики для компонент проекта визуализируются с помощью радиальных диаграмм, даются рекомендации по исправлению кода (рис. 3.37).

ГЛАВА 4

ПОПУЛЯРНО ОБ АРХИТЕКТУРЕ. КЛИЕНТ-СЕРВЕРНЫЕ ПРИЛОЖЕНИЯ И ПРИКЛАДНЫЕ ПРОТОКОЛЫ

Начнем обсуждение вопросов архитектуры и архитектурного проектирования индуктивным методом – анализируя и обобщая конкретные решения и общеизвестные паттерны.

4.1. «Слоеный пирог» клиент-серверной архитектуры

Многослойная структура приложения

Обсуждение клиент-серверной архитектуры и связанных с ней терминов начнем с обычных несетевых, локальных приложений. Хотя бы потому, что любое локальное приложение можно превратить в клиент-серверное при помощи простой хирургической операции: распилив его в точке соединения функциональных слоев и включив в место разреза компоненты протокола. Здесь возникает понятие **«функциональный программный слой»**.

Любая более или менее сложная программная система имеет иерархическую слоистую структуру (рис. 4.1). Каждый слой опирается на сервисы, предоставляемые нижележащими слоями.

Рассмотрим с этих позиций любое типовое приложение, включая его программное окружение – операционную среду, в рамках которой оно выполняется:

- **слой 0**: примитивные элементы интерфейса пользователя и устройства, его обеспечивающие: графический экран, отдельные окна приложений, пискели, графические примитивы, события, связанные с нажатием / отпусканием клавиш и кнопок мыши, данные от различных датчиков, сенсоров, изменение положения движков и т. п.;

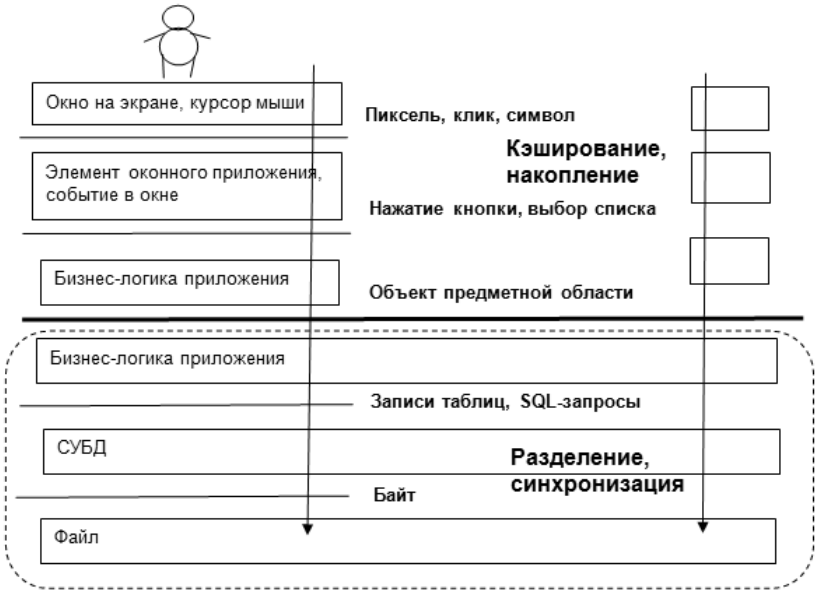


Рис. 4.1. Многослойная структура приложения

- **слой 1**: графические объекты и управляющие элементы оконного приложения. Сюда входят такие компоненты, как меню, кнопки, списки, текстовые поля, а также события, в них происходящие – выбор пунктов меню, нажатие кнопок, выбор элемента списка, потеря и получение фокуса, клик мышью по элементу управления. Этот слой образует *графический интерфейс пользователя* (GUI);
- **слой 2** – собственно код приложения. В свою очередь может состоять из подслоев, реализующих следующие функции:
  - 2.1 – представление (*View*), в которое включаются все элементы реализации интерфейса пользователя;
  - 2.2 – логика бизнес-модели – описание долгосрочного ее поведения (*Controller*);
  - 2.3 – бизнес-объекты – объекты бизнес-модели и методы их преобразования, не связанные с долгосрочным поведением (*Model*);
  - 2.4 – средства сборки объектов бизнес-модели из объектов доступа к данным;
  - 2.5 – объекты доступа к данным – DAO (Data Access Objects);
- **слой 3** – база данных, точнее, система управления БД (СУБД):
  - 3.1 – библиотека программного доступа к БД;
  - 3.2 – сама программная компонента СУБД;

- **слой 4** – данные на физическом носителе (файлы).

Фактически к коду самого приложения относится только слой 2, слои 0 и 4 скрыты и реализуются компонентами операционной системы, слой 1 скрыт средой разработки, слой 3 – сторонние компоненты (приложения или библиотеки). Столь широкий взгляд на вещи объясняется тем, что для клиент-серверных приложений и вариантов их реализации, именуемых **тонкими и толстыми клиентами**, сетевое взаимодействие может осуществляться на границах и внутри всех слоев.

Бизнес-логика, бизнес-объекты, бизнес-слой

Как выглядит разработка простого приложения, использующего БД, по принципу «как получится»? Создается набор экранных форм и обработчиков событий по схеме: *событие – обработчик – SQL-запрос к БД – обработка результатов – препарирование внутренних данных – отображение*. В таком приложении в каждой точке программы присутствуют элементы разных функциональных слоев, а сами слои «размазаны» по всему коду приложения. Например, для изменения структуры БД потребуется отредактировать десяток запросов и связанного с ним кода в разных местах программы.

Без дополнительных аргументов понятно, что внутренняя программная модель предметной области, с которой работает приложение, нуждается в удобном представлении ее бизнес-сущностей, средств описания их поведения и отображения состояния. Отсюда возникают понятия: «**бизнес-объект**», «**бизнес-логика**», «**бизнес-слой**».

Бизнес-объект – представление в программной среде сущности предметной области (**бизнес-сущности**) в виде объекта, системы объектов или структуры данных, а также связанных бизнес-объектов.

Поскольку данные предметной области обычно хранятся в БД, требуются средства сборки бизнес-объектов из записей таблиц БД, а также преобразование связей бизнес-объектов. В этой связи следует упомянуть термины DAO и ORM.

DAO (Data Access Objects) представляет собой средство, позволяющее установить более или менее автоматическое сопоставление реалий ООП-представления данных в программе и реляционной моделью БД. В самом примитивном случае каждой таблице может быть сопоставлен класс с именами свойств, идентичным именам полей таблицы. В классе имеются методы

загрузки по идентификатору записи, добавления и обновления объектов в записи таблиц и наоборот. В более сложном случае возможны манипуляции со связанными записями – группами объектов. В более широком смысле эта же идея фигурирует под термином **ORM (Object-Relational Mapping)** – объектно-реляционное отображение.

Взаимоотношения бизнес-объектов, DAO-объектов и таблиц БД иллюстрирует пример на рис. 4.2:

- бизнес-сущностям «группа» и «студент» соответствуют таблицы БД *Group* и *Student*;
- в БД ассоциация «один ко многим» между сущностями реализована ссылочным полем *idGroup,* содержащим для записи *Student* идентификатор соответствующей записи *Group*;
- DAO-объекты *DBGroup* и *DBStudent* содержат свойства, одноименные и однотипные с полями соответствующих таблиц;
- бизнес-объекты *MDGroup* и *MDStudent* наследуют или ссылаются на соответствующие им DAO-объекты. В любом случае DAO-объекты считаются их неотъемлемой составной частью;

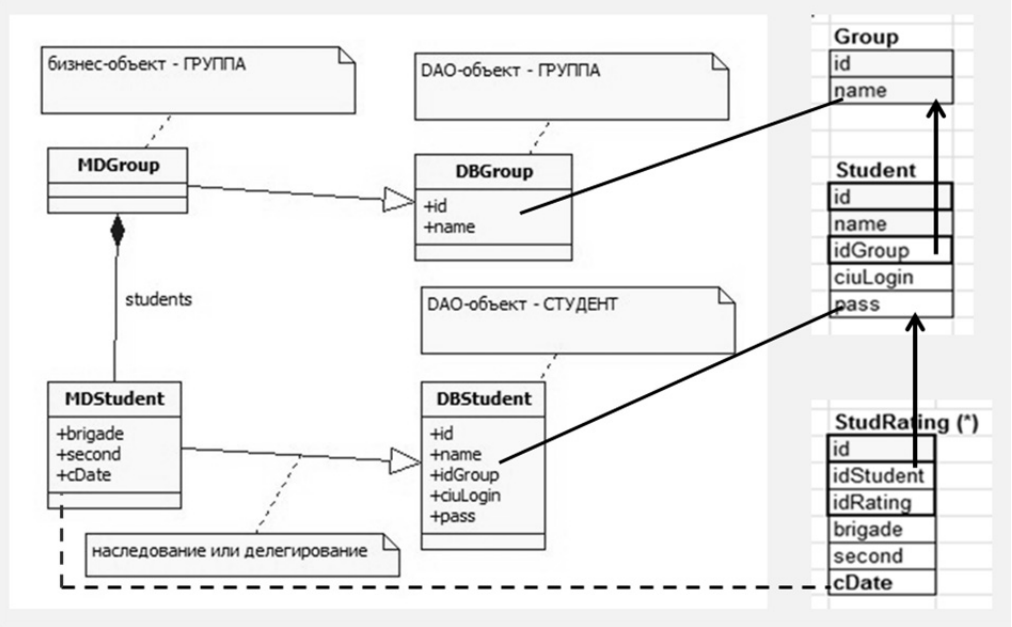


Рис. 4.2. Бизнес-объекты, DAO-объекты и таблицы БД

- ассоциация в БД между студентами и группой отображается композицией между бизнес-объектом *MDGroup* и связанными с ним бизнес-объектами *MDStudent.* На практике это может быть вектор ссылок в объекте *MDGroup*;
- бизнес-объект *MDStudent* содержит дополнительные свойства, связанные с его использованием внутри других бизнес-объектов, например рейтинга успеваемости. Их значения подгружаются из связанной таблицы *StudRating.*

Класс бизнес-объекта *MDGroup* при использовании в бизнес-логике приложения должен обеспечивать определенную целостность представления бизнес-сущности и удобство работы с ней. В нашем случае логично, что загруженный бизнес-объект «группа» содержит ссылки на бизнес-объекты «студент». При наличии значительного числа и глубины связей можно предложить специальное состояние объекта – *глубина загрузки* в вариациях: минимальная, стандартная, полная.

Бизнес-объекты являются также элементом сетевой коммуникации между приложениями, их клиентской и серверной частью. Проще сериализовать весь бизнес-объект (см. раздел 3.2) в стандартный XML- или JSON-формат или собственный двоичный, чем передавать в протоколе отдельные его составляющие.

Многослойная организация приложения как основа клиент-серверной архитектуры

Термин «клиент-серверная архитектура», если не оговорены подробности, сообщает нам не так уж много об организации системы: существует один или несколько типов клиентских приложений, которые обслуживаются серверной компонентой. Логическая структура такой системы практически эквивалентна структуре обычной грамотно спроектированной локальной программы: она содержит несколько функционально-ориентированных слоев. Каждый слой реализуется системой взаимодействующих классов и связан с соседними программными интерфейсами. На рис. 4.3 представлена типовая структура приложения в документе «Руководство Microsoft по проектированию архитектуры приложений» [69].

В приведенной схеме выделены следующие функционально-ориентированные слои:

- **представление (вид, View)** – компоненты и процессы отображения и взаимодействия с пользователем или внешней средой. Первое правило хорошего тона – код взаимодействия с пользователем не должен находиться в компонентах, отвечающих за представление и обработку данных в программе;

- **бизнес-модель** – любая информационная система должна содержать адекватное представление тех внешних физических объектов и их связей, с которыми она взаимодействует или которыми управляет. Второе правило хорошего тона – представление должно быть выполнено в отдельном слое в соответствии с технологией ООП в виде бизнес-объектов и связей между ними, а также классов, реализующих их поведение;

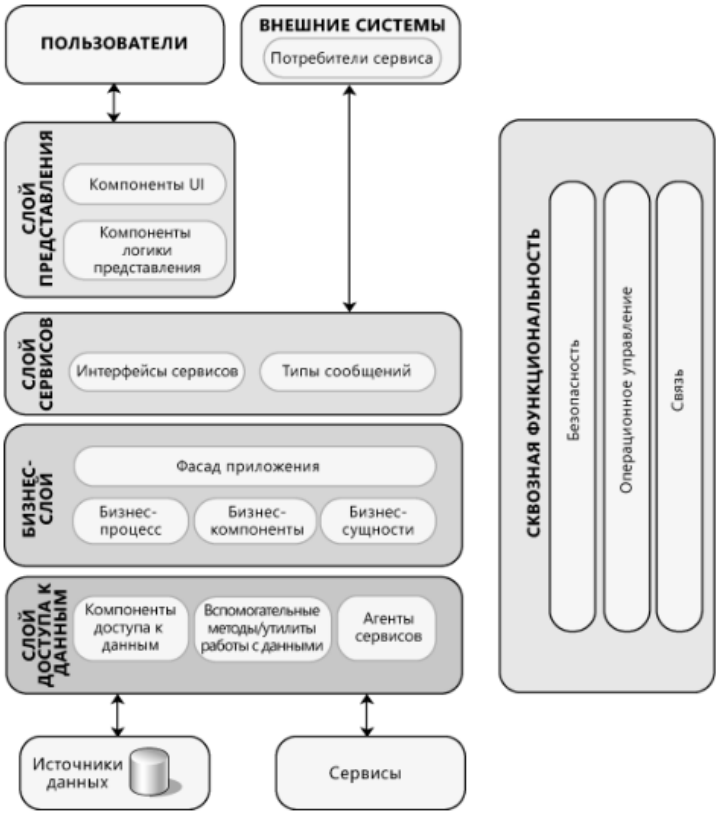


Рис. 4.3. Клиент-серверная архитектура от Microsoft

- **доступ к данным** – объекты бизнес-модели хранятся в таблицах БД, из которых производится их сборка / разборка. Возможна сериализация бизнес-объектов в двоичные или текстовые файлы в XML- или JSON-формате;
- если серверная компонента является универсальной, то она обрамляется **слоем сервиса**, который обеспечивает множественный доступ для сторонних программ. Как правило, это бизнес-слой. Спецификация для работы со слоем сервиса обычно носит название **API**. Протокол программного доступа на основе

протокола HTTP – **WebAPI**. WebAPI может быть оригинальным либо представлять собой альтернативу для удаленного программного доступа к данным существующего web-сервера. Он базируется на таких же HTTP-запросах, что и сервис web-страниц, но в качестве результата возвращает сериализованные текстовые данные в XML- или JSON-формате. Такой сервис является открытым, поэтому он может иметь собственную систему авторизации и защиты;

- в системе также реализуется **сквозная функциональность**, т. е. средства, необходимые для поддержки функционирования на всех уровнях приложения, например обработка сбоев и восстановление, логирование исключений, кэширование данных, синхронизация и т. п.

В клиент-серверной архитектуре разделение компонент между клиентом и сервером делается, как правило, на границе слоев, а программный интерфейс заменяется протоколом.

Граница разделения клиентской и серверной части имеет значение еще и с точки зрения взаимодействия клиентов и синхронизации их работы. *Синхронизацию* и *взаимодействие* проще производить в серверной компоненте, поэтому те слои, в которых ее легче реализовать, переносятся в серверное приложение. В противном случае на сервере необходимы примитивы синхронизации, например транзакции для неделимой последовательности операций в БД. С другой стороны, слои в клиентском приложении могут осуществлять *накопление* и *кэширование* данных, что позволяет реализовать режим автономной работы клиента при отсутствии связи с сервером, а также увеличить его производительность. Но тогда возникает проблема *синхронизации данных клиента с данными сервера*. Такие же проблемы возникают, если клиент изменяет содержимое бизнес-объектов либо создает новые.

Рассмотрим варианты создания клиента в зависимости от выбора точки *разрезания* слоев. Нумерация слоев приложения производится согласно рис. 4.1.

Слои 0–1. Обычно интерфейс между этими слоями скрыт в операционной системе клиента. Фактически он представляет собой сопряжение аппаратных модулей с соответствующими драйверами. Поэтому дилетантская реализация такого супертонкого клиента в универсальных системах не всегда возможна, поскольку требует перехвата потока событий на этом уровне. К тому же объем передаваемых данных в таком случае максимален. Пример реализации: удаленный рабочий стол.

Слои 1–2. Типичный тонкий клиент, который управляется сервером на уровне отображения отдельных графических примитивов, изменения содержимого оконных объектов – тестовых полей, списков и т. п., а также передачи серверу всех событий, происходящих в них – нажатие кнопок, выбор в спис-

ках, получение / потеря фокуса. Данный вариант требует передачи большого объема данных, особенно при использовании графики.

Пример реализации: типичный web-клиент на основе языка разметки HTML. Браузер исполняет роль тонкого клиента, отображая сгенерированное сервером описание графического интерфейса (web-страницы) и уведомляя его о происходящих в нем событиях. В мобильных приложениях тонкие клиенты такого рода не особенно распространены, но могут использоваться, если структура графического интерфейса программируется сервером.

Слой 2. Поскольку этот слой связан с бизнес-моделью, то и объектами протокола являются элементы бизнес-модели – команды управления бизнес-объектами и сами бизнес-объекты. Внутри слоя возможно несколько вариантов установления границы клиент–сервер:

- если граница располагается между подслоями 2.1–2.2, то получается *максимально возможный тонкий клиент*, сами бизнес-объекты и их поведение контролируются сервером, а клиент отображает бизнес-объекты и передает серверу сообщения о действиях, производимых пользователем над ними;
- если граница лежит между подслоями 2.2–2.3, или 2.3–2.4, то клиент уже является *минимально толстым* – управляющая логика приложения находится в клиенте, а на сервер выносятся атомарные методы преобразования бизнес-объектов, а также средства их сборки / разборки из объектов БД.

Слой 3. Стандартным вариантом разделения внутри этого слоя является БД с сетевым доступом, с внутренним протоколом на границе между подслоями 3.1–3.2. Многие СУБД используют соединения сети TCP/IP и свой внутренний протокол для доступа внешних клиентов к БД, в том числе и для администрирования, поэтому принципиальной разницы между локальным и дистанционным доступом к БД нет. Фактически граница разделения клиентской и серверной части находится в библиотеке программного доступа к БД (ODBC, для Java – JDBC). Получившийся клиент является *толстым*. Объем передаваемых данных по сети может быть относительно небольшим, поскольку передаются результаты SQL-запросов, содержащих необходимые выборки данных.

Пример клиент-серверной архитектуры. Система учета рейтинга успеваемости

В приведенной ниже архитектуре имеется несколько вариантов реализации однотипных клиентов, а также ряд функционально подобных клиентов, что создает предпосылки для выделения слоев и компонент с целью повторного использования их кода:

- desktop-клиент куратора-администратора;

- клиентское приложение преподавателя в следующих вариантах: Android-приложение, desktop-приложение, web-приложение;
- клиентское приложение студента в перечисленных вариантах;
- приложение администратора БД.

Программный код некоторых слоев может использоваться в клиентах на различных платформах, а также размещаться либо в клиентском, либо в серверном приложении. Совместно используемый код выделен в библиотеку, компонуемую с различными проектами (рис. 4.4).

Система может использовать три вида серверных компонент:

- сервер базы данных MySQL со стандартными средствами сетевого доступа удаленных Java-клиентов с использованием библиотек JDBC. Сервер обеспечивает непосредственный доступ клиентских программ к таблицам БД с использованием SQL-запросов. Клиентские приложения, взаимодействуя с сервером MySQL, работают в режиме толстого клиента, т. е. реализуют в себе весь функционал, кроме работы с таблицами;
- WebAPI-сервер – серверная компонента реализует в себе бóльшую часть бизнес-логики системы и использует HTTP-протокол для передачи клиентской программе готовых объектов бизнес-модели на основе собственного API. Данные запроса передаются в виде обычной строки запроса к серверу. Клиентские приложения в этом режиме работают как тонкие клиенты;
- Web-сервер содержит набор HTML-страниц клиентского приложения преподавателя. Далее по технологии Ajax организована связь с серверной компонентой, которая содержит слой контроллера приложения и все нижележащие слои.

Для эффективной коммутации слоев использованы следующие решения:

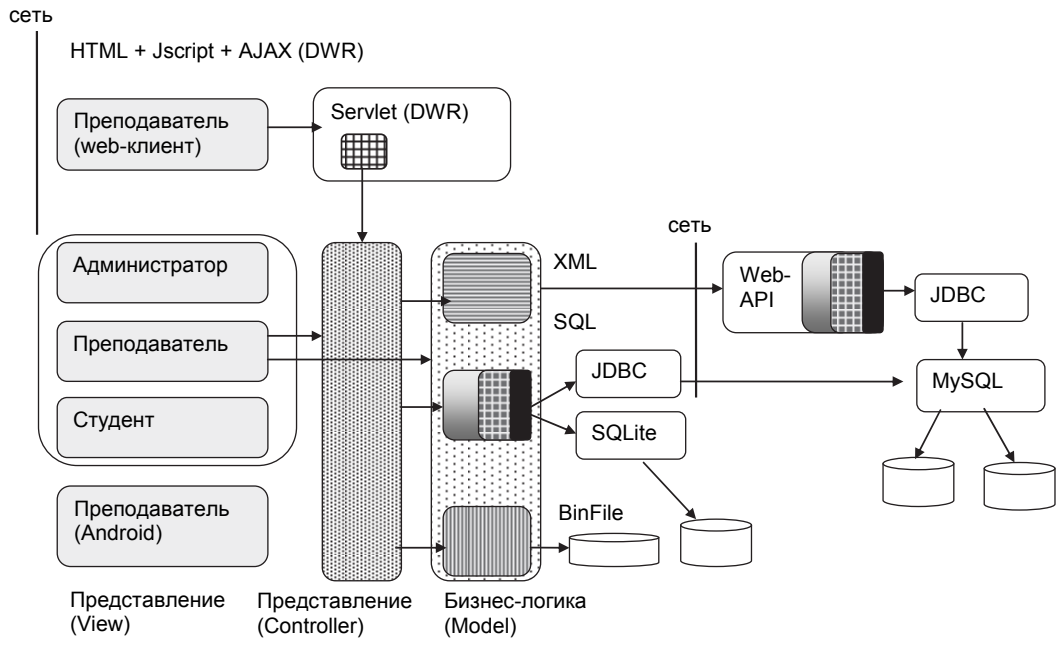
- класс бизнес-логики является абстрактным базовым, производные классы реализуют загрузку бизнес-объектов из двоичных файлов (локальное кэширование), через WebAPI и из БД через слой DAO;
- контроллер бизнес-логики для клиентского приложения преподавателя используется на трех различных платформах и осуществляет платформенно-независимое управление внешним представлением.

Система включает следующие компоненты кода в виде пакетов классов:

1) уровень внешнего представления – оконный интерфейс клиента;

2) контроллер бизнес-логики клиентских приложений преподавателя реализует общую бизнес-логику на различных платформах (Java-desktop, Android, Web);

3) бизнес-объекты, а также базовый класс уровня бизнес-логики, содержащий общую часть алгоритмов работы с бизнес-объектами;



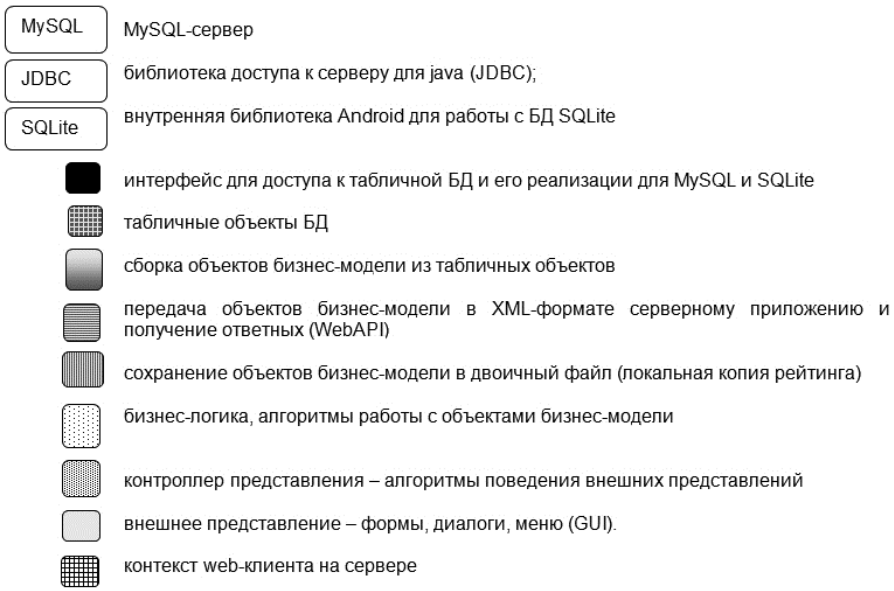


Рис. 4.4. Архитектура системы учета рейтинга успеваемости

4) производный класс для сохранения бизнес-объектов в двоичных файлах;

5) производный класс для сборки бизнес-объектов из DAO;

6) уровень DAO;

7) интерфейс для доступа к реляционной БД и его реализации для БД MySQL и SQLite;

8) библиотеки для программного доступа к БД: JDBC для MySQL и внутренняя библиотека для БД SQLite в Android;

9) собственно СУБД: MySQL – на сервере, SQLite – в Android-клиенте;

10) производный класс бизнес-логики для работы с WebAPI;

11) серверная компонента, реализующая WebAPI;

12) уровень представления web-приложения в клиенте: html-страницы, JavaScript, Ajax, формирование запросов к контроллеру бизнес-логики на сервере и интерпретация ответов;

13) серверная компонента для web-приложения: конвертация запросов web-приложения в команды контроллера бизнес-логики, перехват и преобразование ответов контроллера в скрипты клиента, поддержка контекста соединения с клиентом.

Различные варианты реализации приложений, а также режимы их работы получаются простым соединением слоев, например:

- 1-2-3-4 – автономный клиент с локальной копией данных;
- 1-2-3-5-6-7-8-сеть-9 – толстый клиент на основе стандартной библиотеки доступа к БД;
- 1-2-3-10-сеть-11-5-6-7-8-9 – тонкий клиент с использованием web-сервиса;
- 12-сеть-13-2-3-5-6-7-8-9 – web-приложение.

4.2. Любовный треугольник MVC

Классический архитектурный паттерн и паттерн проектирования

Триада «**модель–представление–контроллер**» (MVC – Model-View-Controller) может интерпретироваться очень широко: от архитектурного принципа независимости описания бизнес-модели от ее визуализации и поведения до конкретного паттерна проектирования, например для работы с отдельным свойством бизнес-объекта.

Рассмотрим, как выглядит паттерн MVC в его первозданном описании [13, 70] (рис. 4.5):

- модель описывает бизнес-сущность и ее атомарное поведение, она существует автономно и способна менять внутреннее состояние под воздействием извне при помощи набора методов;
- представление может запросить и получить данные от модели;
- при изменении данных модель уведомляет об этом представление, для этого представление подписывается на соответствующие события (паттерн *Observer*, см. раздел 3.3);
- для изменения модели представление передает команду контроллеру, который разворачивает ее в набор необходимых действий над моделью.

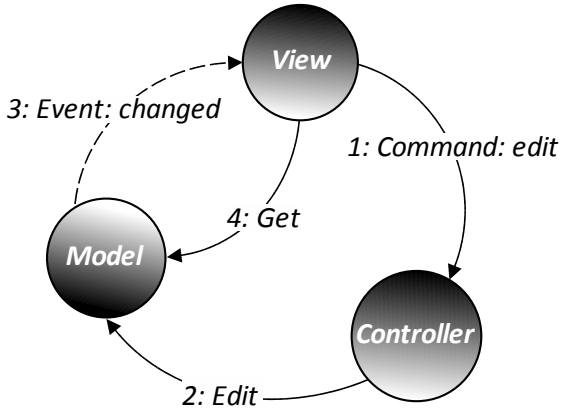


Рис. 4.5. Классический MVC

На рис. 4.6 представлена диаграмма последовательности для такого поведения. Очевидно, что такой вид паттерна подходит, например, для отображения отдельного значения свойства какого-либо бизнес-объекта. Например, в бизнес-объекте имеется свойство – время, оставшееся до некоторого события. Пусть это будет таймер микроволновки. В представлении (экранной форме) ему будет соответствовать поле, а само представление будет подписано на событие – изменение указанного свойства. Событие будет генерироваться моделью каждую секунду, если установлен обратный отсчет. Кроме того, представление может запросить соответствующее значение у модели get-методом.

Изменить значение указанного свойства прямым присваиванием нельзя. Для этого служит контроллер, который проверяет возможность такого изменения и выполняет последовательность команд, например, сбрасывает модель и заново запускает ее с новым интервалом.

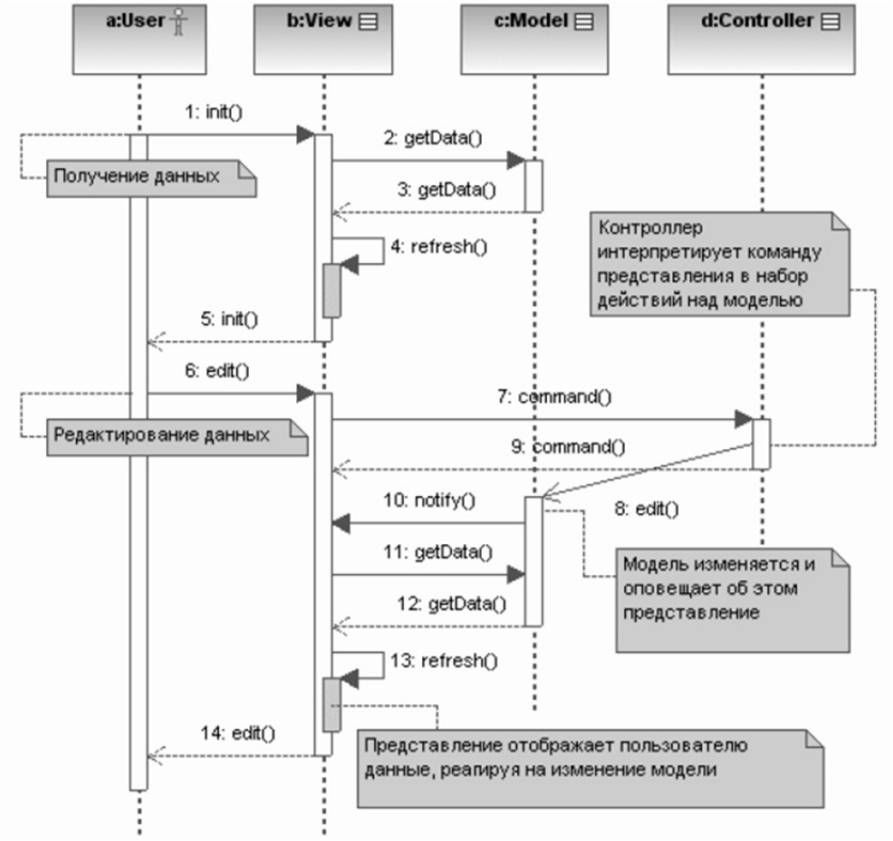


Рис. 4.6. Поведение классического MVC

В таком виде паттерн далеко не идеален. Отметим некоторые недостатки:

- если представление содержит значительное количество полей отображения или связано с несколькими бизнес-объектами, то получится настоящий клубок ссылок;
- контроллер не всегда отслеживает состояние модели, а инициируется только командами представления, события модели контроллеру не передаются.

Архитектурный MVC: контроллер – управление долгосрочным поведением представления

Указанные недостатки шаблона MVC отмечены также в [71]. Там предлагается сделать контроллер посредником между моделью и представлением, отвечающим за *binding* – взаимное соответствие содержания полей представления и свойств бизнес-объекта.

Если же рассматривать MVC на уровне архитектуры, то под представлением следует понимать экранную форму (диалог) или набор основных форм, в которых осуществляется весь бизнес-процесс. Тогда структура и поведение шаблона должны коренным образом измениться. Ключевые идеи такого архитектурного MVC (рис. 4.7):

- под представлением понимается система основных экранных форм, в рамках которых осуществляется бизнес-процесс;
- модель представляет собой набор необходимых бизнес-объектов, которые содержат данные и методы, определяющие их краткосрочное, атомарное поведение. Таким образом, модель не определяет бизнес-процесс в целом, а только набор возможных действий в его рамках;

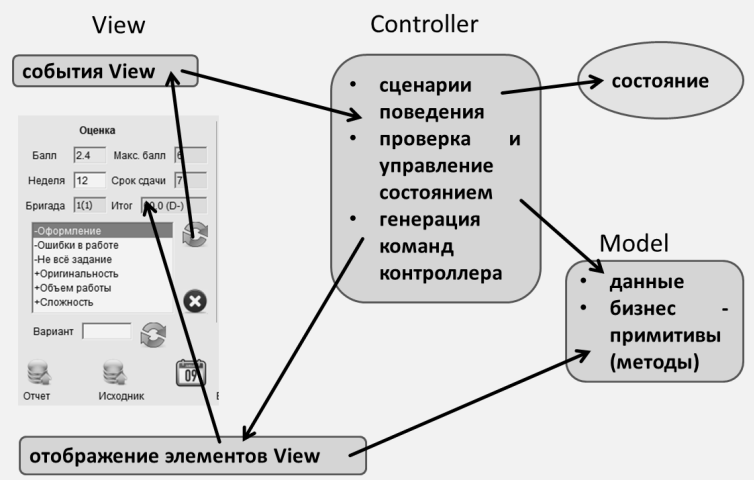


Рис. 4.7. Архитектурный паттерн MVC

- контроллер управляет долгосрочным поведением представления в соответствии с бизнес-процессом. Для этого он отслеживает его текущее состояние на основе автоматной модели (рис. 4.8).

Рис. 4.8. Состояния контроллера при работе с бизнес-моделью

Технологически паттерн работает таким образом:

- главная форма создает контроллер в контексте приложения, при переходе от формы к форме передается контекст, а также интерфейс слушателя событий контроллера. Таким образом, контроллер передается по цепочке активных форм, с которыми он взаимодействует;
- контроллер имеет два встречных интерфейса: интерфейс событий, передаваемых от представления контролеру и интерфейс команд контроллера. Оба интерфейса функционально ориентированы на бизнес-логику и содержат набор событий и команд, необходимых для некоторого абстрактного представления. Их методы передают события и команды, которые могут сопровождать протекание бизнес-процесса, например, *установить видимость группы элементов управления, загрузить вектор имен в выпадающий список студентов* и т. п. Таким образом, контроллер может управлять абстрактным тонким клиентом на уровне элементарных шагов бизнес-процесса;
- контроллер создает модель, связывает ее с источником данных и в дальнейшем обрабатывает события модели, вызывает в ней методы, соответствующие шагам бизнес-процесса.

Помимо основных функций, контроллер в таком виде может выполнять много других общесистемных функций:

- кэширование данных – создание дополнительной бизнес-модели, настроенной на локальный источник данных, использование ее при отсутствии соединения с сервером, синхронизация локальных изменений. Все это в режиме, прозрачном для представления;
- обработка исключений, возникающих в модели и представлении, информирование о них представления через интерфейс событий контроллера в приемлемом для пользователя виде.

Архитектурный MVC обеспечивает максимальное повторное использование кода для приложений, работающих на разных платформах, либо в разных вариациях внешнего вида (см. раздел 7.2).

4.3. Распределенные системы. Компоненты и взаимодействия

В предыдущих параграфах мы рассмотрели, как возникает на практике программная архитектура. «Комок грязи» превращается в многослойную структуру приложения, содержащую пакеты взаимодействующих классов. Такой структурой обладает большинство автономных приложений, на таких же

принципах построены клиент-серверные архитектуры. Для дальнейшего изучения необходимо учитывать два аспекта:

- приложения и их компоненты связываются между собой посредством протоколов. Несмотря на качество предоставляемого сетью сервиса, прикладные протоколы воспроизводят в себе механизмы протоколов нижних уровней сетевого ПО;
- архитектура компонент может быть более сложной, чем простая иерархия слоев, в дальнейшем речь пойдет о более широком термине «распределенные системы».

Распределенные системы

Рассмотренная выше клиент-серверная архитектура является самой популярной, но далеко не единственной **распределенной системой.**

Термин «распределенная система» имеет разные толкования. В широком смысле «распределенная система» – система, элементы которой распределены по пространственно разнесенным **узлам.** Здесь констатируется лишь факт наличия узлов, разнесенных в пространстве.

Если же рассматривать функциональность системы, то исполнение отдельной ее функции может быть как сосредоточенным в одном узле, так и распределенным. В последнем случае можно говорить о функционально распределенной системе. Если же некоторая функциональность системы сосредоточена в одном узле, то это не функционально распределенная система, а система с дистанционным (удаленным) доступом.

И, наконец, если говорить о функциях администрирования, управления и контроля, то можно определить систему с распределенным управлением – это система, в которой административные функции управления и контроля функционально распределены по узлам.

Поэтому, прежде чем говорить о распределенной системе и ее видовой принадлежности, надо описать ее компоненты, распределение функциональности и управления. Рассмотрим несколько примеров.

Компьютерная сеть TCP/IP является функционально распределенной системой и системой с распределенным управлением, поскольку в каждом узле реализуются функции предоставления доступа на уровне логического соединения и сеть не имеет централизованного управления и администрирования.

На прикладном уровне *семейство протоколов HTTP/FTP/SMTP* реализует дистанционный доступ к соответствующим ресурсам – web-сервисам, файлам, почтовым серверам.

Термины «распределенная операционная система», «распределенная файловая система», «распределенная БД» применимы, если основные ресурсы, предо-

ставляемые системой (процессы, операционная среда, файлы, базы данных) не привязаны к конкретному узлу. Для таких систем характерно следующее:

- пользователь не видит отдельных узлов системы, а имеет дело с единым логическим представлением;
- ресурсы могут распределяться по узлам динамически, произвольным образом, иногда ресурс делится на части, которые размещаются по различным узлам;
- имеет место распределенное управление ресурсами.

В противном случае необходимая целостность представления и прозрачность работы не обеспечиваются, и имеет место группа систем с дистанционным доступом.

Централизованная система «клиенты–сервер». На практике большинство простых систем и систем средней сложности представляет собой сервер с уникальной функциональностью и набор разнообразных клиентов, связанных с сервером единым протоколом.

Клиент–сервер: роли и программы, взаимодействия

Распределенная система состоит из взаимодействующих процессов. Сама компьютерная сеть не налагает ограничений на характер обмена данными и способы взаимодействия процессов. Процессы могут передавать данные в любое время по любым соединениям. Однако на практике для каждой пары процессов временно или постоянно устанавливаются роли «ведущий–ведомый» или «клиент–сервер». Термин «клиент» подразумевает не обязательно конечного пользователя, а имеет в виду *инициатора взаимодействия.* Важным является то, что в клиенте происходит *событие,* которое вызывает взаимодействие в распределенной системе, и момент его наступления серверу не известен.

В процессе взаимодействия функциональные компоненты могут играть роль клиента или сервера постоянно, могут менять свои роли, создаваться динамически для соответствующих ролей т. п. Поэтому следует различать *программные компоненты* «клиент–сервер» и *роли* «клиент–сервер», которые обычно закреплены за отдельными потоками в этих программах.

Например, в протоколе FTP при выполнении основных команд роли клиента и сервера распределены естественным образом. При передаче файлов данных и содержимого каталогов в пассивном режиме FTP-сервер выступает в роли сервера, а в активном – в роли клиента, а программа-клиент соответственно – в роли сервера. В программе-клиенте для этих целей создается отдельный поток.

Взаимодействие клиента с сервером может иметь разную степень сложности. В простейшем варианте это *вызов функции – синхронный ответ* (рис. 4.9).

Клиент инициирует выполнение действий на сервере и дожидается результата исполнения. В программной реализации это соответствует обычному вызову функции или метода в объекте-сервере.

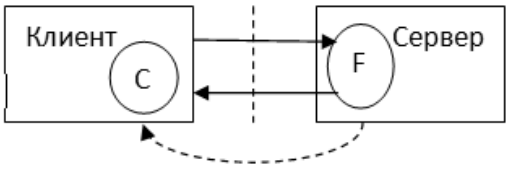


Рис. 4.9. Запрос–синхронный ответ

Для последующего изложения также важен тот факт, что клиент передает серверу данные своего контекста (C на рис. 4.9), в котором он выполняется и с которым сервер может работать.

Если ожидание завершения действия клиентом неприемлемо, то используется схема «вызов функции–асинхронный ответ» (рис. 4.10). Как правило, исполнение действий в сервере осуществляется с использованием внутреннего параллелизма, а ответ возвращается в вызывающий код с помощью *обратного вызова* (см. раздел 3.3). Вызывающий код создает объект-слушатель (L), который выполняет метод обратного вызова в контексте вызывающего кода. Также необходимы средства синхронизации обратного вызова с потоком вызывающего кода (S).

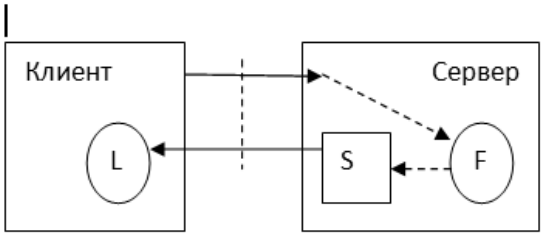


Рис. 4.10. Асинхронный ответ с обратным вызовом

Развивая схему асинхронного ответа, можно реализовать сервер, который однократно инициируется клиентом, после чего клиент может выполнять в нем команды-функции, получая асинхронные ответы на них, а также на события, происходящие на сервере (рис. 4.11). Сервер становится относительно автономной службой с асинхронным внутренним поведением и множественным ответом на запросы.

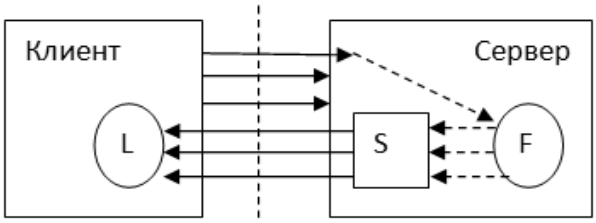


Рис. 4.11. Множественный асинхронный ответ
(асинхронная служба)

В локальной реализации роль сервера исполняет объект соответствующего класса, в котором код-клиент вызывает методы и получает ответы в виде прямого результата либо в виде обратных вызовов в объекте – слушателе событий.

Интерфейсы и протоколы

Способ взаимодействия между программными компонентами распределенной системы зависит от их взаимного расположения. Если взаимодействие локальное, то оно осуществляется через *программные интерфейсы* – непосредственные вызовы, обращения к службам. Если клиент и сервер расположены в различных узлах сети, то взаимодействие между компонентами осуществляется посредством **протокола.** Далее (см. раздел 4.4) будут рассматриваться все нюансы этого термина. Пока достаточно будет определения.

Протокол – описание правил взаимодействия параллельных независимых (асинхронных) процессов путем обмена сообщениями через ненадежную инертную пространственную среду.

В принципе, компоненты протокола могут быть включены в программный код клиента и сервера. Если же следовать принципам модульности и повторного использования кода, то исходные программные интерфейсы взаимодействия следует сохранить, а протокольные – реализовать отдельно (рис. 4.12). В схеме появляются две компоненты протокольных процессов, с которыми у клиента и сервера сохраняются исходные связи.

Чтобы сделать удаленное взаимодействие клиента и сервера окончательно прозрачным, в протокольном процессе сервера создается **агент** – представитель клиента, от имени которого он выполняет локальное взаимодействие с сервером (рис. 4.13). При этом агент создает аналогичный контекст (С) или получает его содержимое от удаленного клиента.

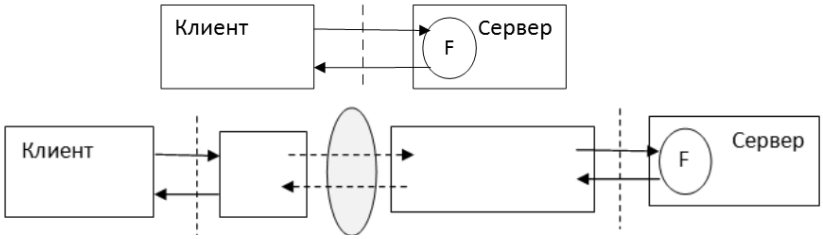


Рис. 4.12. Расшивка локального интерфейса протоколом

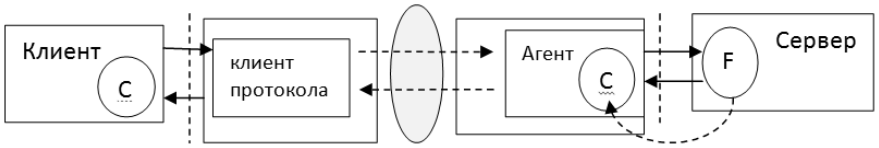


Рис. 4.13. Клиент и агент протокола

Постоянные соединения и транзакции

При проектировании распределенной системы необходимо определить характер соединений между его компонентами.

Короткие соединения – транзакции. Наиболее просто реализуется взаимодействие, если оно происходит в виде короткого сеанса обмена сообщениями без значительных пауз, что имеет свое наименование – *транзакция*. При этом сервер может вообще не использовать многопоточность – запросы клиентов выстраиваются в очередь к нему еще на уровне установления соединения. Однако эта простота имеет существенный недостаток – сервер не сохраняет данные о транзакциях, поэтому клиент обязан каждый раз передавать весь набор параметров, начиная с логина / пароля или идентификатора соединения. Сервер каждый раз должен воссоздавать необходимый контекст для исполнения запроса.

Постоянные соединения. В постоянных соединениях сервер может сохранять данные о состоянии клиента, создавая для него некоторый *текущий контекст*, в котором клиент работает, начиная с авторизации. Эти данные, а также все остальные данные о соединении могут храниться в объекте-потоке, управляющем соединением. Потоки требуют синхронизации при работе с общими данными серверного приложения.

Недостатком постоянного соединения является необходимость проверки его работоспособности – *keep-alive* (см. раздел 4.4). Если это не сделано в самом соединении, то на прикладном уровне применяется передача пустых сообщений, подтверждающих работоспособность. Серверное приложение может закрывать соединения, в которых в течение заданного интервала нет активности.

Например, MySQL-сервер делает это, поэтому клиенту рекомендуется периодически посылать в него фиктивные SQL-запросы.

Постоянные соединения с ограниченным временем существования. Значительное количество постоянных соединений может сопровождаться существенными затратами на их поддержание. Если в течение некоторого времени к серверу нет обращений, то клиент может временно закрыть соединение. Например, в клиенте может работать поток, который по истечении таймаута закрывает соединение, устанавливая необходимый признак. Очередной запрос к серверу, проверяя этот признак, восстанавливает соединение перед выполнением обращения.

Восстанавливаемые соединения. При аварийном разрыве соединения сервер уничтожает поток, контекст клиента, и весь диалог необходимо начинать сначала. Чтобы этого избежать, на прикладном уровне необходимо ввести механизм сессий. При первичном установлении соединения и авторизации клиента сервер передает клиенту уникальный идентификатор (*handle, token*) сессии. При аварийном закрытии соединения сервер в течение некоторого времени хранит контекст клиента и связанный с ним идентификатор. Повторное соединение при восстановлении производится клиентом по старому идентификатору. Его значения должны генерироваться последовательно или псевдослучайно в таком диапазоне, чтобы избежать повторения значения или его подделки.

Стратегии взаимодействия «клиент–сервер»

В приведенных выше схемах программа-клиент всегда выступает в роли клиента, а сервер – в роли сервера. Эта тавтология может быть нарушена, если сервер в процессе взаимодействия перестанет выступать в роли пассивного исполнителя запросов. Для этого возможны различные предпосылки, например:

- клиенты интенсивно взаимодействуют;
- необходимо оперативное обновление у клиента изменяемых данных сервера;
- сервер демонстрирует достаточно сложное собственное поведение;
- клиенту сложно отфильтровать изменения данных на сервере, которые касаются лично его;
- параметры объектов, циркулирующих в системе, могут обновляться как сервером, так и клиентом, необходима синхронизация изменений.

В таких случаях стандартная схема может оказаться неэффективной и привести к необоснованному увеличению трафика или снижению реактивности

системы. На этапе архитектурного проектирования необходимо рассматривать различные стратегии обмена данными между сервером и клиентами. Приведем некоторые из них.

Синхронные запросы клиента к серверу. Единицей взаимодействия в паре программ «клиент–сервер» является передача запроса клиента с ожиданием ответа сервера. Технологическим вариантом такого взаимодействия является удаленный вызов процедур (*RPC – Remote Procedure Call*), когда клиент передает серверу имя и параметры вызываемой процедуры и получает в ответ результаты ее выполнения на сервере.

Циклический опрос состояния сервера (polling). Клиент периодически посылает серверу команды, на которые он отвечает набором данных о событиях, произошедших на сервере и касающихся клиента. Опрос может быть сделан более эффективным за счет повышения частоты при увеличении трафика и постепенном его снижении во время паузы.

Отложенный опрос состояния сервера (long polling). При отсутствии данных сервер может задерживать ответ вплоть до установленного значения тайм-аута, например, до 30 с в http-соединении. Если данные в этот период у сервера не появились, то возвращается пустое сообщение, иначе сервером посылается ответ с появившимися данными. Аналогично сервер может задерживать ответ до накопления нужного количества данных с целью снижения частоты опроса. Недостатком отложенного опроса является необходимость его поддержки отдельным соединением с синхронным протоколом.

Двунаправленная система передачи асинхронных сообщений. Клиент и сервер имеют независимые наборы передаваемых сообщений, на каждое из которых на принимающей стороне предусмотрена обрабатывающая его компонента. В этом случае роли «клиент–сервер» не закреплены на уровне отдельных сообщений, при необходимости клиент должен самостоятельно идентифицировать соответствие ответных сообщений сервера с переданными запросами.

Двунаправленная система передачи синхронных сообщений. Наиболее сложный вариант, в котором как клиент, так и сервер имеют собственный набор команд, которые обрабатываются противоположной стороной, т. е. имеется две пары связок «клиент–сервер». Для идентификации и разделения сообщений из разных связок создается специальный подуровень протокола обмена.

Асинхронное обновление сервером состояния клиента. Для каждого клиента сервер отслеживает изменение состояний объектов, которые касаются только данного клиента, и с требуемой периодичностью выполняет команды обновления клиента, передавая ему соответствующие данные.

4.4. Что такое протокол? Распространенные решения. Терминология

Протокол и интерфейс. Сходства и различия

Теперь позвольте пару слов без протокола,
чему нас учат, так сказать, семья и школа.

В. Высоцкий. «Из милицейского протокола»

Все знают, что термин «протокол» имеет отношение к компьютерным сетям. Но большинство вряд ли сможет точно определить это понятие или хотя бы перечислить его основные свойства. Вот характерные примеры определений.

Протокол передачи данных – набор соглашений интерфейса *логического уровня*, которые определяют обмен данными между различными программами.

Сетевой протокол – набор правил и действий (очередности действий), позволяющий осуществлять соединение и обмен данными между двумя и более включенными в сеть устройствами. Разные протоколы зачастую описывают лишь разные стороны одного типа связи. Названия «протокол» и «стек протоколов» также указывают на программное обеспечение, которым реализуется протокол [74].

Протокол (protocol) – стандарт, определяющий поведение функциональных блоков при передаче данных. Протокол задается набором правил взаимодействия функциональных блоков, расположенных на одном уровне; реализуется одной либо группой программ; описывает синтаксис сообщения, имена элементов данных, операции управления и состояния [75].

Тут вам и функциональные блоки, и соглашения интерфейса, и набор правил и действий, и программы. Рискну дать свое определение.

Протокол – описание правил взаимодействия параллельных независимых (асинхронных) процессов путем обмена сообщениями через ненадежную инертную пространственную среду.

Основные свойства протокола

1. Протокол устанавливается между двумя или более процессами. Они могут быть как аппаратными, так и программными, т. е. процессами или потоками в среде исполнения, начиная от драйверов и заканчивая прикладными программами. Важным здесь является то, что скорость их протекания может меняться в зависимости от обстоятельств, они не имеют иных способов координации работы, кроме обмена сообщениями, т. е. являются асинхронными.

2. Процессы, организующие протокол, передают друг другу *сообщения.* В зависимости от уровня это могут быть как физические сигналы, так и непрерывные или разделенные на блоки последовательности битов или байтов данных – кадры, пакеты, сообщения.

3. Передача сообщений производится через *пространственную среду*, которая по определению является ненадежной, т. е. допускает искажения и потери данных или даже изменение порядка следования. Кроме того, могут быть практически неограниченные задержки.

4. Взаимодействие со средой передачи осуществляется через *интерфейс.* То есть для процесса определены правила сопряжения со средой и порядок передачи / приема сообщений.

Термину «интерфейс» в плане определения повезло больше.

Интерфейс (англ. Interface – сопряжение, поверхность раздела, перегородка) – граница раздела двух систем, устройств или программ, определенная их характеристиками, характеристиками соединения, сигналов обмена и т. п. [76].

У протокола и интерфейса много общего (рис. 4.14). Интерфейс помимо определения правил механического, электрического, программного (алгоритмического) сопряжения также задает порядок действий, обеспечивающий передачу сигналов, данных или сообщений через точку соединения. Однако в интерфейсе соединение является точечным, абсолютно надежным (искажений и потерь сигналов и данных в нем не происходит) и неинерционным. В протоколе же среда передачи обладает прямо противоположными свойствами.

Рис. 4.14. Протокол и интерфейс

Интерфейс – описание соединения в точке с абсолютно надежной передачей сигналов или данных, **протокол** – описание соединения через пространственную среду с искажением, потерями и задержками передачи.

Замечание по теме. Кроме ненадежной среды протокол может *учитывать ненадежность самих протокольных процессов.* Они могут аварийно завершаться по причине сбоев, ошибок программирования, нехватки ресурсов памяти. В конце концов, возможны крах или перезагрузка операционной системы. В простейшем случае это должно сопровождаться разрывом соединения с уведомлением противоположной стороны. Однако в протоколе могут быть предусмотрены и процедуры поддержания и восстановления соединения и потоков данных.

Процессы, реализующие протокол, создают некоторый сервис, которым могут воспользоваться другие процессы-клиенты. В принципе, это может быть устранение некоторых недостатков среды передачи, например, ограничение размеров передаваемых сообщений или восстановление потерянных или искаженных. Тогда мы естественным образом приходим к иерархии протоколов в компьютерных сетях: на первом уровне процессы взаимодействуют непосредственно через физическую среду передачи, а каждый следующий уровень решает ряд проблем, создавая сервис и интерфейс для процессов следующего уровня (рис. 4.15).

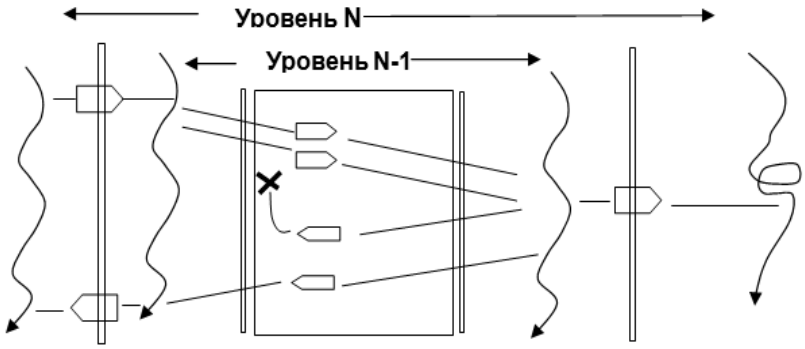


Рис. 4.15. Иерархия протоколов

На самом верхнем, прикладном, уровне реализуются требуемые функции распределенной системы или дистанционного доступа.

Иерархия протоколов в сетях

Компьютерная сеть решает множество разнообразных проблем при передаче данных, поэтому функционал этой системы и его программная реализация являются многоуровневыми. *Иерархия сетевых протоколов* – это иерархия протокольных процессов, в которой группа протокольных процессов некоторого уровня реализует сервис передачи данных и интерфейс к нему,

которым могут пользоваться как прикладные процессы, так и протокольные процессы следующего уровня.

Каждый уровень протоколов характеризуется:

- сущностью предоставляемого сервиса;
- единицей передаваемых данных на этом уровне;
- компонентой архитектуры, реализующей протокол;
- основными задачами, решаемыми протоколом этого уровня и возникающими при этом проблемами.

На этой основе дадим краткую характеристику семиуровневой модели протоколов взаимодействия открытых систем, которой придерживается большинство компьютерных сетей [72].

Физический уровень

Предоставляемый сервис – передача потока двоичных разрядов (битов) через физическую среду передачи данных.

Единица передаваемых данных – двоичный разряд, байт при параллельной передаче.

Компонента архитектуры – контроллер, адаптер к физической среде передачи.

Решаемые задачи: преобразование «поток данных–сигнал», восстановление потока данных из сигнала, синхронизация потока данных, коррекция частоты приемника данных самим входным сигналом.

Канальный уровень

Предоставляемый сервис – двунаправленная достоверная передача потока кадров ограниченного размера между двумя соседними (прямо доступными) узлами сети.

Единица передаваемых данных – кадр – блок данных ограниченного объема, содержащий управляющие и информационные данные, контрольную сумму, заголовок / концевик.

Компонента архитектуры – контроллер с функциями доступа к сети и передачи потоков кадров, драйвер для остальных компонент протокола.

Решаемые задачи: доступ к широковещательной среде передачи данных (подуровень MAC в локальных сетях), установление и контроль состояния соединений с соседними узлами сети, достоверная передача потока кадров – синхронизация начала / окончания передачи, избыточное кодирование, подтверждения приема, тайм-ауты.

Сетевой уровень

Предоставляемый сервис – создание единой системы адресации узлов сети с возможностью передачи пакетов между двумя любыми узлами.

Единица передаваемых данных – пакет – блок данных ограниченного размера без гарантий доставки с адресами отправителя и получателя.

Компонента архитектуры – системный процесс, служба.

Решаемые задачи: маршрутизация – определение последовательности пересылок пакета через промежуточные узлы, управление потоком, управление конфигурацией сети и обмен данными о конфигурации.

Транспортный уровень

Предоставляемый сервис – двунаправленная достоверная передача потока сообщений неограниченного объема между процессами в рамках установленного логического соединения (см. сеансовый уровень). Сервис транспортного уровня повторяет сервис канального, только речь идет не о кадрах между узлами, а о сообщениях между процессами.

Единица передаваемых данных – сообщение – произвольный блок физических данных неограниченной размерности (в протоколе TCP – непрерывный двунаправленный поток физических данных – байтов).

Компонента архитектуры – системный процесс, служба.

Решаемые задачи: разборка сообщения на пакеты и сборка при приеме (в TCP – группирование потока данных в пакеты), восстановление последовательности данных, управление потоком в соединении.

Сеансовый уровень

Предоставляемый сервис – логическое соединение (логический канал) – независимая, идентифицируемая прикладными процессами в узлах сети сущность, обеспечивающая для них функции транспортного уровня.

Единица передаваемых данных – сообщение.

Компонента архитектуры – системный процесс, служба.

Решаемые задачи: адресация процессов и точек доступа в сети, установление, идентификация, поддержание и разрыв соединений.

Представительный уровень

Предоставляемый сервис – создание единой платформенно-независимой сетевой системы представления данных или сервисов.

Единица передаваемых данных – сообщение.

Компонента архитектуры – системный процесс, служба, библиотека.

Решаемые задачи: преобразование форматов, шифрование данных.

Прикладной уровень

На прикладном уровне функционируют процессы, реализующие протокол, определяемый содержанием их взаимодействия. В нем также возможно дублирование функций протоколов нижних уровней – ведение сеансов, восстановление соединений, управление потоками данных.

Стек протоколов TCP/IP

TCP/IP использует средства протоколов канального уровня, предоставляемого программным обеспечением объединяемых узлов – локальных сетей и соединений. IP реализует протокол сетевого, а TCP – транспортного и сеансового уровней [79].

Житейский пример. Разговор по телефону

Функции протоколов весьма разнообразны. В повседневной жизни мы тоже придерживаемся определенного протокола, даже не подозревая об этом. В качестве примера рассмотрим обычный телефонный разговор и перечислим протокольные функции.

Установление соединения. В качестве пространственной среды мы используем мобильную связь, которая создает сервис *устанавливаемого соединения с двунаправленной передачей речи*. Мы используем предоставляемую систему идентификации абонентов – телефонные номера, а также процедуру набора номера. После установления соединения абоненты часто дополнительно называют себя, выполняя *контроль идентификации*, спрашивают, не занят ли собеседник – *проверка наличия ресурсов*.

Подтверждение приема. Иногда, если один из абонентов долго говорит, второй периодически «угукает», подтверждая прием. Со стороны это выглядит забавно.

Избыточное кодирование. Если слово или фамилия не распознаются на слух, переходят к передаче по буквам, например **И**ван, **Г**ригорий, **У**льяна, **А**нтон, **Н**иколай, **А**лексей.

Отрицательное подтверждение. Если абонент не расслышал, что сказал собеседник, он просит повторить сказанное.

Тайм-аут. Если задавший вопрос не получает на него ответа, он через некоторое время его повторяет.

Контроль соединения. Если один из абонентов долго молчит, то другой спрашивает, слышит ли он. Собеседник может отвлечься, может произойти разъединение.

Синхронизация. Несмотря на то что соединение является двунаправленным, как правило, один абонент говорит, а другой слушает. Такова уж особенность человеческого восприятия. При этом говорящий и слушающий периодически меняются ролями, это зависит от логики разговора, например, говорящий задает вопрос. Иногда случается, что оба абонента пытаются перекричать друг друга, но тут уже о достоверности передачи информации не может быть и речи.

Разрыв соединения. Перед тем как разъединиться, абоненты еще раз проговаривают основные итоги разговора, прощаются и только после этого разъединяются.

Составные части описания протокола

Описание протокола должно давать разностороннее представление о процессах, которые его реализуют. В принципе, для этого можно использовать любые технологические средства описания программных систем, например UML. На практике обычно используется текстовое описание – спецификация, которая содержит следующие части.

Функциональность – спецификация функций протоколов. Может быть задана в виде словесного описания предоставляемого сервиса, а также в виде интерфейса к протоколу этого уровня с перечислением видов и форматов передаваемых сообщений – команд и ответов.

Форматы передаваемых сообщений внутри протокола – интерфейс к среде передачи – протоколу нижнего уровня. Включает не только описание правил формирования последовательностей и блоков данных – кадров, пакетов, сообщений, но также форматы представления самих элементов данных. Так как вероятно, что разные компоненты протокола работают на различных платформах и архитектурах, возможны разночтения в представлении данных, например:

- последовательность передачи разрядов передаваемого байта в последовательном канале начиная со старшего или с младшего;
- кодировка символьных данных: байтная – с кодовой таблицей, двухбайтная – Unicode, смешанная – UTF;
- последовательность размещения в памяти байтов машинного слова начиная с младшего байта (*little endian*) или со старшего (*big endian*).

Структуры данных и алгоритмы протокольных процессов. При описании протокола могут оговариваться принципы его функционирования, отраженные во внутренних данных и алгоритмах: переменные состояния, автоматные модели, процедуры обмена, синхронизации, восстановления, поддержки сессий.

Протокол и надежность распределенной системы

Как мы уже видели, протокол исходит из предположения о ненадежности того сервиса, которым он пользуется. Основной его задачей является устранение этой ненадежности. На самом деле речь идет не об устранении, а ее снижении до приемлемого уровня, который устраивает клиентов, использующих этот протокол. В свою очередь, клиенты реализуют собственный протокол следующего уровня, который может доверять надежности протокола нижнего уровня, а может и дублировать некоторые его механизмы.

Например, внутри установленного логического соединения потеря данных принципиально исключается. Однако возможен разрыв соединения как по внутренним причинам самой сети, так и при крахе одного из процессов или операционной системы. Распределенная система может смириться с этим фактом, сообщив о сбое, а может попытаться восстановить соединение после повторного запуска процесса.

Терминология. Синхронный и асинхронный

Сетевое соединение предусматривает равноправие участников обмена, т. е. каждый из них может передавать и принимать данные, когда ему заблагорассудится. Но это не означает, что в конкретном протоколе так оно и будет. Реальный протокол строится на основе элементарных, более жестких взаимодействий, например типа «запрос–ответ», для них существует своя терминология.

Термины «*синхронный*» и «*асинхронный*» в широком толковании обозначают наличие или отсутствие одновременности или привязки во времени. При описании вычислительных процессов, в том числе и протокольных, важен не сам факт простого временного совпадения, так как время исполнения программного кода может зависеть от многих факторов. Термин «синхронный» обозначает наличие причинно-следственной связи в исполнении отдельных частей программного кода или их косвенной связи через используемые данные, следствием чего являются особенности их протекания во времени. Наоборот, термин «асинхронный» обозначает возможность исполнения некоторой части кода в любой момент времени из-за отсутствия такой связи.

В связи с этим *синхронизация вычислительных процессов* – это определение условий, при которых они будут корректно функционировать, их логическая привязка друг к другу.

Термины «*синхронный обмен данными*» и «*асинхронный обмен данными*» в протоколах обозначают наличие или отсутствие логической привязки запроса и ответа.

Термины «*синхронная процедура*» и «*асинхронная процедура*» связаны с логической привязкой исполнения кода процедуры к основному потоку команд. В синхронной процедуре вызов производится явно в определенной точке, т. е. синхронно с исполнением программного кода. Асинхронная процедура вызывается *в контексте основного потока команд*, но в произвольный момент времени, обычно по внешнему событию. Асинхронная процедура является аналогом *прерывания*, но не на уровне архитектуры, а на уровне процесса в операционной системе, для нее должно соблюдаться свойство *прозрачности* по отношению к основному потоку команд.

Терминология. Формат

При передаче данных в протоколе в виде отдельных сообщений возможны вариации их содержимого. Для того чтобы однозначно определить, какого типа сообщение получено, его структура должна быть описана в виде *последовательного саморазворачивающегося формата* типа «скатерть-самобранка». Здесь справедливо все сказанное в разделе 3.2 о форматах представления данных в потоках.

Если протокол определяется как платформенно независимый, то в формате его сообщений также оговаривается порядок следования данных:

- последовательность передачи разрядов передаваемого байта в последовательном канале – начиная со старшего или с младшего;
- последовательность хранения в памяти и передачи байтов машинного слова – начиная с младшего байта (*little endian*) или со старшего (*big endian*). Это справедливо как для многобайтных двоичных данных примитивных типов, так и для многобайтных текстовых в форматах Unicode и UTF.

Что же касается формы представления передаваемых данных, то в одном и том же сообщении могут быть замешаны текстовые и двоичные данные. Если для их чтения используются объекты-потоки различных классов, открытые на одном источнике и не использующие буферизации, то чтение таких сообщений не порождает проблем.

Терминология. Текстовые, двоичные и сериализуемые потоки данных

На прикладном уровне сетевое соединение представляет собой двунаправленный поток физических байтов, на основе которого протокол может открывать *потоки данных* различных типов. Все форматы представления данных, приемы кодирования и передачи идентичны описанным в разделе 3.2. Это касается тестовых и двоичных потоков, собственных форматов текстовых

и двоичных сообщений, сериализации объектов с использованием собственных и стандартных XML- и JSON-форматов.

Распространенные решения из протоколов низших уровней

При разработке протоколов прикладного уровня мы исходим из того, что протоколы низших уровней обеспечивают достаточную надежность и достоверность передачи данных. И это действительно так, если речь идет об установленном соединении в стабильно работающей сети. В этом случае нет необходимости изобретать велосипед. Однако иногда приходится дублировать известные решения из протоколов низших уровней по разным причинам:

- в имеющихся реализациях протоколов не всегда поддерживаются необходимые средства;
- при аварийном разрыве соединения нет гарантии, что все взаимодействия оказались завершенными. Например, если клиент посылает серверу команду и в процессе ожидания ответа происходит разрыв соединения, то нет гарантии исполнения команды сервером. Поэтому на прикладном уровне должна быть своя процедура восстановления сеанса, создающая иллюзию его непрерывности;
- потери и искажения данных, а также зависания в протоколе могут быть обусловлены *внешними причинами, особенностями работы окружающего системного ПО, ошибками программирования* и т. п. Например, при аварийном завершении программы или крахе операционной системы, использующей открытое соединение, на другом конце соединения его разрыв может произойти как минимум со значительными задержками либо соединение может оказаться зависшим;
- возможны более экзотические ситуации, например, у приложения в ОС Android нет 100%-ной гарантии его нахождения в памяти при условии дефицита последней, даже если приложение является фоновым сервисом, причем операционная система может выгрузить приложение без предупреждения. Точнее, приложение предупреждается о возможной выгрузке событием, по которому оно должно на всякий случай сохранить необходимые данные. Отсюда следует, что фоновые Android-приложения, имеющие долгоживущие сетевые соединения, должны предусматривать их восстановление, хотя вероятность этого может быть ничтожно мала.

Тайм-аут

Тайм-аут – интервал времени, в течение которого протокольный процесс ожидает наступления какого-либо события. По истечении тайм-аута выполняется процедура обработки тайм-аута. При наступлении события тайм-аут

отменяется. Тайм-аут используется в канальном и транспортном протоколах для повторной передачи потерянных кадров или пакетов.

На прикладном уровне рекомендуется устанавливать тайм-ауты при работе с низкоскоростными или ненадежными сетями, в которых возможны длительные задержки или зависания при потере связи, например, в мобильных приложениях. Тайм-ауты устанавливаются также на все действия, которые могут зависнуть по каким-либо причинам, вплоть до ошибок программирования, например, на процедуру завершения работы многопоточного сервера.

Поддержка активности соединения

Keep-alive (с англ. «оставаться живым») – периодическая передача сообщений в соединение, свидетельствующее о нормальной работе программы на одном из его концов. Неполучение keep-alive-сообщений на другом конце соединения в течение заданного интервала рассматривается как сбой, соединение принудительно разрывается. Использование keep-alive-сообщений сервером позволяет своевременно исключать зависшие или неиспользуемые соединения.

Некоторые протоколы и библиотеки сетевого доступа позволяют программно устанавливать и отменять режим keep-alive.

В протоколах канального уровня узел сети может рассылать широковещательные keep-alive-сообщения для определения конфигурации абонентов сети и их работоспособности.

Избыточность. Контроль достоверности передачи

Протоколы низших уровней обеспечивают достаточную достоверность передачи данных, прежде всего за счет их избыточности. Как правило, это различные виды контрольных сумм, которые в зависимости от уровня избыточности позволяют обнаруживать отдельные ошибки, исправлять их, обнаруживать двойные ошибки и т. д. За подробностями следует обращаться к дисциплинам «Кодирование информации» и «Помехоустойчивое кодирование». На прикладном уровне в подобных средствах нет необходимости.

Прозрачность информационных потоков данных

Передаваемый по сети поток данных протокола представляет собой последовательность, имеющую определенный формат. Он содержит символы команд и ответов – *управляющие данные*, которые используют протокольные процессы для собственных нужд, а также блоки данных, которые они передают на более высокий уровень – *информационные данные*.

Если на информационные данные не накладываются ограничения, то в них возможны любые значения, в том числе соответствующие кодам управляющих символов команд и ответов. Чтобы не спутать информационные данные с управляющими, необходима перекодировка с внесением *избыточности*, чтобы полученные коды управляющих и информационных данных не совпадали. Этим обеспечивается *прозрачность* информационных данных по отношению к управляющим.

Рассмотрим наиболее известные решения проблемы прозрачности и синхронизации в протоколах.

Байт-стаффинг, бит-стаффинг. Протоколы, использующие непрерывные потоки символов или битов, могут не иметь технической возможности для выделения начала / окончания блока по паузе в процессе передачи. В них используются избыточные символы или биты, создающие уникальную последовательность кодов для обозначения заголовка / концевика кадра путем вставки символов (битов) в передаваемую последовательность данных – *стаффинг*.

Пример байт-стаффинга. Заголовок и концевик кадра передаются парой символов DLE-SYN, каждый символ DLE в кадре дублируется, т. е. передается в виде DLE-DLE.

X X DLE X DLE DLE X -> DLE SYN X X X DLE-DLE X DLE-DLE DLE-DLE X DLE SYN

Пример бит-стаффинга. В непрерывном битовом потоке разделитель кадров данных содержит не менее шести единичных битов и стартовый нулевой бит, в самих кадрах данных после пяти подряд идущих единичных битов вставляется нулевой.

101110011111111000 -> 1111110101110*1111*0111000111111…

Возможен и другой формат передачи информационных данных, когда передаваемый блок предваряется счетчиком длины, в таком случае обеспечения прозрачности не требуется. Но при этом не решается *проблема синхронизации потоков данных* (см. ниже).

Протокол передачи файлов FTP. Сам протокол реализуется на текстовом байтном потоке, а для передачи файла устанавливается дополнительное соединение, по которому он передается в виде двоичного байтного потока. О длине передаваемого файла и номере порта, на котором будет установлено соединение, клиент и сервер договариваются путем обмена соответствующими сообщениями. Возможен *пассивный* режим сервера, в котором он передает номер порта клиенту и ждет запроса на соединение, и *активный* режим, в кото-

ром в аналогичной роли выступает клиент. Таким образом, управляющие и информационные данные никогда не смешиваются в одном потоке.

Почтовый стандарт MIME. Расширение почтового стандарта позволяет передавать в текстовом потоке протокола электронной почты двоичные данные – прикрепленные файлы. Для обозначения начала / конца раздела используются уникальные текстовые метки, а для кодирования двоичных данных – метод Base64, в котором для представления трех двоичных байтов используются четыре символа из 64-символьного алфавита, т. е. фактически с шестиразрядным кодом. Сюда включаются латинские буквы, цифры и некоторые символы. Пример кодирования Base64:

0x44 0x66 0x77 0x55 -> MHg0NCAweDY2IDB4NzcgMHg1NQ==

Строковая константа с Си. Прозрачность необходимо соблюдать и в формальных символьных системах, например, в текстах программ на языке программирования, когда символы, используемые в определении синтаксиса объекта, могут появиться и в его содержимом. Например, *в текстовой строке на языке Си* символы одиночный и двойной апостроф, а также обратный слэш должны предваряться обратным слэшем, например

char c[]="Это текстовая строка с символами \",\' и \\";

Синхронизация потоков данных

Логическое соединение должно обеспечивать передачу потоков данных без потерь. При различных сбоях в протокольных процессах операционная система должна гарантированно разрывать или сбрасывать логические соединения. Поэтому протокольный процесс будет правильно разграничивать передаваемые сообщения в потоке данных, зная их форматы.

Однако такая идиллическая картинка не всегда соответствует действительности. Даже если все совершенно в сети, ошибки программирования протокольного процесса могут приводить к тому, что процесс начнет чтение сообщения не с начала, а с середины.

Аналогичная проблема существует и на физическом уровне. В принципе, приемник может начать прослушивание канала в любое время, т. е. с середины последовательности. Для решения этой проблемы используются различные методы обеспечения прозрачности передачи информационных данных по отношению к управляющим.

Синхронизация, основанная на прозрачности. За счет введения прозрачности данных сообщения по отношению к заголовку / концевику приемная сторона на основе избыточности при отсутствии ошибок всегда может определить начало и конец сообщения.

Синхронизация, основанная на паузе. На канальном уровне протоколов в широковещательных локальных сетях пауза между кадрами используется для синхронизации – приемник определяет начало кадра по паузе, следующей за ней цепочки нулевых синхронизирующих битов и стартового единичного бита.

Последовательная нумерация данных в потоке

Если последовательно нумеровать передаваемые сообщения, то можно гарантированно восстановить переданную последовательность при потере отдельных сообщений, используя механизм нумерованных подтверждений и тайм-аутов. Сущность его такова:

- последовательность сообщений нумеруется, на передающем и приемном концах имеются переменные: у передатчика – номер очередного передаваемого **ns,** у приемника – номер ожидаемого сообщения **nr**, первоначально они обнуляются или согласованно устанавливаются;
- передаваемые сообщения получают текущее значение **ns,** которое увеличивается на единицу после присваивания очередному сообщению, переданные сообщения сохраняются;
- если приемник получает очередное сообщение с номером, совпадающим с **nr**, т. е. очередное ожидаемое, он высылает положительное подтверждение с номером **nr** и увеличивает **nr** на единицу;
- если номер принятого не совпадает с **nr**, т. е. происходит нарушение последовательности, то оно выбрасывается, а высылается отрицательное подтверждение опять же с номером **nr**, т. е. с ожидаемым номером;
- передатчик, получив положительное подтверждение с номером **nr**, удаляет все сообщения до номера **nr** включительно;
- передатчик, получив отрицательное подтверждение с номером **nr**, удаляет все сообщения до номера **nr** и передает оставшиеся;
- от начала передачи сообщения устанавливается тайм-аут, он сбрасывается и переустанавливается при приеме подтверждений, а по истечении тайм-аута передаются все неподтвержденные сообщения;
- чтобы не загружать канал потоком подтверждений, приемник может выслать подтверждение на последовательность принятых сообщений, выдав подтверждение только на последнее. Чтобы это не воспринималось как потеря сообщения, на этот процесс также можно установить тайм-аут: отслеживать интервал поступления сообщений и при его превышении выдавать подтверждение на последнее.

При потере единственного сообщения повторно передаются все последующие, метод «не латает дырки» в принимаемой последовательности. Это приемлемо

при низкой вероятности ошибок, а более сложные процедуры восстановления потребовали бы новых управляющих сообщений и усложнили бы алгоритм.

На самом деле уникальность номера сообщения совсем не обязательна. Размерность последовательного номера может быть ограничена количеством переданных, но неподтвержденных сообщений. Если передающая сторона ограничивает это количество величиной N и тормозит передающий процесс, если он пытается превысить этот лимит, то сообщения можно нумеровать циклически по модулю N. Если в протоколе в процессе передачи допускается не более одного сообщения, то их номера и номера подтверждений могут кодироваться логической переменной.

Протокол TCP оригинален тем, что в нем отсутствует привязка сообщений прикладного уровня к пакетам сетевого. Между сообщениями и пакетами находятся два циклических буфера: один – на поток передачи, другой – на поток приема. В буфер передачи транспортный уровень записывает сообщения, а сеть «нарезает» получаемые из него данные на пакеты. В связи с этим TCP использует *сквозную последовательную нумерацию физических байтов в потоке,* помещаемых в буфер. Начальный номер первого байта выбирается случайным образом при установлении соединения и известен приемной и передающей стороне. При нарезке содержимого буфера на пакеты каждый пакет содержит в себе последовательный номер первого байта данных пакета. Приемная сторона помнит последовательный номер очередного ожидаемого байта и при получении пакета с ожидаемым номером байта корректирует этот номер на длину пакета и передает подтверждение. Отрицательных подтверждений нет, вместо них повторно высылается положительное подтверждение со старым номером. Подтверждения используют для своей передачи пакеты встречного потока данных.

Управление потоком. Механизм окна

Передача сообщений без ожидания подтверждений предыдущих предполагает наличие свободной буферной памяти в приемнике. Это определяется множеством факторов, в том числе тем, насколько быстро принимающий прикладной процесс успевает вычерпывать принятые данные. При отсутствии памяти приемник может высылать отрицательное подтверждение или использовать еще один вид нумерованных подтверждений, по которым передатчик блокирует передачу на некоторое время.

Этих усложнений можно избежать, если передающая сторона будет ограничивать передачу сообщений на основании полученных данных о свободной памяти на приемной стороне. Если на приемной стороне соединение располагает фиксированной памятью в **N** единиц, то этим же количеством должно

быть ограничено максимальное число единиц данных, переданных без подтверждения. На самом деле, из этой величины вычитается объем данных, которые еще не выбраны прикладным процессом приема. Эта величина называется шириной окна, а сама свободная память – *окном.*

Передающая сторона принимает от прикладного процесса данные и преобразует их в нумерованные сообщения, сокращая размер окна. Если последний размер окна – **W**, а номер последнего подтвержденного сообщений – **nr**, то можно принимать данные и создавать нумерованные сообщения до ns < nr + W. При достижении этого значения протокольный процесс передачи блокирует прикладной процесс, а возобновляет его при получении нового значения окна от приемника.

Восстановление долгоживущих соединений

Сетевое соединение является надежным в смысле достоверности и сохранности передаваемых данных, но само может быть разрушено по внешним причинам. Это могут быть падение и перезагрузка процессов и самой операционной системы, проблемы самой сети, в конце концов, ошибки программирования. Неожиданный разрыв соединения может привести к потере ценных данных, находящихся в соединении, либо к возникновению неопределенности: выполнены ли над ними действия на другом конце соединения?

В этом случае необходимы средства для восстановления текущего состояния прикладных протокольных процессов и последовательности передаваемых данных при установлении нового соединения взамен разорванного. Для этих целей могут использоваться те же самые решения, что и в протоколах низших уровней.

Идентификация сессий. При установлении соединения протокольный процесс сервера генерирует уникальный *идентификатор сессии* и передает его процессу клиента. Серверный процесс создает необходимые структуры данных для установленной сессии и ассоциирует их с выданным идентификатором. При повторном соединении процесс клиента передает команду восстановления, в которой указывает этот идентификатор. Серверный процесс находит у себя данные сессии и привязывает к ним новое соединение. Дополнительно серверный процесс устанавливает тайм-аут восстановления, по истечении которого, если не произошло восстановления, сессия сервером закрывается.

Последовательная нумерация транзакций. Описанный выше способ нумерации сообщений может использоваться и для нумерации транзакций с целью исключения их потери. Все передаваемые команды клиентского процесса последовательно нумеруются, процесс сохраняет текущую команду до получения ответа, серверный процесс запоминает номер последней обработанной команды и на каждую из них высылает подтверждение, содержащее ее

порядковый номер, сохраняя при этом результат последней. Если разрыв соединения произошел при передаче команды, то после восстановления она повторно передается с тем же номером. Если серверный процесс не обработал команду с таким номером, он ее выполняет и высылает ответ, если уже обрабатывал – высылает фиктивное подтверждение или сохраненные данные.

Автоматные модели протоколов. Диаграмма состояний / переходов

Распространенный способ реализации протокольного процесса основан на использовании технологии *событийного программирования*. Протокольный процесс представляет собой набор процедур, вызываемых при наступлении событий. Такими событиями в протоколе являются:

- асинхронный прием данных от другого протокольного процесса;
- истечение установленного тайм-аута;
- выполнение команд процесса-клиента по отношению к протокольному процессу.

Если событийная модель не является примитивной, а требует реакции на *последовательность событий,* то описание поведения протокольного процесса можно представить в виде *конечного автомата – автоматной модели.*

Составные части автоматной модели:

- множество состояний, в каждый момент времени одно из них является текущим. Текущее состояние является в автомате единственным запоминаемым параметром;
- переход из состояния в состояние происходит по событию: событием в автоматной модели протокола может быть получение данных от другого протокольного процесса, истечение тайм-аута, команда процесса-клиента;
- при переходе из состояния в состояние автоматная модель выполняет некоторое действие, например, передает сообщение другому протокольному процессу, устанавливает тайм-аут, уведомляет процесс-клиент о принятых данных.

Таким образом, автоматная модель является удобным средством описания поведения системы, ориентированной на события, ее можно использовать как средство документирования и верификации протокола.

Автоматную модель процесса следует отличать от конечного автомата – математической модели. Математическая модель конечного автомата предназначена для доказательства справедливости некоторых утверждений относительно моделируемого объекта, а также для выполнения над ней различных формальных преобразований. Можно сказать, что автоматная модель использует не

формальную, а содержательную сторону понятия «автомат» для описания процесса. Поэтому автоматная модель допускает различные расширения, выходящие за рамки классического автомата (см. раздел 2.2 – диаграммы состояний).

Поскольку все входные события в автомате являются асинхронными, то они должны быть синхронизированы к автомату как к общему ресурсу. Обработка одного события является неделимой операцией и не может прерываться обработкой другого.

В качестве примера рассмотрим диаграмму состояний протокольного процесса, в функции которого входят установление соединения с сервером, авторизация и восстановление соединения при его разрыве (рис. 4.16).

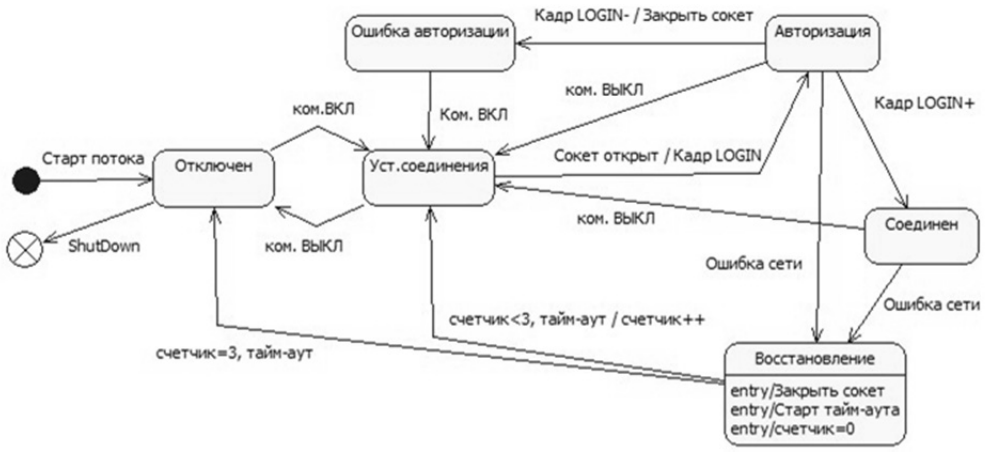


Рис. 4.16. Диаграмма состояний соединения с сервером

Для корректного управления соединением и синхронизации приема и передачи вводится множество состояний соединения, переключения между которыми производятся как по асинхронным сообщениям сервера, так и по внешним командам клиента. Протокольный процесс использует отдельный поток для установления соединения и асинхронного приема данных.

ГЛАВА 5

СУРОВАЯ ПРОГРАММНАЯ ИНЖЕНЕРИЯ. МЕТОДОЛОГИИ, МОДЕЛИ, ФРЕЙМВОРКИ, СТАНДАРТЫ

5.1. Унифицированный процесс: рабочий инструмент или недостижимый идеал

Унифицированный процесс (Unified Process – UP) – тяжеловесная методология на основе итеративной модели жизненного цикла с использованием ООП и UML [21]. **Rational Rose** – фреймворк компании IBM для поддержки вариации этой методологии – **RUP** (Rational Unified Process) [22, 25]. Трудно не заметить почти полного совпадения технологических процессов UP с соответствующими документами SWEBOK хотя бы по названию. Это вполне логично: методология, охватывающая все стороны и все виды деятельностей в жизненном цикле системы, объединяет их в технологические процессы естественным образом. Поэтому UP может служить недостижимым идеалом любой другой методологии. Если описанная в нем деятельность в другой методологии отсутствует, это означает, что:

- она не нужна в проекте данного масштаба или из-за его специфики;
- она реализуется неформально и не документируется.

Нет смысла воспроизводить здесь подробное содержание UP. Все нижеследующее – конспективное изложение UP, отдельные технологические процессы жизненного цикла рассматриваются в последующих главах. Здесь же дается:

- собственно представление о методологии;
- терминология жизненного цикла. Она довольно объемна, в разных методологиях используются различные вариации терминов. Здесь термины UP выделены полужирным шрифтом, синонимы, оригиналы терминов на английском и варианты их перевода – курсивом.

Структура UP

Базовые элементы UP, из которых состоит процесс:

- **роли** – исполнение участником процесса ряда взаимосвязанных деятельностей обычно в рамках одного технлогического процесса, реже – в рамках нескольких;
- **деятельности** (*activity*) – элементарные работы;
- **артефакты** – рабочие продукты процесса: модели, документы, компоненты программного кода в исходном и скомпилированном виде, библиотеки, конфигурационные файлы, сборки, релизы и т. п.;
- **технологические процессы** (*дисциплины*) – совокупность связанных между собой деятельностей, имеющих общую цель, результаты и артефакты;
- **фазы** (*этапы*) – самый крупный временной отрезок жизненного цикла, в результате которого достигается уникальное качественное состояние разработки;
- **итерации –** временной отрезок процесса, завершающийся созданием согласованного набора артефактов. Каждый этап состоит из последовательности итераций.

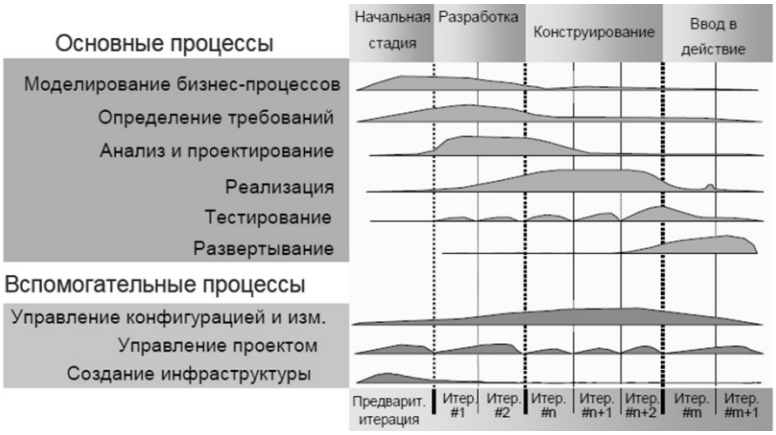


Рис. 5.1. Этапы и технологические процессы в UP

Фазы жизненного цикла и результаты (рис. 5.1):

- **исследование** (в оригинале **inception** – начало) завершается принятием решения о разработке проекта, основным результатом является документ *видения проекта.* Все деятельности направлены на получение информации, необходимой для принятия такого решения на основании полученных оценок характеристик будущего проекта;

- **развитие** (в оригинале **elaboration** – разработка, развитие, уточнение, в оригинале перевода – *уточнение плана*) представляет собой «теоретическую» проработку будущей системы во всех ее аспектах. Завершается созданием *архитектурного прототипа*;
- **построение** (в оригинале **construction** – построение) – непосредственная реализация системы в действующих артефактах, завершается созданием бета-версии продукта;
- **развертывание** (в оригинале **transition** – перемещение, переход) – перенос, установка системы, распространение и сопровождение.

Технологические процессы жизненного цикла:

- **моделирование производства** – бизнес-аналитика;
- **управление требованиями** – системная аналитика;
- **анализ и проектирование** – функциональное, архитектурное и детальное проектирование;
- **реализация** – конструирование, кодирование (кодинг);
- **тестирование** – верификация и испытание всех артефактов системы, в том числе программного кода;
- **развертывание** – распространение, установка, обучение.

Вспомогательные процессы:

- **управление проектом** – организация, менеджмент;
- **управление конфигурацией** – сборка системы из различных версий артефактов, управление изменениями, сопровождение;
- **управление средой разработки**.

Основные артефакты UP изображены на рис. 5.2.

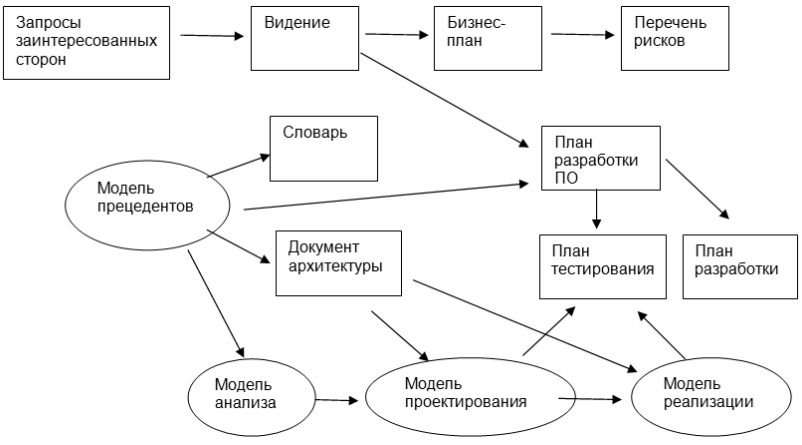


Рис. 5.2. Основные артефакты UP

Фаза исследования проекта

Цель фазы исследования – сбор и анализ информации, необходимой для старта разработки, оценка характеристик будущей системы.

Результаты фазы исследования:

- область применимости, граничные условия (для документа видения проекта);
- основные и критические прецеденты системы;
- план возможной архитектуры;
- предварительная оценка стоимости;
- предварительная оценка рисков.

Деятельности фазы исследования:

- формулировка *области действия проекта, рамкок проекта* – важнейшие требования, ограничения, среда исполнения;
- разработка *бизнес-плана* – оценка рисков, кадровое обеспечение, цена, график работ, рентабельность;
- разработка возможной *архитектуры*.

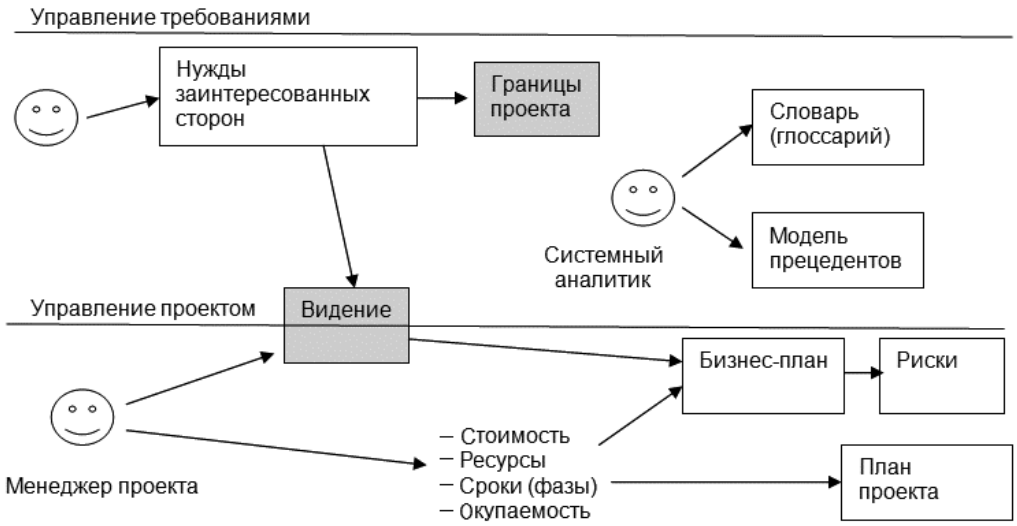


Рис. 5.3. Артефакты фазы исследования

Артефакты фазы исследования (рис. 5.3):

- документ видения;
- обзор модели прецедентов;
- начальный глоссарий проекта;

- начальный бизнес-план;
- начальная оценка рисков;
- план проекта – фазы и итерации.

Фаза развития

Цель фазы развития – получение полного и согласованного набора описательных и исполнимых артефактов, необходимых для реализации системы. Проработка и принятие окончательных решений относительно фаз построения и развертывания.

Результаты фазы развития:

- базовая линия архитектуры;
- базовая линия видения;
- базовая линия точного плана построения;
- обоснование того, что базовая архитектура позволяет получить требуемое видение за разумную цену, сроки и с разумными рисками;
- выполнимый архитектурный прототип.

Деятельности фазы развития:

- детальная проработка видения;
- разработка критических прецедентов;
- проработка технологического процесса среды разработки;
- подробная разработка архитектуры и выбор компонентов. Создание, покупка и повторное использование компонентов, разработка графика фазы построения.

Артефакты фазы развития:

- модели прецедентов, охватывающие 80 % функционала;
- дополнительные спецификации на нефункциональные требования;
- описание архитектуры;
- выполнимый архитектурный прототип;
- скорректированный перечень рисков и бизнес-план;
- план проекта, итераций и критерии оценки итераций.

Все деятельности по созданию артефактов системы в UP подробно расписаны. Например, так выглядит в каждой итерации процесс создания архитектурного прототипа [22]:

- *управление проектом*. Разработка плана итерации, оценка архитектурных рисков;
- *управление требованиями*. Выбор прецедентов и сценариев, направляющих разработку архитектуры. На основании этого проводят обновление плана итерации и перечня рисков;

- *управление требованиями*. Создание прототипа пользовательского интерфейса;
- *анализ и проектирование*. Выделение очевидных классов, разбиение системы на подсистемы и подробное рассмотрение прецедентов. Артефакт: документ архитектуры;
- *анализ и проектирование*. Уточнение и гомогенизация классов, определение архитектурно значимых классов;
- *анализ и проектирование*. Рассмотрение низкоуровневого разбиения на пакеты;
- *анализ и проектирование*. Согласование среды реализации, принятие решений относительно ключевых сценариев, определение формальных интерфейсов классов;
- *анализ и проектирование*. Рассмотрение параллелизма и распределения архитектуры;
- *реализация*. Модель реализации и ее физическое представление, планирование интеграции, планирование интегральных и системных испытаний, реализация классов и интеграция, интеграция реализованных подсистем;
- *тестирование*. Оценка выполнимой архитектуры.

Фаза построения

Фаза построения в содержательном плане является наиболее тривиальной. Она включает модульное конструирование и тестирование, сборку проекта, интеграционное и системное тестирование. Содержание завершающей итерации фазы построения выглядит так [22]:

- *управление проектом.* Планирование итерации;
- *реализация.* Планирование интеграции на системном уровне;
- *тестирование*. Планирование и проектирование тестирования на системном уровне;
- *анализ и проектирование.* Уточнение реализации прецедентов;
- *тестирование.* Планирование и проектирование интегральных тестов на подсистемном и системном уровнях;
- *реализация*. Разработка кодов и тестирование блоков;
- *реализация.* Планирование и реализация блочных испытаний;
- *реализация.* Тестирование блоков в подсистеме;
- *реализация и тестирование.* Интеграция подсистемы;
- *реализация.* Тестирование подсистемы;

- *реализация.* Выпуск подсистемы;
- *реализация.* Интеграция подсистемы;
- *тестирование.* Тестирование интеграции, тестирование системы.

Фаза развертывания

Деятельности фазы развертывания:

- бета-тестирование;
- параллельное функционирование с существующей системой и ее замещение;
- конвертирование данных, миграция данных;
- подготовка пользователей и персонала;
- маркетинг, распространение и продажа.

Замечание по теме. Как следует из описания фаз UP, существует четкий водораздел между фазами развития и реализации, между теоретическим проектированием в документах и моделях совместно с прототипированием на фазе развития и фактической «заливкой проекта кодом» в фазе реализации. Тем самым предполагается, что результаты фазы развития безупречны. Если вдруг в фазах реализации или развертывания возникнут непреодолимые препятствия, то откат к предыдущей фазе технологически невозможен. Здесь рассматриваемая методология обнаруживает общие черты с каскадной.

Далее рассмотрим содержание технологических процессов UP.

Технологический процесс моделирования производства

Используемый далее синоним названия процесса – **анализ предметной области**. Основная цель – разработка модели технологического процесса, подвергаемого автоматизации. Используются те же средства, что и при управлении требованиями, только модель составляется для действующего процесса. Деятельности в технологическом процессе подпадают под общий термин «*бизнес-аналитика*».

Артефакты технологического процесса:

- документ видения производства;
- модель производственных прецедентов – описание бизнес-процессов предметной области;
- модель объектов производства – диаграмма классов предметной области;
- глоссарий производства;
- оценка целевой организации;

- правила производства – бизнес-правила;
- дополнительная производственная спецификация.

Сценарии, по которым осуществляется моделирование, зависят от того, как будет меняться схема технологического процесса в результате автоматизации. Возможны следующие виды моделей:

- организационная схема. Если схема технологического процесса производства не меняется, то создается описание текущего производства, которое переносится один к одному в процесс управления требованиями;
- моделирование предметной области. Разрабатывается статическая модель данных предметной области без описания процессов, которая входит далее в модель объектов производства;
- новое производство;
- реорганизация;
- общая модель производства – усреднение моделей производства нескольких организаций при разработке общего программного проекта;
- одно производство для нескольких систем – части одного производства рассматриваются как основа для нескольких программных проектов.

Технологический процесс управления требованиями

Технологический процесс управления требованиями интегрирует все, что касается функционала разрабатываемой системы, в целостную систему артефактов и управляет ими.

Цели процесса:

- единое представление функционала системы;
- определение границ системы;
- основа для планирования содержания итераций, оценки стоимости и сроков исполнения;
- определение пользовательского интерфейса системы.

Основным элементом процесса являются требования. Требования в UP классифицируются как функциональные требования и прочие свойства – нефункциональные требования. Виды нефункциональных требований: практичность (*Usability*), надежность (*Reliability*), производительность (*Performance*), безопасность, возможность поддержки (*Supportability*) – требования к поддержке системы после ее выпуска, а также требования к среде проектирования.

UP классифицирует требования по характеристикам (приоритеты, полезность, решаемость), по источникам возникновения (запросы заинтересованных сторон, требования абстрактного вида, требования к поведению отдельных компонент, прецеденты) и т. п.

Артефакты процесса управления требованиями:

- глоссарий – словарь;
- архив прецедентов;
- прототип пользовательского интерфейса.

В технологическом процессе UP выделяет следующие деятельности: анализ проблемы, понимание нужд заинтересованных сторон, определение системы на основе набора свойств, управление масштабом (границами системы), уточнение определения системы, разработка прецедентов и пользовательского интерфейса.

Технологический процесс анализа и проектирования

В технологическом процессе создается **модель анализа** – классы анализа и подсистемы, построенные на основе модели предметной области, модели прецедентов и функциональных требований к системе. Далее она эволюционирует в **модель проектирования,** согласованную с нефункциональными требованиями, средой реализации и архитектурой. Также сюда включаются *артефакты систем реального времени и архитектуры* – шаблоны проектирования, протоколы, сигналы, события.

Деятельности и артефакты технологического процесса:

- определение потенциальной архитектуры на основе:
  - эскиза архитектуры;
  - модели классов анализа с учетом архитектурно-значимых прецедентов;
  - взаимодействия прецедентов и классов анализа, представленных с помощью диаграмм устойчивости;
- уточнение архитектуры;
- анализ поведения;
- проектирование компонентов;
- проектирование компонентов реального времени;
- проектирование баз данных.

Технологический процесс конструирования

Состоит из множества итераций, на каждой из которых интегрируются компоненты, реализованные и прошедшие блочное тестирование. Технологический процесс сборки осуществляется по цепочке: *компоненты – интеграция – подсистема реализации – интеграция – система*. UP вводит для этого процесса несколько терминов.

Конструкция – функциональная версия системы или ее части, демонстрирующая сокращенный вариант возможностей системы. Синоним: *сборка* – вариант интеграции компонентов в подсистему реализации, *сырой* вариант действующей системы на текущий момент.

Поэлементная интеграция – интеграция на каждом этапе одной новой или измененной компоненты, **поэтапная интеграция** – одновременная интеграция несколько новых или измененных компонент.

Прототип – работоспособный, но функционально неполный вариант системы или ее части, предназначенный для оценки, анализа и экспериментирования:

- поведенческий – моделирование поведения системы, например, образец интерфейса пользователя, эмулятор для анализа API сервера при отсутствии полной документации;
- структурный – заготовки компонент проектируемой системы, которые эволюционируют в итоговый проект;
- пробный – временный, используемый для текущих нужд;
- эволюционный – в дальнейшем развиваемый как часть проекта.

Артефакты процесса конструирования:

- компонента;
- подсистема реализации (сборка);
- план проведения интеграции.

Технологический процесс тестирования

Цели процесса тестирования:

- проверка взаимодействия компонентов;
- проверка правильности интеграции;
- проверка точности реализации всех требований;
- выявление дефектов и их устранение до фазы развертывания.

Классификация тестов:

- тестируемый параметр качества:
  - надежность – отсутствие обнаруженных программных ошибок;
  - функциональные возможности – соответствие прецедентам и ожидаемому поведению;
  - производительность;
- этапы тестирования:
  - блочное тестирование – модульное тестирование при конструировании компоненты;
  - интегральное тестирование составных компонент или подсистем;

- системное тестирование;
- приемочное тестирование перед развертыванием;

- типы теста:
  - аттестационный – оценка производительности в сравнении с прототипом или в виде некоторой меры;
  - конфигурационный – оценка работоспособности системы в различных конфигурациях;
  - функциональные испытания – тестирование прецедентов;
  - установочные испытания – проверка установки в различных конфигурациях с различными ресурсными ограничениями;
  - тестирование целостности – устойчивость к ошибкам времени исполнения;
  - испытание под нагрузкой – функционирование при изменении числа пользователей, транзакций, соединений – рабочей нагрузки;
  - стресс-тестирование – испытание в жестком режиме, тестирование при ограниченных ресурсах или высокой рабочей нагрузке;
  - эксплуатационные испытания – производительность в различных конфигурациях при постоянной рабочей нагрузке.

Регрессионное тестирование – выполнение всех ранее проведенных тестов на очередной версии после внесенных изменений.

Артефакты модели тестирования:

- контрольная задача – набор тестовых данных, условий выполнения и ожидаемый результат;
- методика испытаний – описание набора контрольных задач, их настройки, выполнения и оценки результатов;
- сценарий испытаний – компонента автоматизации методики испытаний;
- классы и компоненты испытаний – программные компоненты системы тестирования;
- описание программных средств тестирования в виде диаграмм последовательностей или диаграмм взаимодействия.

Артефакты тестирования:

- модель тестирования;
- план тестирования;
- результаты тестирования и данные, собранные в процессе тестирования;
- модель рабочей нагрузки для эксплуатационных испытаний;
- дефекты – запросы на внесение изменений для процесса управления конфигурацией;
- классы и пакеты тестов, подсистемы и компоненты тестирования.

Технологический процесс развертывания

Деятельности процесса развертывания:

- бета-тестирование в целевой операционной среде;
- оформление ПО;
- распространение;
- установка;
- обучение конечных пользователей, продавцов;
- замещение существующего ПО, конвертирование БД.

Типы развертывания:

- развертывание заказных систем;
- развертывание из архива с помощью программы установки;
- загрузка из Интернета.

Артефакты развертывания:

- исполняемое ПО;
- артефакты установки – сценарии, инструментальные средства, файлы, руководства;
- материалы поддержки – руководство пользователя и руководство по эксплуатации;
- обучающие материалы.

Технологический процесс развертывания включает следующие деятельности:

- планирование распространения;
- разработка материалов поддержки;
- тестирование продукта в среде разработчика;
- создание версии;
- бета-тестирование версии;
- тестирование продукта в целевой системе;
- оформление продукта;
- обеспечение доступа к узлу загрузки.

Технологический процесс управления конфигурациями

Цель процесса управления конфигурацией – обеспечение целостности проекта в условиях его непрерывного развития и изменения. Он учитывает три фактора изменений в виде трехмерной модели – **куб управления конфигурациями**.

Управление конфигурацией рассчитано на три уровня артефактов конфигурации – компоненты, подсистемы, система. Каждый артефакт существует в виде дерева версии. Основные задачи: отслеживание зависимости между артефактами, обеспечение целостности системы артефактов и правильности сборки конфигурации из различных версий.

Управление запросами на внесение изменений поддерживает технологическую цепочку внесения изменения в проект, включающую несколько технологических процессов: *управление требованиями, анализ и проектирование, конструирование, тестирование.* Включает деятельности по классификации изменений, отслеживанию состояния, накоплению связанных данных и артефактов.

Управление состоянием и изменениями затрагивает также процесс *управления проектом* – сбор и анализ метрических характеристик процесса внесения изменений: прогресс проекта относительно изменений, количество изменений в различных состояниях, возраст изменений, распределение изменений по серьезности, приоритету, влиянии на архитектуру, причинам появления.

Артефакты процесса управления конфигурацией:

- план управления конфигурацией;
- запросы на внесение изменений;
- модель реализации;
- метрики и отчеты.

Технологический процесс управления проектом

Деятельности процесса управления проектом:

- планирование жизненного цикла и последовательности итераций;
- управление рисками;
- наблюдение за прогрессом проекта, сбор метрик и их оценка;
- подбор команды;
- стратегическое планирование: крупнозернистый план фаз – продолжительность основных фаз и итераций и их результаты;
- оперативное планирование текущей и следующей итерации с использованием сетевых графиков, диаграмм Ганта и т. п.

В процесс управления проектом не входят управление персоналом, наем, обучение, тренировка, управление контрактами, управление ресурсами. Они являются частью общего менеджмента и не рассматриваются как проектные деятельности.

5.2. Гибкое и экстремальное программирование. Scrum

Гибкие методологии разработки

Опубликованный в 2001 г. «Манифест гибкой методологии разработки программного обеспечения» отразил тот факт, что тяжеловесность методологии проекта не является гарантией его успешности. Основные идеи манифеста:

- люди и взаимодействие важнее процессов и инструментов;
- работающий продукт важнее исчерпывающей документации;
- сотрудничество с заказчиком важнее согласования условий контракта;
- готовность к изменениям важнее следования первоначальному плану.

В разделе 1.1 уже обсуждалась необходимость документирования программного кода, а также его потенциальные возможности служить документом разработки и сопровождения.

Продуктивная, а не анархически понимаемая сущность гибкой методологии состоит в том, что формализация не отвергается, а признается необходимой только там, где не работают механизмы самоорганизации и неформальные методы.

Идеи гибкой (*agile*) методологии разработки [23] воплощаются в следующих основных принципах:

- ранние сроки поставки программного обеспечения – прототипа, предварительной версии;
- вариативность требований на любом этапе разработки;
- частая поставка рабочего программного обеспечения – релизов;
- постоянное взаимодействие заказчика и пользователей с разработчиками на протяжении всего проекта;
- мотивация коллектива на успех проекта, создание условий для эффективной работы;
- прямое взаимодействие исполнителей, минимизация формальных документированных отношений;
- инкремент функциональности в работающем ПО как измеритель прогресса разработки;
- спонсоры, разработчики и пользователи должны иметь возможность поддерживать постоянный темп на неопределенный срок;
- постоянное внимание улучшению технического мастерства и удобному дизайну;
- простота, искусство не делать лишней работы;

- самоорганизация команды как основа эффективных решений;
- гибкость, адаптация к изменяющимся обстоятельствам.

На базе принципов гибкого проектирования работает множество методологий с различным наполнением:

- набор практик и рекомендаций в русле гибкой методологии – экстремальное программирование [26];
- синтез идей гибкой методологии и унифицированного процесса, стремление сохранить строгость UP, придав ему необходимую гибкость – AgileUP [33];
- сбалансированный фреймворк с прописанным набором артефактов, ролей и процессов – Scrum.

Кроме того, в гибкой методологии присутствует много средств и приемов повышения мотивации, тренингов коллективного труда и самоорганизации процесса, которые здесь не рассматриваются.

Экстремальное программирование

Экстремальное программирование (XP, e**X**treme **P**rograming) представляет собой набор практик разработки кода, которые органически вписываются в идеи гибкого программирования:

- короткий цикл обратной связи:
  - разработка через тестирование – практика создания тестового покрытия под интерфейсы неработающего кода с последующей реализацией последнего в процессе рефакторинга;
  - игра в планирование – практики активного коллективного определения трудоемкости и других метрик проекта;
  - постоянное присутствие заказчика – взаимодействие с заказчиком и пользователями проекта в процессе определения и демонстрации его функциональности;
  - парное программирование – инспекция, верификация и сопровождение чужого программного кода в паре с программистом-автором при постоянной смене ролей;
- непрерывный, а не пакетный процесс:
  - непрерывная интеграция – максимально быстрая интеграция вносимых изменений в проект в целом;
  - рефакторинг – постоянное улучшение качества кода при сохраняемых интерфейсах и функциональности;
  - частые небольшие релизы – сокращение длительности итерации до минимально возможной;

- понимание, разделяемое всеми:
  - простота;
  - метафора системы – понимание функционала и архитектуры системы на основе моделей-аналогов;
  - коллективное владение кодом, использование шаблонов проектирования – понимание структуры кода и соблюдение стандартов кодирования до такого уровня, когда каждый член команды может вносить изменения в любую часть кода независимо от его авторской принадлежности;
  - стандарт кодирования – необходимое условие для предыдущего пункта;
- социальная защищенность программиста – 40-часовая рабочая неделя, качественное планирование процесса, отсутствие авралов.

Scrum

Scrum (скрам, англ. *scrum* – схватка) – популярный фреймворк на основе гибкой итеративной методологии с довольно глубоким охватом процессов жизненного цикла и оригинальными решениями по планированию и управлению проектом [47]. В соответствии с его структурой в нем присутствуют следующие принципы экстремального программирования (рис. 5.4):

- игра в планирование в части оценки затрат;
- постоянный контакт с заказчиком;
- коллективное владение кодом;
- стандарты кодирования.

Список действующих лиц (ролей) в Scrum минимален:

- владелец продукта (Product owner, PO) – ответственный за формирование требований и их приоритетов, осуществляет планирование и управление проектом со стороны функционала;
- скрам-мастер (Scrum Master, SM) – лидер команды, технический руководитель, отвечает за процессы разработки, координацию работы команды, поддержание социальной атмосферы и решает организационные вопросы – планирование спринта, проведение скрам-митингов, демонстраций;
- команда исполнителей – 7 ± 2 человек.
- возможный системный аналитик для сложных проектов, который берет на себя часть работы у владельца продукта.

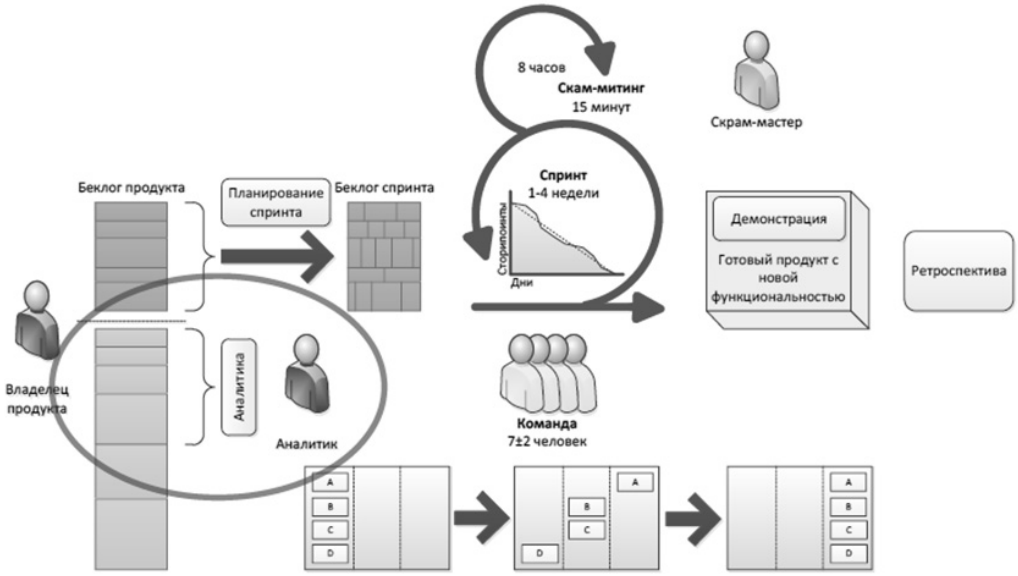


Рис. 5.4. Структура процесса разработки в Scrum [47]

Из количественного состава команды следует, что Scrum ориентирован на разработку проектов размером не выше среднего. Для более крупных проектов Scrum масштабируется. Ключевая фигура в Scrum – владелец проекта. Он совмещает несколько ролей и дисциплин в терминологии унифицированного процесса:

- бизнес-аналитика и системная аналитика: анализ предметной области, управление требованиями;
- управление проектом: планирование и метрика проекта.

Технологический процесс Scrum включает следующие деятельности:

- **спринт** – итерация продолжительностью от двух недель до месяца, по окончании которой создается очередной релиз или прототип проекта, одновременно он является единицей планирования проекта;
- планирование спринта – наполнение спринта задачами (истории пользователей), оценка их приоритетов и трудоемкости;
- **скрам-митинг** – ежедневная короткая планерка для обсуждения командой текущих технологических и технических вопросов по выполняемым задачам;
- **обзор спринта (демонстрация)** – обсуждение и анализ очередного релиза по окончании спринта, содержательное планирование следующей итерации с участием владельца продукта и всей команды;
- **ретроспектива** – общее собрание по результатам нескольких итераций для обсуждения глобальных перспектив и результатов проекта.

Основные артефакты Scrum:

- единица исполняемой членами команды работы, в терминах Scrum – **история пользователя (User Story)**;
- **бэклог продукта (Product Backlog)** – список текущих и планируемых задач;
- **бэклог спринта (Sprint Backlog)** – часть бэклога продукта с самой высокой важностью и суммарной оценкой, не превышающей производительность команды, отобранная для спринта;
- **инкремент продукта** – функциональность продукта, созданная во время спринта.

Планирование процесса разработки

Единица исполняемой работы в Scrum носит название *история пользователя* (*UserStory*). Это одновременно основной элемент планирования и единица функционала, допускающая непосредственную демонстрацию. Возможность продемонстрировать результат в очередном обзоре спринта и ориентация на функциональность являются принципиальными: без демонстрации задача считается невыполненной полностью и не учитывается в метриках спринта. Законченная функциональность – основа тесного взаимодействия с заказчиком.

История пользователя реализуется одним программистом за исключением случаев, когда она затрагивает несколько подсистем. В этом случае он разбивается на **задачи**, которые все равно планируются и оцениваются как единое целое.

Ориентация на функциональность является одновременно сильной и слабой стороной Scrum и гибкой разработки в целом. Архитектурные составляющие системы – слои, службы, сервисы, API, которые обеспечивают работу системы и не привязаны к конкретному функционалу, зачастую невозможно продемонстрировать или хотя бы объяснить заказчику. Да и какое они имеют к нему отношение?

Частным решением проблемы является введение **технических историй** для разработки общесистемных компонент и планирования деятельностей из области управления конфигурацией и средой разработки, например:

- разработка архитектурного прототипа или прототипа подсистемы;
- рефакторинг или реинжиниринг подсистемы;
- общий рефакторинг;
- оптимизация;
- решение задач производительности и масштабируемости;
- исправление сложного дефекта;
- изменение инфраструктуры – конвертирование БД, «переезд» сервера.

Для оценки уровня выполнения историй / задач вводится система их состояний – *планирование, аналитика, разработка, тестирование, готовность***.** Для каждого состояния устанавливается соответствующий процент накопления трудоемкости, одинаковый для всех историй, например, 0, 20, 60, 80, 100 %. Это позволяет выполнять общий мониторинг производительности команды. Для коммуникаций внутри команды обычно используется **StoryMapping** – визуализация текущего набора историй/задач и их состояний на доске со стикерами.

Для мониторинга и планирования история снабжается необходимыми параметрами:

- уникальным числовым идентификатором истории;
- названием, коротким описанием функционала с точки зрения пользователя в виде тройки «роль–действие–цель»;
- подробным описанием, сценарием;
- важностью – числовым приоритетом истории;
- оценкой – числовой относительной оценкой затрат на реализацию, выраженной в *StopyPoint*;
- демонстрацией – тестовым сценарием по окончании спринта;
- категорией – обычная, техническая;
- компонентами и подсистемами, затронутыми реализацией. Если их несколько, то история разбивается на соответствующие задачи;
- инициатором.

Основанием для установления важности истории пользователя могут быть как потребительские свойства функционала, так и обстоятельства текущего момента эксплуатации системы.

Метрика и оценка трудоемкости

Планирование и метрики производительности труда программиста привязаны к историям пользователя. Вводится условная единица измерения – **StoryPoint,** в которой оценивается функциональная сложность реализации истории. Оценка трудоемкости производится в процессе *итеративного сведения к консенсусу*, в терминологии Scrum – **покер-планирования**, которое выглядит следующим образом:

- у членов команды есть набор карточек с весами (значениями *Story Point*);
- владелец продукта описывает задачу настолько подробно, чтобы были понятны детали ее реализации;
- члены команды обдумывают решение, не совещаясь друг с другом, и одновременным открытием карт оценивают ее трудоемкость;

- затем заслушиваются доводы выставивших минимальную и максимальную оценки;
- процесс повторяется, пока оценки не сойдутся достаточно близко.

Следует отметить, что функциональная оценка не является ни строго качественной, ни строго количественной. Для исполнителя линейная шкала излишне подробная и необоснованная (15 или 17 единиц), а логарифмическая – слишком грубая. Поэтому выбирают некоторую последовательность с промежуточной степенью роста, например, числа Фибоначчи: 1, 2, 3, 5, 8, 13, 21, 34, где каждое последующее равно сумме двух предыдущих. Кроме этого, вводятся оценки «0» – *элементарно* и «затрудняюсь» – *отказ*.

Замечание по теме. Почему оценка трудозатрат не производится непосредственно в человеко-часах? На это есть свои резоны:

- исключается субъективный момент, связанный с разной производительностью программистов, исполнитель думает «не за себя, а за задачу»;
- реальная производительность команды, определяемая за предыдущие спринты, рассчитывается как сумма значений *StoryPoint* выполненных и продемонстрированных задач. В результате получается вес одного *StoryPoint* как частное от деления затраченного времени в человеко-часах на эту величину. Наполнение последующего беклога производится исходя из обратного подсчета: фонд времени спринта делится на вес *StoryPoint*, полученного на предыдущей итерации. Таким образом, удается сохранить за *StoryPoint* абстрактный показатель сложности и итеративно привязать его к реальной производительности команды.

Планирование и метрика процесса (бэклог спринта)

Единицей планирования в Scrum является спринт, а единицей работы – *UserStory* с оценкой трудоемкости, выраженной количеством *StoryPoint*. Процедура наполнения спринта выглядит в этих рамках вполне естественно:

- спринт имеет определенную сформулированную цель;
- длительность спринта варьируется по времени, необходимому для достижения цели спринта. Она определяет объем спринта в *StoryPoint* на основе веса *StoryPoint*, оцененного в предыдущих спринтах и объема спринта в часах;
- веса отдельных историй оцениваются командой с помощью процедуры покер-планирования;
- бэклог спринта наполняется собственником продукта в соответствии с важностью историй;
- существует возможность разбиения большой истории на части для оптимизации бэклога спринта либо объединения нескольких мелких.

Наличие весов историй, а также фиксация этапов их исполнения соответствующими процентами веса позволяет подсчитать сумму весов невыполненной части бэклога и построить для этого значения **диаграмму сгорания спринта** (рис. 5.5) во времени.

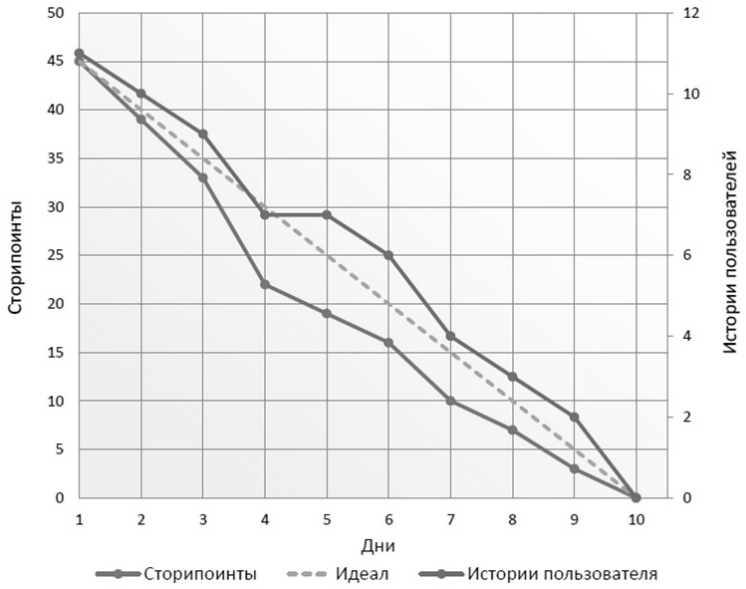


Рис. 5.5. Диаграмма сгорания спринта [47]

Диаграмма сгорания позволяет оценивать реальную производительность относительно прогноза и прогнозировать перспективы завершения спринта.

Стабильность оценок производительности команды определяется следующими правилами:

- вес истории во время исполнения спринта не меняется, даже если во время исполнения реальная трудоемкость оказалось существенно отличной от оцененной;
- реальная производительность оценивается по законченному функционалу, незавершенные истории к моменту окончания спринта не учитываются в общей производительности полностью.

Бизнес-аналитика, системная аналитика и архитектура

Слабостью Scrum, как и экстремального программирования в целом, является неопределенность этапов исследования, анализа и архитектурного проектирования: процесс накопления функционала, ориентированный на пользова-

теля, не касается эволюции архитектуры. Указанные пробелы отчасти восполняет **нулевой спринт** – предварительную итерацию с привлечением бизнес-аналитика для проведения предварительной проектной работы, аналогичной первым двум фазам UP. Нулевой спринт включает:

- разработку видения проекта – бизнес-модель, риски, способы монетизации, оценка сроков и трудоемкости;
- моделирование предметной области;
- анализ требований и разработку модели прецедентов;
- создание проекта архитектуры;
- планирование первого спринта.

В проектах со сложной функциональной составляющей может вводиться отдельная роль системного аналитика (см. рис. 5.4): он не заменяет владельца проекта в функциях взаимодействия с командой, но на него возлагаются задачи углубленного анализа и перспективного планирования, которое в формате Scrum выглядит как планирование *на один спринт вперед.* Оно не столько замещает владельца проекта, сколько позволяет увидеть цели в более глубокой перспективе.

Масштабирование Scrum

При использовании методологии Scrum для более крупных проектов команда сохраняется как структурная и административная единица. Принцип объединения команд называется *Scrum of Scrum*. При этом от проекта требуется такая степень модульности, чтобы доля необходимых оперативных коммуникаций между отдельными командами была невысока. Обычно команды занимаются целостными функциональными компонентами проекта, при этом необходимость согласований существует только на уровне скрам-мастеров и владельцев проекта:

- взаимодействие в исполнительской части проекта реализуется через совещания (митинги) скрам-мастеров, проводимые техническим руководителем проекта;
- взаимодействие в части функциональности проекта осуществляется объединением владельцев проектов в отдельную команду, в которой обсуждение и планирование функциональности ведется на уровне беклогов в целом, иногда с созданием общего бэклога.

ГЛАВА 6

ПРОЦЕССЫ, ДЕЯТЕЛЬНОСТИ И АРТЕФАКТЫ ЖИЗНЕННОГО ЦИКЛА

6.1. Системная инженерия

Система – общность хиппи в СССР как неформальное молодежное объединение.

Ф.И. Рожанский. Сленг хиппи

Не поминай имени Господа Бога твоего всуе.

Третья заповедь Ветхого Завета

Слово «система» является настолько общим местом, что предметом системной инженерии становятся не все системы по определению, а только обладающие следующими свойствами:

- сложностью, большим количеством составляющих элементов порядка $10^6 \ldots 10^9$ и выше, несколькими уровнями иерархии;
- междициплинароностью изучения и описания: система не может быть описана в рамках одной дисциплины и ее законов;
- целостностью – система воспринимается и существует как единое целое, а не как простой набор элементов, демонстрирующих взаимосвязанное поведение;
- множественностью представлений – система имеет различные целостные *представления* – аспекты или виды с точки зрения разных групп лиц, имеющих к ней отношение.

В качестве примера рассмотрим наиболее сложные известные системы – живые организмы и программно-аппаратные компьютерные системы. Здесь важен не только количественный фактор, но и наличие примерно одинакового числа уровней иерархии. Для программной системы это:

1) среда функционирования: твердотельные молекулярные структуры, полупроводники;

2) базовые элементы: транзистор, вентиль;

3) схемотехника: комбинационные схемы, триггеры, счетчики, регистры;

4) микроархитектура: функциональные узлы, автоматы, операции, микропрограммирование;

5) макроархитектура: управление памятью, кэш, контроллеры, конвейеры;

6) аппаратная архитектура: система команд, ассемблер;

7) среда исполнения: языки программирования, трансляторы, средства сборки и исполнения, операционные системы;

8) программная единица: ООП, классы, паттерны, конструирование, технологии программирования;

9) программный комплекс: архитектура ПО, архитектурные стили, слои, подсистемы, взаимодействие, протоколы, функциональное и бизнес-проектирование.

Аппаратная и программная составляющие имеют сопоставимую сложность [51]. Проведем оценку иерархии программной системы:

- большинство уровней создается на основе предыдущего по принципу количественной композиции элементов (*состоит из*...), отдельные из них определяются как функционал предыдущего, например, уровень 6;
- каждый уровень базируется на собственной области знаний и технологиях;
- каждый уровень может быть оценен в $10^2 \ldots 10^3$ элементов предыдущего, т. е. $10^6 \ldots 10^9$ на аппаратные средства в количестве вентилей, и столько же – на программное обеспечение в количестве команд, строк исходного кода;
- на верхних уровнях ПО отсутствуют формальные методы создания системы под конкретную задачу, их заменяют технологические приемы и тестирование.

Высшие живые организмы состоят из подсистем. В качестве примера рассмотрим уровни иерархии нервной системы человека [52] и изучающие их научные дисциплины. Попытаемся сопоставить уровни с предыдущей системой. В скобках указаны отрасли науки, занимающиеся соответствующей проблематикой:

1) среда функционирования: гены, белки, ферменты (*молекулярная биология, генетика*);

2) базовые элементы: нейромедиаторы, рецепторы, мембраны (*нейрофизиология*);

3) схемотехника: нейрон, синапсы, аксон, дендриты, спайки, пластичность, ионотропные, метаботропные и генетические процессы (*нейрофизиология*);

4) микроархитектура: виды нейронов, взаимодействие нейронов, нейронные ансамбли, пластичность (*нейрофизиология*);

5) макроархитектура: кора головного мозга – колонки, слои, структура других отделов нервной системы (*нейрофизиология*);

6) архитектура нервной системы: кора и ее зоны, таламус, гипокамп, полушария, механизмы обучения и памяти (*нейрофизиология, психофизиология*);

7) высшие функции нервной системы: поведение, высшая нервная деятельность (*психофизиология, психология*).

Проведем оценку сложности системы:

- верхние функциональные и нижние структурные уровни изучаются различными отраслями науки – психофизиологией, нейрофизиологией, молекулярной биологией и генетикой;
- нейрофизиология дает значительный объем фактического, экспериментально проверяемого и систематизированного материала об уровнях 2 и 3, как о нейронных структурах, так и о происходящих в них процессах. Например, одних только нейромедиаторов известно несколько тысяч видов;
- уровни 4 и 5 описаны на уровне структур, происходящие в них процессы не поняты и не имеют формального, воспроизводимого в опыте описания;
- психофизиология не имеет средств детального наблюдения за соответствующими нейродинамическими процессами, а довольствуется интегральными оценками;
- структурные и функциональные компоненты уровней связаны между собой, имеет место генезис структуры в процессе функционирования, который отсутствует в компьютерных системах. Например, синаптическая пластичность – изменение структуры связей нейронов в процессе обучения.

Приведенное сопоставление показывает, насколько важным является целостное представление о системе как при создании искусственных, так и для понимания и воспроизведения естественных систем. К примеру, попытки во всей полноте смоделировать или воспроизвести высшую нервную деятельность человека на уровне искусственного разума при современном уровне знаний можно проиллюстрировать достаточно простой аналогией. Требуется разработать компьютерную систему на следующей основе:

- целостные представления о физике полупроводников и схемотехнике;
- фрагментарные знания о микроархитектуре;
- общие сведения о всем остальном, включая внешний вид, структуру и интегральное поведение;
- отсутствие знаний о моделях, методах, системе команд, технологиях программирования.

В то же время такие дисциплины как *искусственный интеллект* и *нейронные сети* успешно развиваются на основе моделирования отдельных элементов

нервной системы и когнитивных процессов. Однако не стоит забывать об ограниченных возможностях таких моделей по сравнению с оригиналом.

Системная инженерия занимается сложными системами, создаваемыми человеком. Поскольку в настоящее время они буквально пронизаны информационными технологиями, то она по определению имеет отношение к программной инженерии.

Общность программной и системной инженерии

Системная инженерия [50] имеет много общего с программной инженерией. Схожие черты имеют методы и артефакты, различия наблюдаются в предметных областях. Можно сказать, что наиболее развитые тяжеловесные методологии программной инженерии и сам свод знаний (SWEBOK) стремятся реализовать принципы системной инженерии применительно к программной, и наоборот. Системная инженерия впитала в себя многое из программной. Рассмотрим кратко основные положения системной инженерии согласно [44].

Система – иерархическая композиция элементов – **холонов** по принципу «часть–целое». Не является системой классификация (например, периодическая система элементов Менделеева) или набор практик, рекомендаций (например, система Станиславского). Для систем справедлив принцип **эмерджентности** – целое больше суммы ее частей. Система, в свою очередь, является составной частью других систем. Системный подход классифицирует саму систему и ее окружение следующим образом:

- целевая система – собственно создаваемая система;
- операционное (эксплуатационное) окружение – системы, поддерживающие функционирование целевой системы или связанные с ней в процессе эксплуатации;
- обеспечивающие системы, создающие и поддерживающие систему в продолжение ее жизненного цикла, например, системы разработки и сопровождения ПО.

Системная инженерия занимается вопросами **онтологии** – фундаментальными сущностями. В основе системной инженерии лежат функциональные абстракции – **альфы** системной инженерии и ассоциации между ними (рис. 6.1). Альфы могут состоять из подальф, а также иметь воплощение в рабочих продуктах и артефактах. Альфа имеет *состояния*, отражающие «прогресс» соответствующей абстракции системы (ALPHA – Abstract-Level Progress Health Attribute). Известно семь основных альф системной инженерии:

- *стейкхолдеры* (заинтересованные стороны) – лица, деятельность которых может иметь отношение к системе. В программной инженерии это более широкая категория, чем пользователи программной системы, она включает разработчиков, хакеров и т. п.;
- *возможности* – обстоятельства, которые обусловливают разработку: требования пользователей, технологическое, финансовое и ресурсное обеспечение, наличие команды и ее квалификация, временные рамки и т. п.;
- *определение системы и воплощение системы* – проектное описание системы и ее материальная реализация. Для программных систем воплощение – процесс развертывания и исполнения кода системы. Определение системы, в свою очередь, состоит из трех подальф: требований, архитектуры и внеархитектурных составляющих *проекта*;
- *работы* – деятельности, технологические процессы, связанные с ними метрика, планирование и операционный менеджмент;
- *команда и технологии*.

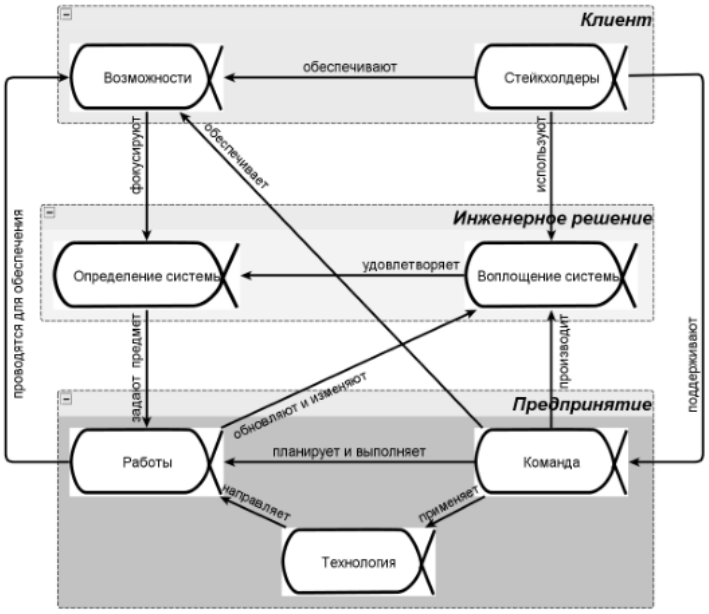


Рис. 6.1. Альфы системной инженерии [44]

Альфы, подальфы и рабочие продукты – не единственные составные элементы, которые используются для описания методологий разработки. Метамодель (модель для описания моделей) закреплена в стандарте

ISO/IEC 24744:2014. Разработка программного обеспечения. Метамодель для методик разработки». Аналогичная модель доведена до языка спецификаций в документе OMG (Object Management Group) «SMSC/15-12-02. Ядро и язык методов программной инженерии» [53]. В нем в виде диаграмм классов определяются объектно-ориентированная модель для набора сущностей системного подхода: *альф, их состояний, рабочих продуктов (артефактов), деятельностей и их пространств (видов), компетенций, практик*, *уровней детализации*, *паттернов*, а также их графическая и текстовая нотации. Фактически имеет место графическая и текстовая среда описания методологии (рис. 6.2).

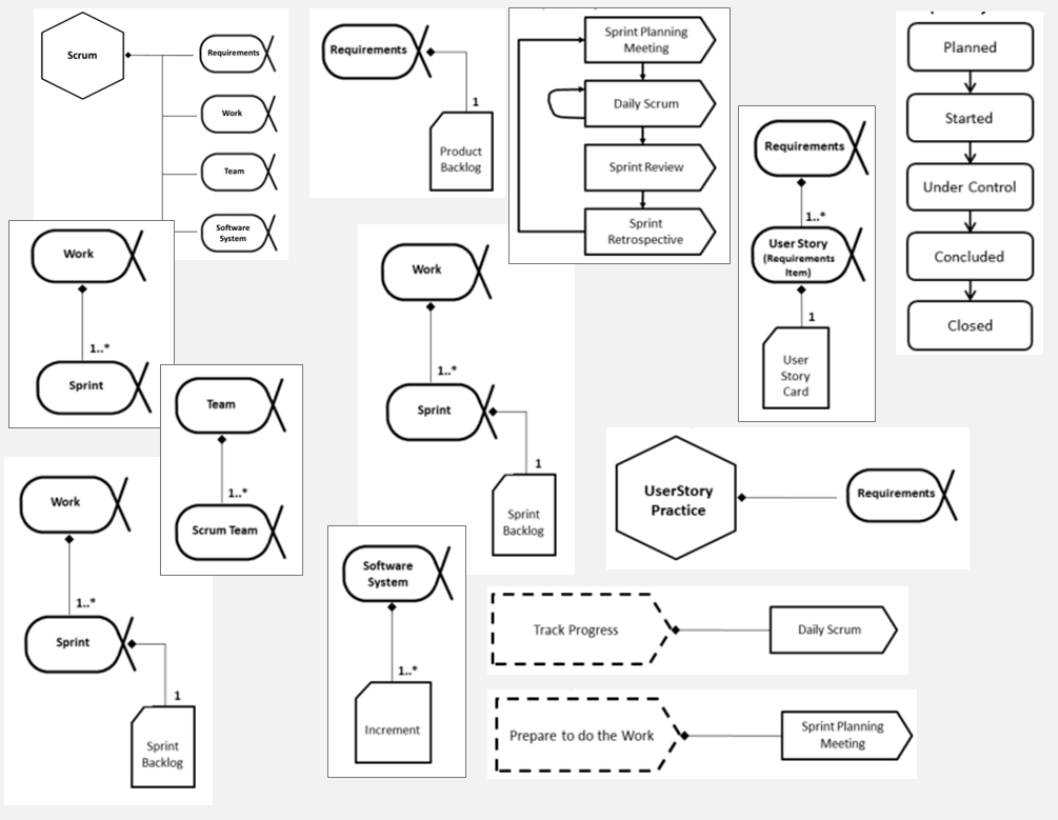


Рис. 6.2. Описание методологии Scrum в соответствии с SMSC/15-12-02

Другой принцип, который роднит программную и системную инженерию – единство представления системы через множество **срезов, аспектов** или **ипостасей.** В программной инженерии имеет место модель представления

архитектуры в пяти аспектах – «4 + 1» (см. раздел 7.1). Для технических систем используется представление в трех аспектах:

- компонента – функциональный аспект, функциональная схема, компоненты и их связи;
- модуль – продуктный аспект, физический элемент, воплощение компоненты;
- размещение – аспект места (пространственный аспект), размещение экземпляра модуля, схема размещения.

Описание системы в каждом из срезов представляет собой иерархическую структуру «целое–часть» и называется *разбиением* (breakdown). Варианты разбиения:

- функциональное разбиение (functional breakdown structure) – функциональная декомпозиция;
- разбиение работ;
- разбиение установки (размещения);
- разбиение документов.

Для технических систем с аспектами «компонент», «модуль» и «размещение» разработаны стандарты (ISO/IEC 81346) для спецификации составляющих элементов трех указанных иерархий.

Из программной инженерии унаследовано понятие «жизненный цикл системы», закрепленное в стандарте «ISO/IEC 15288–2005. Системная инженерия. Процессы жизненного цикла систем».

И, наконец, в системной инженерии определено понятие «архитектура», закрепленное в стандарте «ISO/IEC 42010:2011. Системная и программная инженерия. Описание архитектуры» [54], согласно которому архитектура определяется как «*основные понятия и свойства системы в ее окружении, воплощенные в ее элементах, отношениях и в принципах ее дизайна и эволюции*». Архитектура является абстракцией (альфой), в то время как *описание архитектуры* – рабочий продукт (артефакт). Также в контекст описания архитектуры входят сама система, окружение и стейкхолдеры, каждый из которых имеет собственные интересы в системе (рис. 6.3).

В контексте описания архитектуры стандарт фиксирует следующие положения (рис. 6.4):

- со стороны стейкхолдеров архитектура характеризуется системой интересов, по отношению к ее описанию у них имеют место *факторы беспокойства* (system concern). Это могут быть опасения относительно удовлетворения системой потребностей стейкхолдера, начиная от функциональных и заканчивая

атрибутами качества, такими как надежность, производительность, доступность. Напомним, что разработчики также являются заинтересованными сторонами с собственными системными проблемами – факторами беспокойства (доступность информации, проблемы параллелельной обработки, стоимость, график, модифицируемость, модульность, наличие тупиков, проблемы интеграции, доступность данных и пр.);

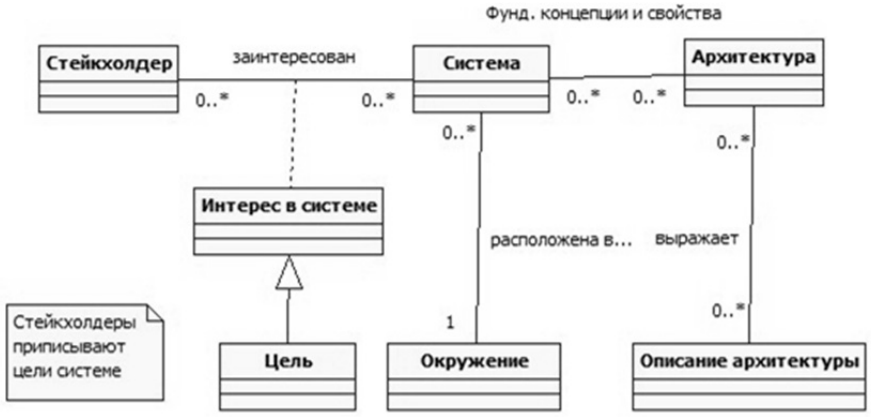


Рис. 6.3. Контекст описания архитектуры в ISO/IEC 42010:2011

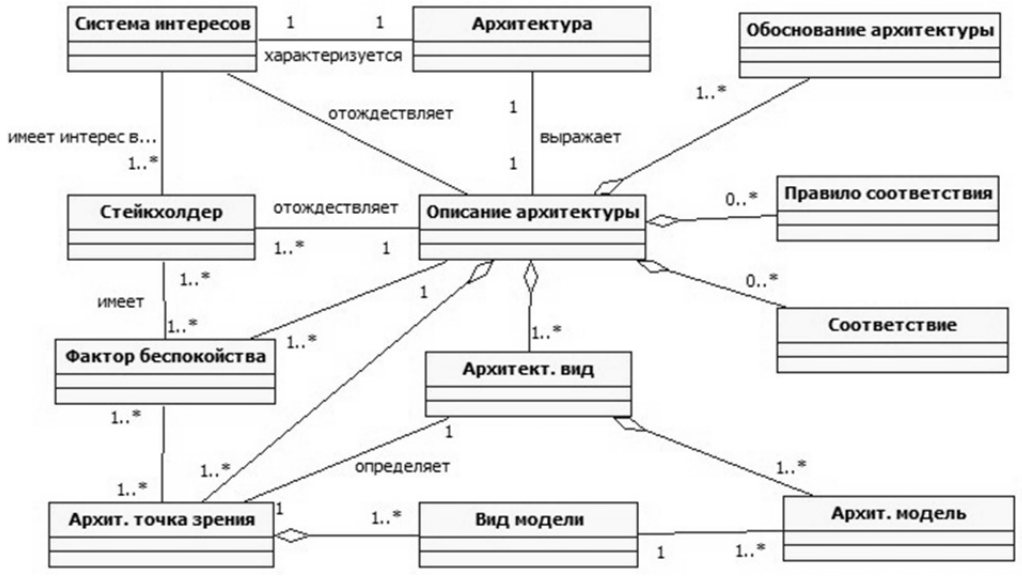


Рис. 6.4. Описание архитектуры в ISO/IEC 42010:2011

- *описание архитектуры* включает ряд обоснований, архитектурных видов и точек зрения. Архитектурный вид описывается рядом архитектурных моделей, для которых определены виды моделей, например, диаграмма классов или диаграмма состояний в UML;
- *архитектурный вид* (архитектурное представление) – рабочий продукт, выражающий отдельный аспект системы и ее архитектуры, что соответствует принципу множественности представлений системы;
- *архитектурная точка зрения* – рабочий продукт для консолидации в архитектурном представлении конкретных проблем системы – факторов беспокойства. Здесь рассматривается архитектурное представление в связи с отношением к системе стейкхолдеров, но не прямо, а через обозначаемые ими проблемы.

Следует заметить, что система имеет разнообразие архитектурных представлений в соответствии с отношением к ней заинтересованных сторон, в число которых входят и разработчики. Применительно к программной инженерии архитектурные представления – это бизнес-архитектура, системная архитектура и архитектура ПО.

С описанием архитектуры связаны *системные решения*, но не прямо, а посредством реализации в них отдельных *элементов описания* (рис. 6.5). Решения могут быть зависимы друг от друга, могут повышать одни факторы беспокойства и иметь отношение к другим. Элементы описания должны быть согласованы через соответствия, которые определяются правилами соответствия.

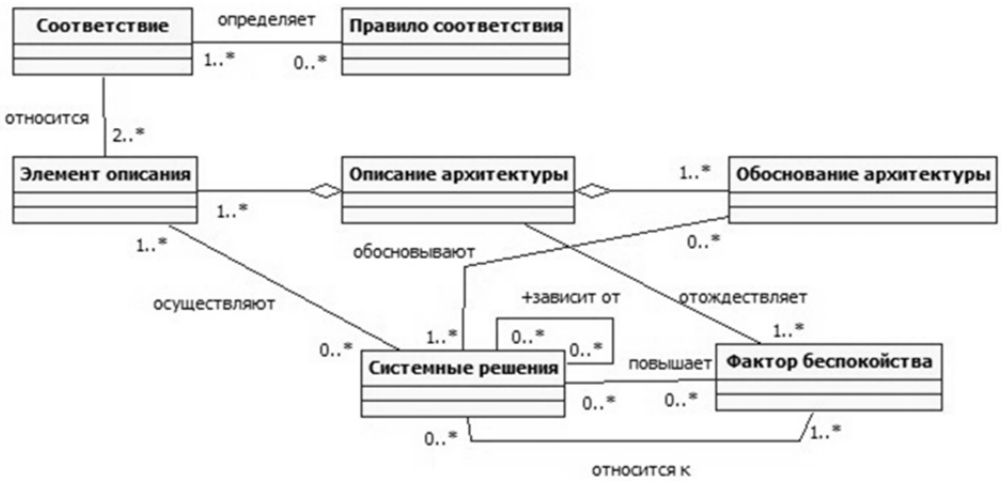


Рис. 6.5. Системные решения и обоснование архитектуры в ISO/IEC 42010:2011

И, наконец, в стандарте определены роль и место таких понятий, как «*каркас архитектуры*» (*архитектурный фреймворк*) (рис. 6.6) и «*язык описания архитектуры*» (рис. 6.7).

Рис. 6.6. Архитектурные каркасы в ISO/IEC 42010:2011

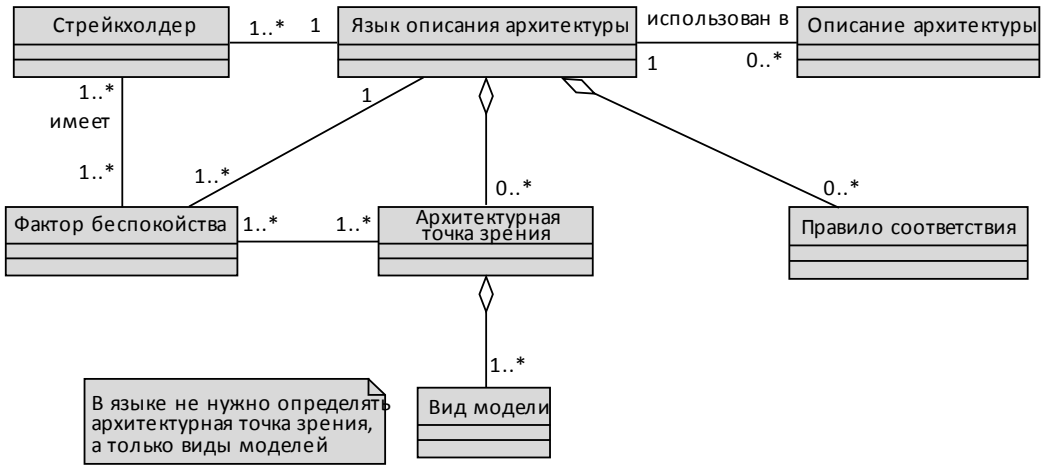


Рис. 6.7. Язык описания архитектуры в ISO/IEC 42010:2011

Замечание по теме. В технических системах существенные внешние воздействия или подсистемы логично рассматривать как стейкхолдеры с собственными факторами беспокойства и системой интересов. Например, в

шторм в буровой платформе является существенным фактором, влияющим на архитектуру системы, поэтому резонно отразить это воздействие в виде «заинтересованного лица», стремящегося повредить платформу. Важность этого воздействия заслуживает того, чтобы отразить его в соответствующем архитектурном виде, точках зрения и прочих артефактах системного подхода.

6.2. Бизнес-архитектура и бизнес-аналитика

Слово «бизнес» как первая часть составного слова в программной инженерии означает, что указанный термин имеет отношение исключительно к предметной области или ее проекции в систему, например бизнес-объект. Деятельности, прямо или косвенно связанные с предметной областью, присутствуют в разных технологических процессах и на разных этапах разработки:

- на фазе исследования это деятельности, связанные с бизнес-архитектурой – бизнес-цели проекта, обоснование его успешности, экономические, социальные и прочие аспекты разработки программных систем. Проще говоря, доказательства того, что проект «выстрелит». Для этого в UP существует отдельная роль бизнес-архитектора. Ее деятельности и артефакты включены в технологические процессы *управления требованиями* (бизнес-требования) и *управления проектом*;
- на фазах исследования и развития проекта активен технологический процесс *моделирования бизнес-процессов* (моделирование производства). В целом это называется бизнес-аналитика – описание бизнес-процессов до и после включения в них программной системы. Как и для любой динамической системы, для бизнес-процессов необходимо описание структуры (модель предметной области) и поведения (бизнес-процессы как таковые);
- в многослойной архитектуре приложения в технологическом процессе архитектурного и детального проектирования обычно создается *бизнес-слой* – слой, предназначенный для манипулирования бизнес-объектами в программной среде. Классы бизнес-объектов содержат согласованный набор методов: инициализация, утилизация, связывание с другими объектами, загрузка, сохранение, сериализация, примитивы поведения. Они являются основной компонентой для реализации функционала (см. раздел 7.2);
- в системной аналитике (технологический процесс управления требованиями) выделяются *бизнес-правила* – особый вид требований, связанный с предметной областью: факты, ограничения, условия, зависимости, которые необходимо учитывать в функционале (см. раздел 6.4).

Разработка бизнес-архитектуры тесно связана с другими деятельностями на фазе исследования, формально она имеет отношение к *управлению проектом*. Но, так как мы решили разобраться с «бизнесом» окончательно, мы будет рассматривать ее именно здесь (см. также раздел 6.7). Сюда же отнесем все деятельности в бизнес-аналитике.

Бизнес-архитектура проекта

Описание бизнес-архитектуры не нуждается в формализации. Идеи, благодаря которым проект «выстрелит», могут быть преподнесены как беллетристика, эссе, результаты социологического исследования или анализа бизнеса. Это не значит, что материал не должен быть структурирован. В отдельных разделах видения проекта должны быть отражены аспекты бизнес-архитекуры.

Пример. Бизнес-архитектура. Система выдачи ключей аудиторий

В настоящее время журналы учета ведутся не слишком аккуратно из-за большого количества записей, ключи от аудиторий на переменах передаются от преподавателя к преподавателю не всегда с фиксацией этого факта в журнале, записи делаются неразборчиво. В то же время полное следование регламенту создает очереди. Необходима автоматизация, максимально сохраняющая существующие бизнес-процессы, учитывающая квалификацию и консерватизм пользователей с минимальным набором действий со стороны основных пользователей – вахтеров.

Особенности и требования к реализации:

- для вахтера система должна быть предельно простой, с минимальной последовательностью действий и использованием привычных устройств, которые встречаются в обыденной жизни. Необходим максимально возможный уровень подсказок;
- необходимо использовать штрих-коды и оборудование для их считывания. С помощью штрих-кодов необходимо дублировать ввод всех данных и команд, т. е. штрих-коды должны быть уникальными во всех группах, например *KEY_7_211* или *TCH_BT_РомановЕЛ*. Штрих-кодами должны обладать:
  - ключи аудиторий, а лучше ячейки, где они должны находиться. Это надежнее и позволит идентифицировать ключ при его отсутствии, например, при передаче;
  - удостоверения преподавателей;
  - бейджики персонала;
  - бейджики вахтеров;

- доска меню основных команд, которые вахтер может использовать для работы с системой. Штрих-коды команд можно выводить прямо на экран монитора;
- сама вахта;

• сканер кодов желательно использовать беспроводной, так как перемещаться с ним необходимо на расстояние до нескольких метров;

• факты выдачи / приема ключей должны документироваться и подтверждаться подписью. Вместо журнала удобнее будет использовать:

- чековый принтер или любой узкий принтер с документом – чеком, на который преподаватель ставит подпись;
- обычный принтер, в который вставляется узкая лента;
- какой-либо иной способ обеспечения доверия к факту выдачи ключа;

• в угоду консерватизму пользователей бизнес-процессы автоматизированной системы должны быть максимально схожи с прототипом;

• вопросы безопасности. При вводе данных со штрих-кодов пароли не запрашиваются. Контроль осуществляется вахтером, сканирование штрих-кода персонала сопровождается выводом фотографии. Сам вахтер при регистрации в начале смены и при повторном входе вводит пароль. При отсутствии активности в течение некоторого времени система может закрываться и требовать повторной регистрации;

• решение проблемных ситуаций. Система должна не только сообщать об ошибках, но и корректировать состояние, соответствующее действительности – исправлять ситуацию, добавлять события. Например, если ключ физически присутствует, но в БД отмечен как выданный, то надо ввести фиктивную сдачу в текущее время – возможно, вахтер не отметил факт сдачи;

• вопросы надежности. Система должна обеспечивать ограниченную функциональность в различных нештатных ситуациях:

- при пропадании или отсутствии связи – накапливать данные при выполнении основного функционала, а при восстановлении связи – сбрасывать в БД сервера. Требуется иметь локальную копию части БД, касающейся текущей вахты и периодически обновлять ее;
- при отключении электропитания на резервном питании вывести документы ручного режима: ведомость выдачи с принятыми для ручного режима форматами. При восстановлении питания привести БД системы в соответствии с ведомостью, а ведомость оставить как печатный документ.

Пример. Бизнес-архитектура. Система учета рейтинга успеваемости

> Вот смотри, – сказал он. – Эта самая маленькая монетка называется сантик, а вот эта, побольше, – два сантика; … А сто сантиков составляют один фертинг.
>
> *Н. Носов. Незнайка на Луне*

Актуальность. При достаточно высоком уровне информатизации делопроизводства и учебного процесса в НГТУ контроль текущей успеваемости ведется по старинке. В то же время введение балльно-рейтинговой системы со 100-балльной шкалой требует фиксации большого объема данных о сроках и качестве выполненных работ. Общие положения о вычислении рейтинга определены в нормативных документах НГТУ, конкретные данные – единицы контроля, нормативные баллы, правила вычисления рейтинга должны быть определены в рабочих программах преподавателей.

Обоснование полезности проекта. Проект не направлен на получение коммерческой выгоды. Полезными эффектами проекта являются:

- повышение объективности оценок и оперативности контроля успеваемости;
- открытость и доступность данных о текущей успеваемости;
- возможность экспорта накопленных данных;
- повышение имиджа учебного заведения.

Основные проблемы, требующие решения:

- приемлемая формализация процесса вычисления рейтинга;
- очевидность оценки в стандартных ситуациях;
- использование мобильных клиентских приложений, в том числе и при недоступности БД – отсутствии или низком качестве сетевых соединений в аудиториях;
- ведение архива отчетных документов;
- авторизация с учетом доступа к персональным данным;
- низкая степень централизации и затрат на обслуживание.

Степень централизации и затраты на обслуживание. В целом информационная система университета является централизованной. Значительная часть функционала реализована в виде web-приложений, их сопровождение ведется *центром информатизации университета* (ЦИУ). Для реализации проекта в рамках ЦИУ существуют важные сдерживающие факторы внедрения:

- *информационная безопасность*. Обычно доступ к корпоративным ресурсам осуществляется из кабинета преподавателя, здесь же речь идет об

учебной аудитории или лаборатории, в которой есть вероятность оставить без контроля залогиненную программную компоненту;

- *объем архивных данных достаточно велик*;
- *отсутствие постоянного доступа к компьютеру.* Несмотря на тотальную компьютеризацию, возможны ситуации, когда компьютера просто нет под рукой, например, при проверке посещаемости в начале лекции, либо при приеме заданий в учебной аудитории;
- *разделение ответственности и обязательный регламент.* До сих пор ответственность за текущий документооборот по учебному процессу несли преподаватели, разделение этой ответственности с централизованными службами в массовом масштабе может породить ненужные трения и психологический дискомфорт;
- *свобода выбора преподавателя.* Нормативные документы определяют рамочные конструкции для вычисления рейтинга по основным схемам учебных дисциплин, а преподаватель в рабочей программе дисциплины наполняет их конкретным содержанием. С этой позиции введение единообразия не имеет под собой необходимых оснований.

Автоматизация сверху не смотрится по общесистемным соображениям. Она может породить неэффективного, слабо управляемого «монстра». Предлагается вариант:

- независимые БД для групп преподавателей, администрирование БД одним из преподавателей – куратором;
- централизованный хостинг БД на сервере ЦИУ или серверах подразделений.

Ведение архива отчетных документов. Необходимо сохранять текущие документы по единицам контроля – отчеты и исходные данные, например, код программного проекта или файлы собранных данных. Для целей временного хранения и систематизации имен документов в каждой БД должно быть организовано *хранилище*. Оно используется не как архив, а как средство перекачки и систематизации документов в личный архив ведущего преподавателя дисциплины.

Артефакты фазы исследования

К бизнес-архитектуре относятся артефакты фазы анализа:

- границы проекта;
- перечень заинтересованных лиц и пользователей проекта;
- глоссарий.

В глоссарий выносятся все термины, обозначающие сущности – бизнес-сущности, архитектурные и конструктивные. Сюда же могут относиться

специфические термины, касающиеся архитектуры, поведения, алгоритмов и других характеристик системы.

Недопустимо, когда в документах используются различные термины для обозначения одной и той же сущности (например, заказ, заявка, запрос) или выражения типа «список заказов в колонке слева» вместо «список необработанных заказов».

Допускается использование не принятых в данной области терминов, жаргонных терминов или метафор для обозначения сущностей, если оригинальный термин состоит из слов, употребляемых в разных контекстах, например «система», «этап». Если термин слишком длинный, то аналогично можно использовать метафору или сокращение, например, «мешок курьера» вместо «комплект заказов для перевозки курьером».

Расшифровка сущности требуется, если она не является общепринятой. Расшифровка не должна быть банальностью или тавтологией. Например, пассажир – лицо, использующее такси для поездки. Наоборот, в расшифровке должны быть указаны свойства сущности, специфической для данного проекта.

Пример. Глоссарий. Система учета рейтинга успеваемости

Предмет – оригинальная дисциплина учебного плана, читаемая в одном семестре и имеющая одну итоговую оценку.

Единица контроля – составная часть предмета, по которой выносится оценка, включаемая в рейтинг.

Рейтинг – набор данных для одной учебной группы и одной дисциплины.

Хранилище – область данных сервера для временного хранения ограниченного количества файлов отчетов и других документов (исходников) до момента их скачивания преподавателем в собственный архив.

ЦИУ – Центр информатизации университета. Сервис и данные авторизации используются для авторизации студентов и преподавателей. Для преподавателей могут вводиться оригинальные логины / пароли.

Подгруппа – выполнение некоторых видов единиц контроля происходит в двух подгруппах, для которых устанавливаются разные сроки сдачи.

Показатель качества исполнения единицы контроля – набор качественных характеристик, включаемых по принципу «да–нет», за которые снимается или начисляется фиксированный процент к нормативному баллу.

Срок сдачи устанавливается для единиц контроля, для которых предусмотрено формальное вычисление балла.

Отчет – документ, загружаемый в хранилище, отчет о выполнении единицы контроля.

Исходник – документ или архив, загружаемый в хранилище, содержит программы, данные и другие результаты работы в виде архива или отдельного файла.

Куратор – преподаватель, имеющий права на редактирование и архивирование метаданных в БД рейтингов, включая структуру предмета, списки групп и преподавателей, права доступа к данным рейтингов.

Администратор сервера БД – лицо, имеющее право создавать и уничтожать БД на сервере, а также просматривать лог-файл ошибок.

БД рейтингов – отдельная БД для нескольких предметов, студенческих групп, преподавателей и рейтингов. Создается для цикла родственных дисциплин.

Сервер БД рейтингов – сервер, хранящий несколько независимых БД рейтингов.

Преподаватель, группа, бригада, вариант задания – без комментариев.

Видение проекта

Все данные, касающиеся проекта, полученные на фазе исследования, соединяются в едином документе – **видение (концепция, устав) проекта**. Приведем основные разделы документа (курсивом выделены разделы, относящиеся к бизнес-архитектуре):

- название проекта;
- *цели проекта*;
- *приоритет проекта;*
- *результаты проекта;*
- допущения и ограничения;
- *ключевые участники и заинтересованные стороны;*
- ресурсы проекта;
- сроки;
- риски;
- критерии приемки;
- *обоснование полезности проекта*.

Приоритет проекта – это предварительная качественная оценка его финансовой и стратегической ценности для организации.

Финансовая ценность имеет следующую градацию:

- высокая. Ожидаемая окупаемость до одного года. Ожидаемые доходы от проекта не менее чем в 1,5 раза превышают расходы. Все допущения при проведении этих оценок четко обоснованы;

- выше среднего. Ожидаемая окупаемость проекта от одного года до трех лет. Ожидаемые доходы от проекта не менее чем в 1,3 раза превышают расходы;
- средняя. Проект позволяет улучшить эффективность производства в компании и потенциально может снизить расходы компании не менее чем на 30 %. Проект может иметь информационную ценность или помочь лучше контролировать бизнес;
- низкая. Проект снижает расходы компании не менее чем на 10 % и дает некоторые улучшения производительности производства.

Стратегическая ценность имеет следующую градацию:

- высокая. Проект обеспечивает стратегическое преимущество, дает устойчивое увеличение рынка или позволяет выйти на новый рынок. Решает значительные проблемы, общие для большинства важных клиентов. Повторение конкурентами затруднено или потребует от одного до двух лет;
- выше среднего. Проект создает временные конкурентные преимущества. Выполнение обязательств перед многими важными клиентами. Конкурентное преимущество может быть удержано в течение одного года;
- средняя. Поддерживается доверие рынка к компании. Проект повышает мнение клиентов о качестве предоставляемых услуг или способствует выполнению обязательств перед несколькими клиентами. Конкуренты уже имеют или способны повторить новые возможности в течение одного года;
- низкая. Стратегическое воздействие отсутствует или незначительно. Влияние на клиентов несущественно. Конкуренты могут легко повторить результаты проекта.

Цели проекта – формулировка качественно нового состояния, которое будет достигнуто при реализации проекта:

- изменения в компании или в ее бизнес-процессах для повышения эффективности основной производственной деятельности;
- реализация стратегических планов, расширение рынка;
- выполнение контрактов;
- решение специфических проблем.

Результаты проекта – описание ключевых характеристик продукта или выгод его использования:

- перечень бизнес-выгод заказчика;
- описание продукта или услуги;
- высокоуровневые требования к продукту;

Допущения и ограничения – наиболее важные особенности предметной области, условия и требования к реализации, внешние обстоятельства, способные повлиять на исполнение проекта:

- риски, переложенные на заказчика, – допущения;
- специфические нормативные требования. Например, обязательная сертификация продукта, услуги на соответствие определенным стандартам;
- специфические технические требования. Например, разработка под заданную программно-аппаратную платформу;
- специфические требования к защите информации.

Краткое технико-экономическое обоснование проекта – сжатое описание экономической и технологической ситуации до и после реализации проекта:

- для кого предназначены результаты проекта;
- описание текущей ситуации, существующих проблем у потенциального заказчика;
- каким образом результаты проекта решают эти проблемы;
- насколько значимо для клиента решение данных проблем, оценка экономического эффекта;
- какие преимущества в итоге из этого может извлечь компания – исполнитель проекта.

Ключевые участники и заинтересованные стороны могут быть определены в широком диапазоне, начиная с социальных групп и ведомственной принадлежности и заканчивая персоналиями:

- спонсор проекта;
- заказчик проекта – лицо или организация, которые будут использовать продукт, услугу или результат проекта;
- пользователи результатов проекта;
- куратор проекта – представитель исполнителя, уполномоченный принимать решение о выделении ресурсов и изменениях в проекте;
- руководитель проекта – представитель исполнителя, ответственный за реализацию проекта в срок, в пределах бюджета и с заданным качеством;
- соисполнители проекта.

Ресурсы проекта – предварительная оценка затрат на реализацию проекта, включающая:

- трудоемкость и срок исполнения;
- людские ресурсы и требования к квалификации персонала;
- оборудование, услуги, расходные материалы, лицензии на ПО;
- критические компьютерные ресурсы;

- бюджет проекта, план расходов с разбивкой по статьям и фазам / этапам проекта.

Для некоммерческих, исследовательских и инновационных проектов возможна более свободная форма обоснования, включающая:

- *бизнес-требования* – обоснование полезности проекта, особенности проекта, обеспечивающие его привлекательность, предполагаемые отличия от аналогов, проблемы предметной области и способы их решения в проекте, возможности коммерческого использования, способы монетизации;
- *границы проекта* – перечень бизнес-процессов, поддерживаемых и не поддерживаемых системой;
- *перечень пользователей* проекта.

Диаграмма классов в модели предметной области

Предметная область, как и любая другая система, в методологии ООП описывается в виде структурной и поведенческой компонент.

Описание структуры начинается с установления **бизнес-сущностей** и структурных отношений между ними – ассоциаций и генерализации / наследования. В UML такому представлению соответствует диаграмма классов, здесь она называется **диаграмма классов предметной области.** Если быть совсем точным, то – диаграмма классов модели предметной области. Формулировки бизнес-сущностей заносятся в глоссарий.

Замечание по теме. При описании связей бизнес-сущностей необходимо учитывать особенности поведения системы. При выполнении сценариев системы необходимые сущности должны быть достижимы по связям-ассоциациям. Например, билет в театр может быть куплен в кассе, забронирован по Интернету, оплачен через платежную систему. Во всех трех случаях необходимы его ассоциации с различными бизнес-сущностями – касса / смена, клиент, аккаунт в платежной системе. Поэтому вопрос, что определять в первую очередь, структуру или процессы, описывающие поведение, остается открытым.

Модель предметной области в бизнес-аналитике тесно связана с системной аналитикой – *управлением требованиями*. При внедрении система сама становится частью бизнес-процесса, поэтому модель предметной области преобразуется в *модель анализа* как основы для адекватного представления состояния предметной области в программной системе. Возможные варианты такого преобразования:

- бизнес-процессы не меняются, часть их переходит в безбумажные технологии, часть физических процессов отслеживается системой путем фиксации их текущего состояния в бизнес-объектах и БД;

- бизнес-процессы оптимизируются с учетом технологий и решений, предоставляемых системой;
- система создает собственную модель бизнес-процессов, опираясь на существующие аналоги.

Отметим типичные ошибки при разработке диаграммы классов предметной области:

- диаграмма классов предметной области описывает содержание системы, а не ее поведение. Поэтому сущности типа «оформление заказа», «приобретение билета» сюда не относятся;
- в модель предметной области не включаются временные сущности и отношения, например данные об отчете, если он только генерируется и выводится, но не хранится в архиве отчетов в виде файла;
- в модель предметной области не включаются отношения между сущностями, если они не сохраняются в системе и не используются в описании ее поведения. Например, зритель, покупающий билет в кассе, взаимодействует с системой опосредованно через кассира, данные о нем в этом случае не сохраняются, поэтому отношения «клиент – кассир (смена)» в модели отсутствуют.

При наличии различных вариантов связи бизнес-сущности с другими отношение должно устанавливаться между этой сущностью и *интерфейсом* или *абстрактным классом.* Например: билет может быть продан через кассу, забронирован через Интернет либо оплачен через платежные системы. Отношение устанавливается между сущностями «билет – способ продажи». Способ продажи – это интерфейс или абстракция, от которой наследуются сущности: «оплата через кассу», «бронирование», «оплата через платежную систему».

Проект средней руки. Диаграмма классов предметной области системы учета рейтинга успеваемости изображена на рис. 6.8.

Ключевые моменты в модели предметной области:

- связанные пары сущностей «группа–студент», «дисциплина–единица контроля» – обычная двухуровневая иерархия;
- *рейтинг* – основная сущность модели, соответствующая паре «группа–дисциплина», при создании рейтинга устанавливаются необходимые отношения с другими сущностями и ассоциированные с ними данные. С отношением «студент–рейтинг» связана ассоциированная сущность «бригада», связанными сущностями являются также «срок сдачи», «оценка», «пропуск»;
- на диаграмме (рис. 6.8) не отражена семантика ассоциаций, связанных с сущностью «рейтинг». Например, для ассоциации «студент–рейтинг» вида *многие ко многим* существует ассоциированный класс «бригада». Экземпляры ассоциации создаются не между любыми парами объектов, а для тех и только

тех объектов «студент», которые имеют отношение к *группе*, связанной с данным объектом «рейтинг». Аналогично, отношение «студент–единица контроля» устанавливается только между экземплярами сущностей, связанных с общим рейтингом через сущности «группа» и «предмет»;

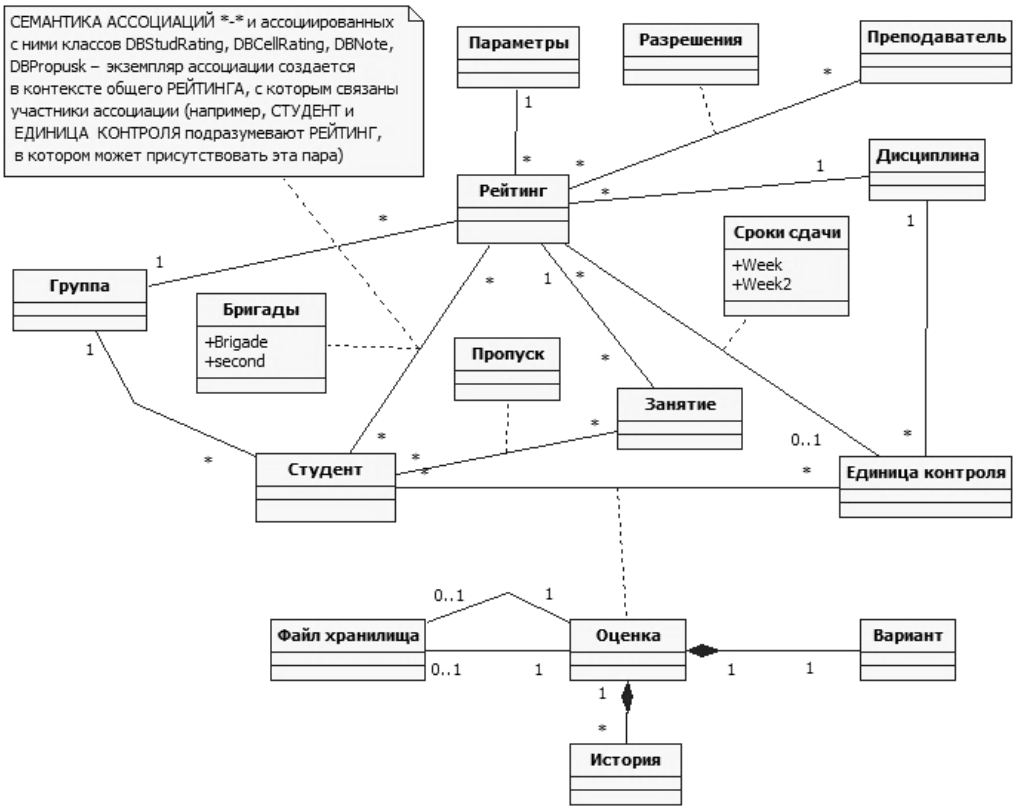


Рис. 6.8. Диаграмма классов предметной области системы учета рейтинга

- с отношением «преподаватель–рейтинг» связан ассоциированный класс «разрешения»;
- с рейтингом связан набор параметров, используемых для его вычисления.

Бизнес-анализ процессов предметной области

В простых системах бизнес-процессы могут быть описаны словесно в виде сценариев, в которых фигурируют *бизнес-сущности* – материальные объекты и люди-акторы. В процессе анализа выделяются бизнес-сущности, а их определения записываются в глоссарий.

Пример. Неформальный бизнес-анализ. Система выдачи ключей аудиторий

Модель предметной области. Курсивом отмечены бизнес-сущности и их свойства.

Вуз территориально состоит из *учебных корпусов*, в каждом из которых имеется, как правило, одна *вахта*, изредка несколько. На вахтах организовано дежурство *вахтеров*, обычно в виде суточных *смен.* Графики дежурств вахтеров более или менее постоянны, хотя возможны подмена и перевод из корпуса в корпус.

На вахте находятся ключи от *аудиторий,* аудитория имеет основной и *запасной комплект ключей*. На каждой вахте ведутся *журнал выдачи ключей* и *журнал постановки на пультовую охрану* (постановка ни пульт). В журнале выдачи ключей на *каждую дату* открывается новый блок записей, содержащих *время выдачи, номер аудитории, фамилию сотрудника* и *подпись взявшего*, *время сдачи* и *подпись сдавшего*. Журнал постановки на пульт заполняется пустыми записями на *каждую дату под каждую аудиторию*. При выдаче ключа в нее вносятся *время выдачи, номер аудитории, фамилия сотрудника* и *подпись взявшего*, при сдаче – *время сдачи, фамилия сотрудника* и *подпись сдавшего*. Если ключ берется и сдается несколько раз, то в журнале постановки на пульт делаются добавочные записи.

Кроме вахтеров, к системе имеют отношение *преподаватели,* ведущие учебный процесс, *персонал*, обслуживающий здание и учебный процесс (технички, электрики, буфет), а также *сотрудники отдела охраны*, в том числе *начальник отдела.*

Преподаватель имеет *удостоверение*, в котором записаны фамилия, имя и отчество, подразделение, сроки действия (продления), номера аудиторий, а также вклеена фотография.

Аудитории подразделяются на *кафедральные лаборатории*, для пользования которыми необходимо иметь разрешение, прописанное в удостоверении, *учебные аудитории*, доступ в которые разрешен все преподавателям, а также *служебные помещения*, доступ в которые разрешен персоналу. При получении ключа *мультимедийной учебной аудитории* преподаватель должент оставить на вахте удостоверение.

Пример описания бизнес-процесса «выдача ключа преподавателю». Для получения ключа преподаватель представляет вахтеру удостоверение и называет номер аудитории. Вахтер идентифицирует преподавателя по фотографии в удостоверении. При отсутствии ключа на вахте может быть выдан

запасной. Преподаватель по журналу может определить, кем и когда был взят требуемый ключ. Преподаватель вносит запись в журнал выдачи ключей, содержащую время, аудиторию, фамилию и подпись. При наличии в аудитории пультовой охраны заполняется запись во втором журнале для соответствующей аудитории. Если запись уже заполнена, вносится новая. Если аудитория является мультимедийной, преподаватель оставляет в залог удостоверение. После всех этих процедур он получает на руки ключ.

Замечание по теме. Легко заметить, что приведенное описание является неполным. В нем не отражены разнообразные форс-мажорные ситуации, например, потеря ключа, оставление аудитории открытой и т. п. Если в обыденной жизни такие ситуации решаются зачастую неформально, то при встраивании в бизнес-процесс программной системы они должны быть урегулированы на уровне системной аналитики (см. раздел 6.3).

Пример неформального бизнес-анализа. Система учета рейтинга успеваемости

Система ориентирована на хранение и интеграцию данных об успеваемости и артефактов учебного процесса. Взаимодействие участников бизнес-процесса здесь сведено к минимуму. Поэтому основными элементами бизнес-анализа здесь являются бизнес-правила формирования рейтинга.

Формальная схема вычисления рейтинга. Общий рейтинг определяется как сумма рейтингов по единицам контроля минус снижение за пропуски занятий:

- **способы подсчета**: *ручной* (экспертный, субъективный) – исходный балл ставится преподавателем в пределах указанного максимума, система в формировании оценки участия не принимает (экзамен, зачет, индивидуальное задание, курсовая работа), *автоматический* – исходный балл устанавливается по нормативу и снижается или увеличивается по формальным критериям;
- **формальные критерии качества** – при автоматическом подсчете балла вводится список критериев качества, по которым производится увеличение или снижение балла на фиксированный процент по каждому из них, например, на 10 %. Признаки можно выбирать группой. Процент снижения устанавливается при создании рейтинга;
- **учет сроков сдачи** – для ряда единиц контроля вводится линейная шкала снижения балла на заданный процент за каждую просроченную неделю относительно установленного срока сдачи. Срок сдачи для каждой единицы контроля устанавливается при создании рейтинга в соответствии с расписанием.

Число недель, в течение которых балл снижается, ограничено, вводится предельное снижение, например 50 %. При досрочной сдаче производится симметричное увеличение;

- **учет посещаемости** – для некоторых видов единиц контроля предусмотрен контроль посещаемости, они вносятся в отдельный список при создании рейтинга, этот список может дополняться преподавателем. Предусмотрено фиксированное снижение рейтинга, устанавливаемое при создании, например 0,5 балла за пропуск. Учет посещаемости может проводиться по желанию преподавателя;
- **весовые коэффициенты вычисления рейтинга** составляют типовые наборы и назначаются отдельно каждому рейтингу:
  - процент снижения / увеличения балла по сроку сдачи – 7–10 % за неделю;
  - количество недель, в течение которых действует предыдущее снижение / увеличение – 5–7 недель, т. е. максимум 50 % в обе стороны;
  - процент изменения балла за каждый параметр качества – 10 %;
  - количество баллов, снимаемых за каждый пропуск, – 0,5–1 балл.

Очевидность оценки в стандартных ситуациях. Очевидная выгода предлагаемой схемы – естественное соотношение закона и справедливости в стандартных ситуациях, что делает процедуру оценки более прозрачной и снижает психологическую напряженность:

- «*двоек не ставим*». Если вся учебная нагрузка выполнена в срок и без претензий, то студент зарабатывает рейтинг, достаточный для получения тройки на экзамене. При этом средний и высокий уровень ответов на экзамене оценивается соответственно;
- «*досрочно и автоматом*». Если студент сдает все досрочно, то ему достаточно небольшого индивидуального задания для получения «автомата». Можно также сдавать часть экзамена, например, без теории;
- «*так себе, на троечку*». При наличии небольших долгов студент может быть допущен к экзамену, но по общему рейтингу выше четверки не получит, а реально может претендовать только на тройку;
- «*оптом не дешевле*». Если студент в течение семестра не занимается, то, сдав в конце семестра полностью все задания, он получает максимум 50 % баллов от исходного, т. е. необходимый минимум для допуска к экзамену. При этом все должно быть выполнено без претензий к качеству. На экзамене он фактически получает на один традиционный балл ниже и на тройку сдавать не имеет права.

Формальные модели описания бизнес-процессов

Для формального описания бизнес-процессов используются средства, развитые в таких аспектах, как обработка потоков объектов и описание параллелизма – взаимодействие и синхронизация.

Из арсенала структурных моделей (см. раздел 1.2) часто используются диаграммы потоков данных. Основными элементами модели являются *внешние сущности, системы* / *подсистемы, процессы, накопители данных, потоки данных* (рис. 6.9). Как и в моделях потоков данных, используемых при описании вычислительных процессов, граф связей отражает не последовательность исполняемых действий в одном компоненте системы, а перемещение объекта данных или реального объекта по технологической цепочке его накопления и обработки.

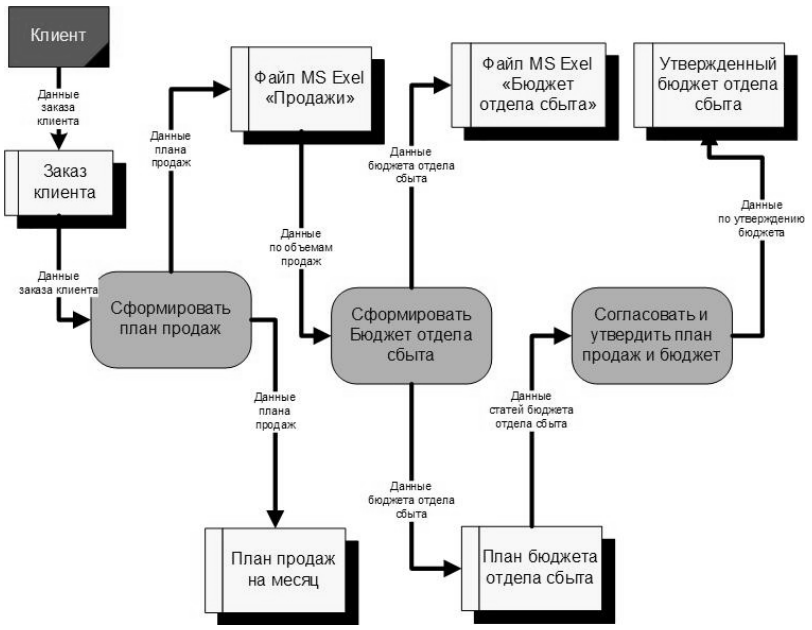


Рис. 6.9. Диаграмма потоков данных

Из всего арсенала диаграмм UML наиболее популярны в бизнес-анализе диаграммы деятельности. Они представляют собой сочетание нотации обычных блок-схем и сетей Петри (рис. 6.10). Последние дают средства для распараллеливания потоков управления и их обратного слияния в один. Дорожки на диаграмме обозначают акторов или бизнес-сущности, которые реализуют части этого процесса.

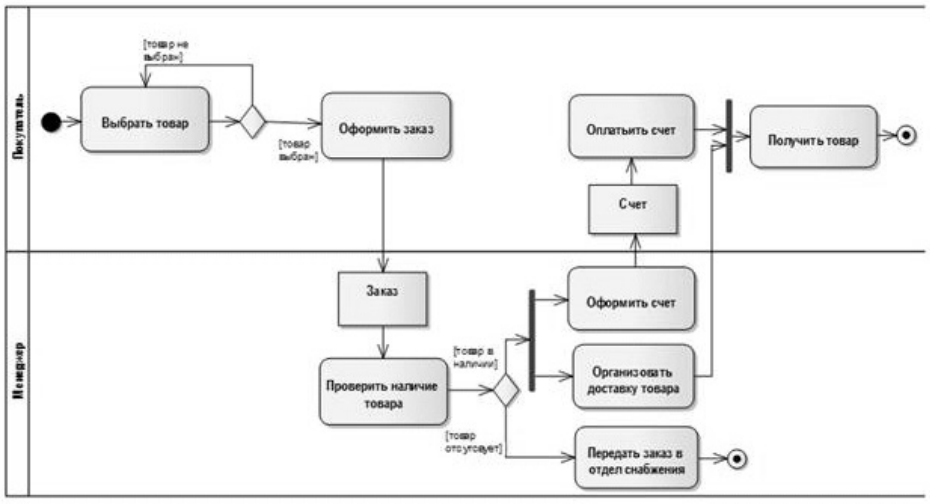


Рис. 6.10. Диаграмма деятельности в UML

Специализированные языки и средства для описания бизнес-процессов, например BPMN (Business Process Modeling Notation) [42, 43], фактически представляют собой расширение нотации диаграмм деятельности UML для практических целей (рис. 6.11).

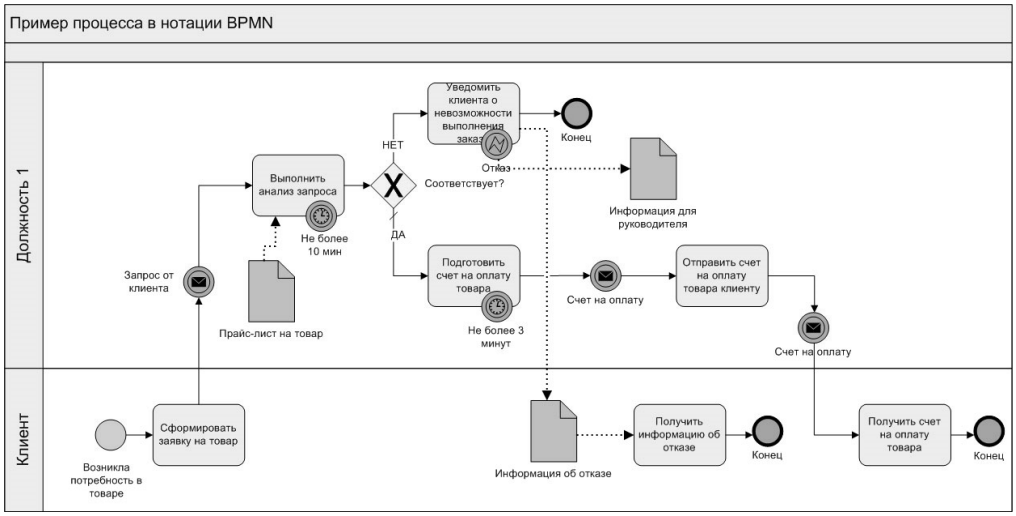


Рис. 6.11. Пример модели в BPMN

Они содержат значки для обозначения стереотипов (расширений) базовых элементов диаграмм.

6.3. Системная аналитика: прецеденты, сценарии, модели

> Остап окончил свой труд, вынул из «дела Корейко» чистый лист бумаги и вывел на нем заголовок: «ШЕЯ» Многометражный фильм. Сценарий О. Бендера.
>
> *И. Ильф, Е. Петров. Двенадцать стульев*

Функционалом системы занимается системная аналитика. После того как в процессе бизнес-анализа построена модель предметной области, к бизнес-процессу подключается программная система, для описания функционала которой необходимо:

- создать адекватное отображение структуры предметной области в системе – *диаграмма классов анализа*;
- описать интерфейс взаимодействия предметной области с системой – *прецеденты*;
- описать бизнес-процессы в системе как продолжение бизнес-процессов предметной области – *сценарии*;
- создать базу знаний фактов и свойств системы, касающихся как отдельных сценариев, так и системы в целом – *функциональные требования*.

Описание программной системы, включенной в бизнес-процесс, на функциональном уровне называется **моделью анализа**. Между моделью предметной области и моделью анализа возможны различные варианты взаимоотношений:

- в простейших случаях можно не делать различий между предметной областью и системой. Предметная область может создаваться системой. Например, сетевая игра, социальная сеть. Материальная сторона бизнес-процессов может отсутствовать вовсе или сводиться к взаимодействию пользователей вне системы, например передача ведомости доставки от курьера к менеджеру;
- включение системы в бизнес-процесс может поменять его структуру. Обычно это оптимизация и новые возможности, предоставляемые информационными технологиями. Это должно быть в общих чертах отражено в бизнес-архитектуре. Естественно, что модель анализа должна отражать соответствующие изменения функционала.

Диаграмма прецедентов

Прецедент (вариант использования, use case) – ограниченное по времени и функционалу атомарное взаимодействие пользователя с системой. Самым общим представлением функционала является диаграмма прецедентов – пере-

чень атомарных сценариев взаимодействия (прецедентов) и связанных с ними пользователей (актеров) (рис. 6.12). Между овалами прецедентов и фигурками актеров устанавливаются связи – ассоциации. Между двумя прецедентами может быть установлена зависимость одного из видов:

- *включение* (include) – целевой прецедент является частью прецедента-источника;
- *расширение* (extend) – целевой прецедент выполняется при определенных условиях исполнения прецедента-источника.

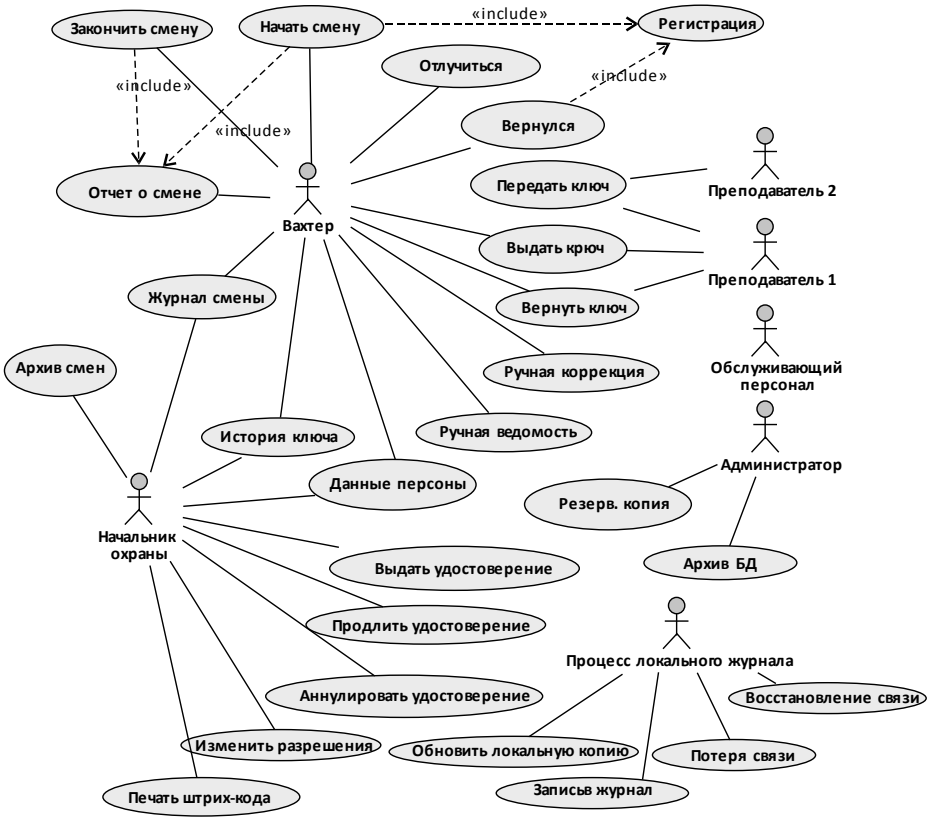


Рис. 6.12. Диаграмма прецедентов системы выдачи ключей аудиторий

Между двумя актерами или двумя прецедентами может быть определено отношение обобщения (наследования) – целевой прецедент является более общим, а источник – его расширением.

На самом деле вместо термина «пользователь» применяется термин «актер» (англ. actor – актор, эктор). Он имеет более широкий смысл: в качестве

актеров могут выступать самые различные стимуляторы описываемого функционала:

- обычные пользователи. Реальный пользователь может соответствовать нескольким актерам, с каждым из которых связана определенная *роль* в системе. Например, при авторизации пользователь получает определенные привилегии, каждая из которых соответствует отдельной роли – актеру;
- внешние подсистемы по отношению к системе, реализующей этот функционал;
- внешние датчики / приемники сигналов, ассоциируемые с соответствующими физическими объектами управляемой системы;
- время как актер для периодически реализуемого функционала.

Например, фоновые периодические процессы, такие как обновление или синхронизация данных могут быть описаны как внутренний актер, лишь бы у него была внятная роль. Такими актерами могут быть, например, автоматическое распределение заказов в службе управления вызовами такси, служба управления локальным журналом в системе управления выдачей ключей аудиторий и т. п.

Следует обращать внимание на полноту списка прецедентов, чтобы не допускать пропуска видов работ, выполняемых пользователями (рис. 6.13).

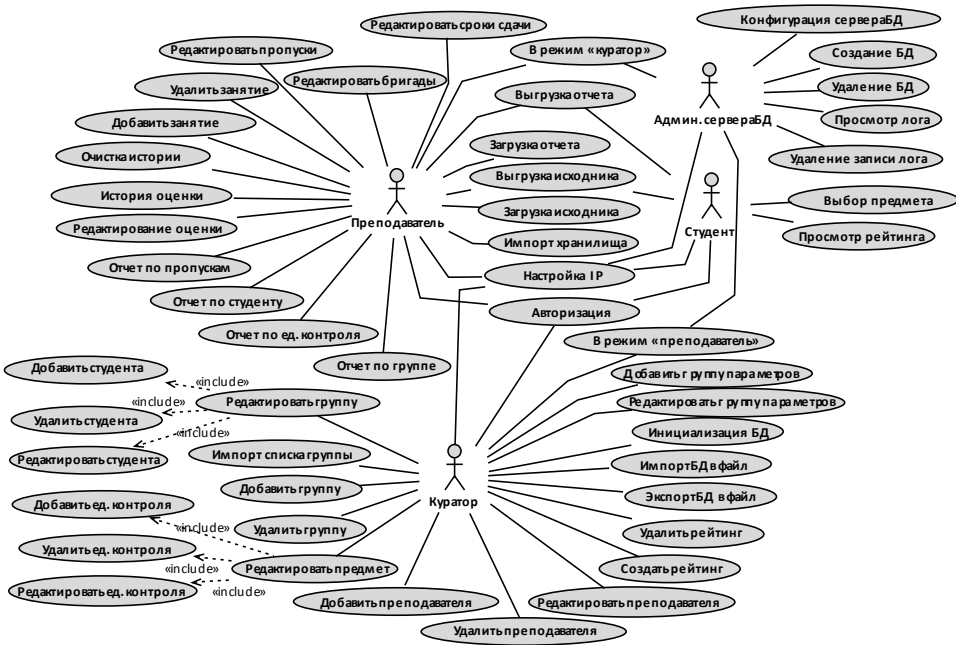


Рис. 6.13. Диаграмма прецедентов системы учета рейтинга успеваемости

Модель анализа. Диаграмма классов анализа

Рассмотрим в качестве примера, как будет эволюционировать модель анализа в системе выдачи ключей аудиторий. Исходная модель предметной области, приведенная на рис. 6.14, разработана на основе анализа бизнес-процессов (см. раздел 6.2) и выделения в нем бизнес-сущностей.

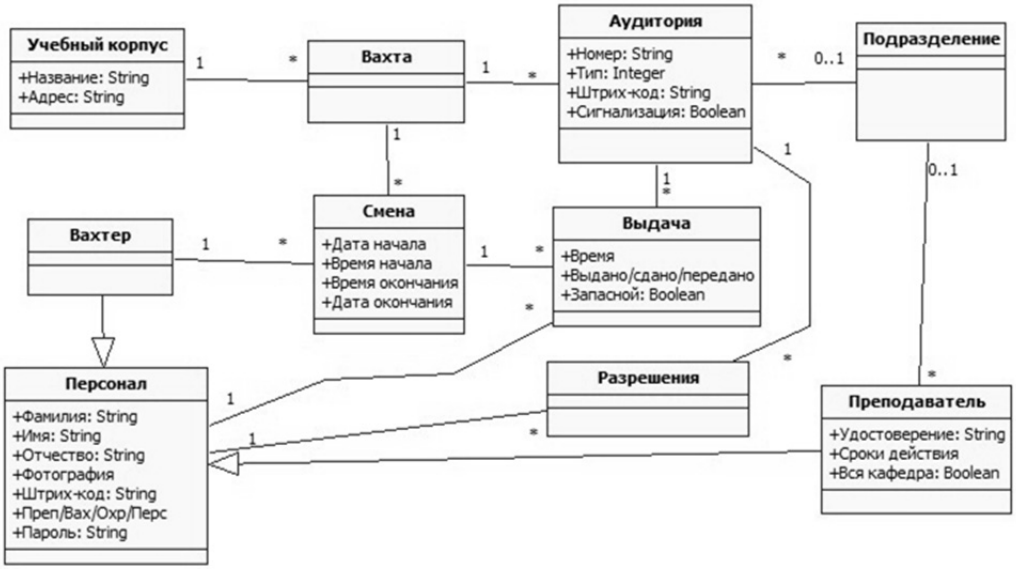


Рис. 6.14. Диаграмма классов предметной области системы выдачи ключей

В функционал системы вносятся следующие изменения, которые отражаются в структуре диаграммы классов модели анализа (рис. 6.15):

- выдачу разрешений удобно производить не только по отдельным преподавателям, но и для подразделения в целом. В модели появляется базовый класс разрешений и два производных класса – разрешения *для персонала* и разрешения *для подразделений*;
- для установления состояния аудитории «открыта / закрыта» необходимо производить поиск связанного объекта с последней датой / временем в классе «выдача». К тому же возможны разные нестандартные ситуации, когда ключ сдан, а аудитория открыта, либо наоборот – ключ не сдан, но аудитория закрыта. Поэтому в класс «аудитория» необходимо продублировать данные о последней выдаче для основного и запасного ключей. Тогда класс «выдача» становится *чистым журналом событий*, доступ к которому не обязателен для ведения текущих операций.

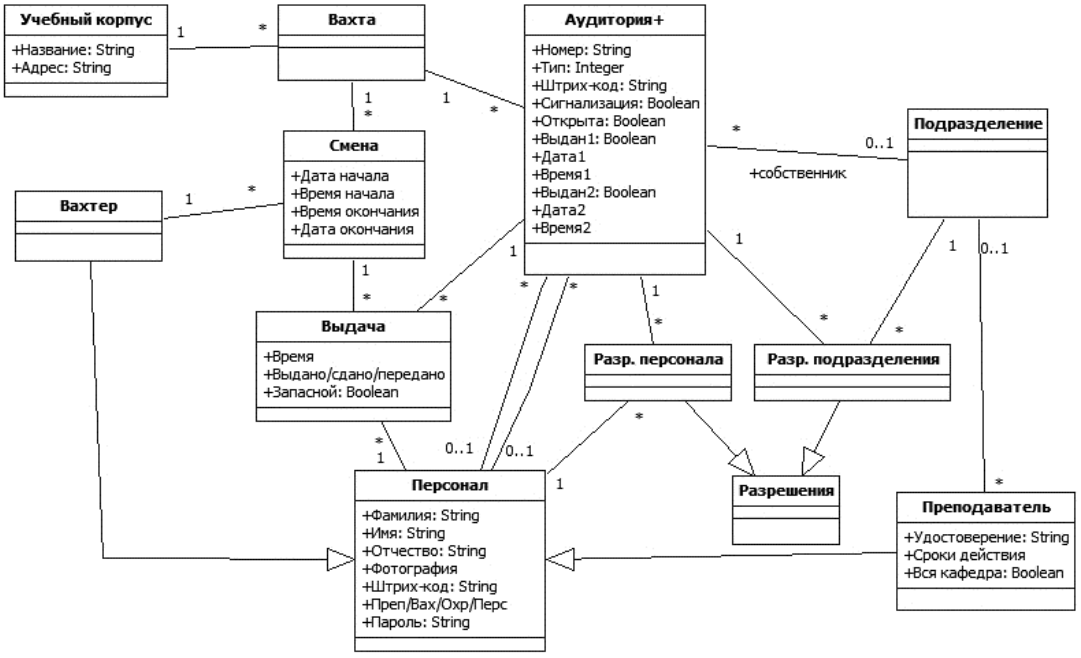


Рис. 6.15. Модель классов анализа на основе модификации модели предметной области

Тестирование диаграммы классов модели анализа состоит в проверке возможности исполнения над ней сценариев, соответствующих прецедентам. При этом проверяется доступность связанных данных, правильность ассоциаций между классами.

Сценарии

Сценарий – это текстовое описание потока событий при выполнении прецедента. Сценарии служат для описания функционального поведения системы. В исходном виде сценарий может представлять собой содержательное описание взаимодействия актера с системой в рамках прецедента. При его формализации создается заголовок, содержащий ряд атрибутов (табл. 6.1) и собственно описание потока событий в виде пар «стимул–реакция» (табл. 6.2). Анализируются имена существительные в тексте сценария на предмет принадлежности к модели анализа: актеры, сущности предметной области или их атрибуты.

Таблица 6.1

Заголовок сценария

Атрибут	Значение
Наименование прецедента	Выдать ключ
Актеры	Вахтер. Преподаватель
Предусловия	Вахтер зарегистрирован. Смена открыта. Открыт локальный журнал
Активаторы	Преподаватель спрашивает ключ от аудитории
Цель	Получение ключа на руки
Краткое описание	Получение ключа с фиксацией события в БД и документированием факта выдачи

Таблица 6.2

Поток событий сценария

Действие пользователя (стимул)	Реакция системы
1. Вахтер: сканирует штрих-код ключа на стенде	Если есть соединение с БД, система берет состояние ключа из БД, иначе – из локального журнала
	Если ключ уже выдан, выводится поле со справкой – когда и кому
2. Вахтер: сканирует штрих-код преподавателя	Если есть соединение с БД, данные берутся оттуда, иначе – из локального журнала. Если в локальном журнале нет данных, то преподаватель заносится в локальный журнал без разрешений на получение ключей по фамилии на штрих-коде
	Если тип аудитории – лаборатория, то проверка разрешения. Если нет разрешения – отказ (держится 10 с) и завершение сценария, иначе выводится сообщение «Выдать ключ»
3. Вахтер: сканирует штрих-код команды «выдать»	Если ключ отмечен как выданный, выводится вопрос «Ключ на вахте?», иначе переход к п. 6
4. Вахтер: сканирует штрих-код команды «да»	Оформляется фиктивный возврат ключа. Переход к п. 6
5. Вахтер: сканирует штрих-код команды «нет»	Если есть соединение с БД – редактируется объект «аудитория» и добавляется объект «выдача». Редактируется локальный журнал. Печатается чек.
6. Преподаватель подписывает чек	–

Архитектура или функционал?

При функциональном проектировании возникает одна методологическая проблема. Функциональное проектирование отвечает только на вопрос «что», но не отвечает на вопрос «как», следовательно, она принципиально не должна затрагивать вопросы программной архитектуры. При этом само понятие функционала может быть размытым.

В качестве примера рассмотрим проблему повышения надежности системы выдачи ключей аудиторий. При пропадании соединения с сервером система должна продолжать работать с локальной копией журнала, синхронизируя внесенные изменения при восстановлении соединения – вести *локальный журнал*. Возможные варианты:

- считать локальный журнал элементом программной архитектуры, тогда он по определению является прозрачной, т. е. невидимой пользователю компонентой и никак не проявляет себя на функциональном уровне;
- считать локальный журнал частью функционала. В этом случае логика его работы – предмет системной аналитики, и он должен быть отражен в модели анализа и как элемент структуры, и как элемент поведения (см. приведенный выше сценарий). Что же касается конечного пользователя, то совсем не обязательно делать журнал видимым, он может быть частью скрытого функционала.

Еще одна проблема возникает при описании поведения системы на функциональном уровне. В сценариях описано, что происходит с бизнес-сущностями при его исполнении, но не указано, какие компоненты системы этим занимаются. Если такие компоненты **функциональной архитектуры** внести в описание сценариев, то получим полноценную систему функциональных классов трех видов – стереотипов:

- *boundary* – граничный класс, соответствующий актеру;
- *entity* – бизнес-сущность;
- *control* – управляющий класс, контроллер, элемент функциональной архитектуры.

Элементы функциональной архитектуры на уровне системной аналитики могут быть выделены только в самом общем виде, например, *подсистема авторизации, хранилище файлов, основная система контроля*. На этапе архитектурного проектирования они должны быть спроецированы в программную архитектуру.

С использованием этой триады можно разрабатывать UML-диаграммы *устойчивости, последовательности* или *коммуникаций* для описания сценария реализации прецедента в системе (рис. 6.16).

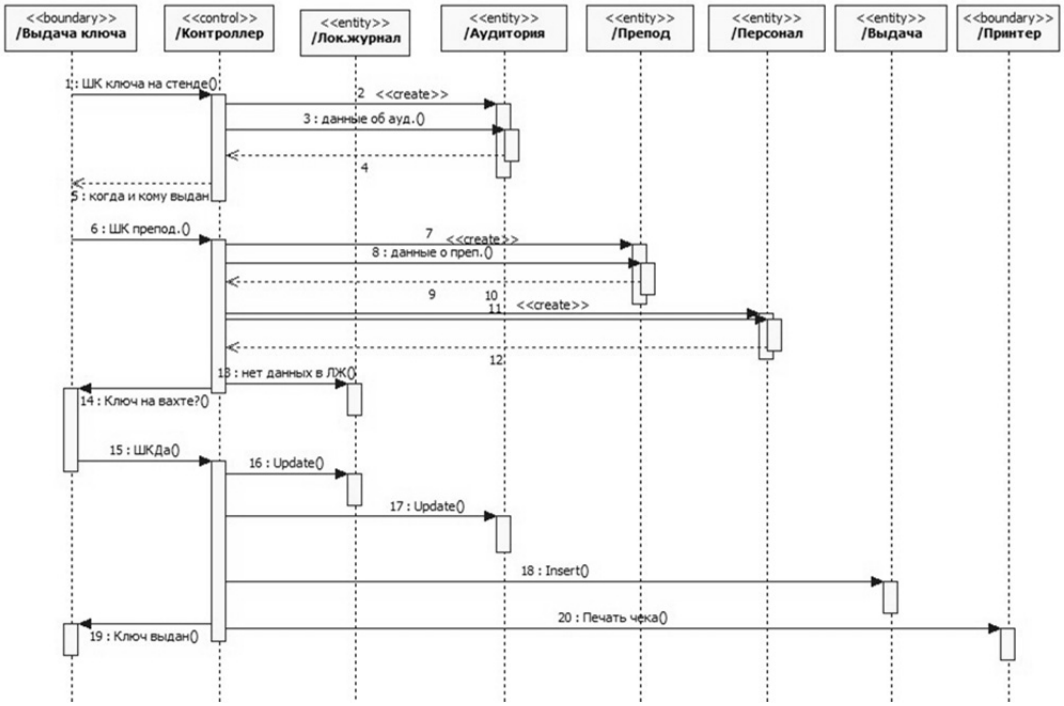


Рис. 6.16. Диаграмма последовательности для прецедента

Данный пример является номинальным, так как функциональная архитектура представлена единственным объектом, т. е. фактически не проработана.

Замечание по теме. В небольших проектах, а также при использовании гибких методологий, ориентированных на коммуникации, в подробном расписывании всех сценариев нет необходимости. Исполнитель, получив прецедент для его реализации, вполне может сконструировать умозрительный сценарий. Для этого ему потребуется:

- исходные данные по представлению в технологическом процессе проектирования моделей предметной области и анализа – бизнес-объекты, структура БД;
- спецификации требований в виде иерархического справочника – базы знаний фактов, установленных для системы (см. раздел Документирование требований);
- прототип графического интерфейса.

Еще одним элементом функционального описания является определение возможных состояний объектов бизнес-сущностей и актеров. Для класса определяется набор состояний объекта, переходы между ними производятся при

исполнении прецедентов. Таким образом получается автоматная модель, которая описывается UML-диаграммой состояний (рис. 6.17).

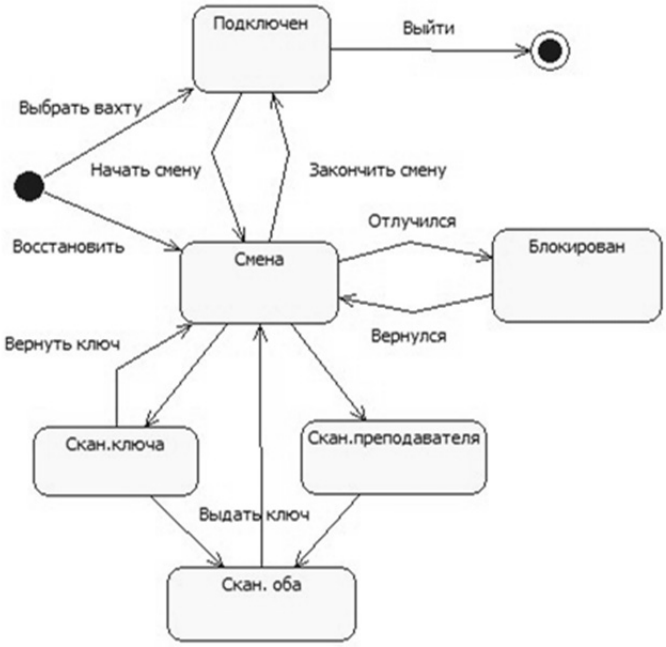


Рис. 6.17. Диаграмма состояний граничного класса «вахтер»

Автоматная модель описывает интегрированное поведение сущности на принципах событийного моделирования. Состояние автомата проверяется и меняется в сценариях, реализующих прецеденты.

6.4. Управление требованиями

Функциональное описание системы – это не только сценарии поведения, отображение предметной области и графический интерфейс. Сюда входит множество фактов, каждый из которых может отражаться в различных компонентах функционала – своего рода база знаний о проекте. В узком смысле они и являются **требованиями** к системе. В широком смысле в технологический процесс **управления требованиями** входят все перечисленные выше артефакты и процессы работы с ними. На рис. 6.18 изображены компоненты функционального описания, связанные с требованиями.

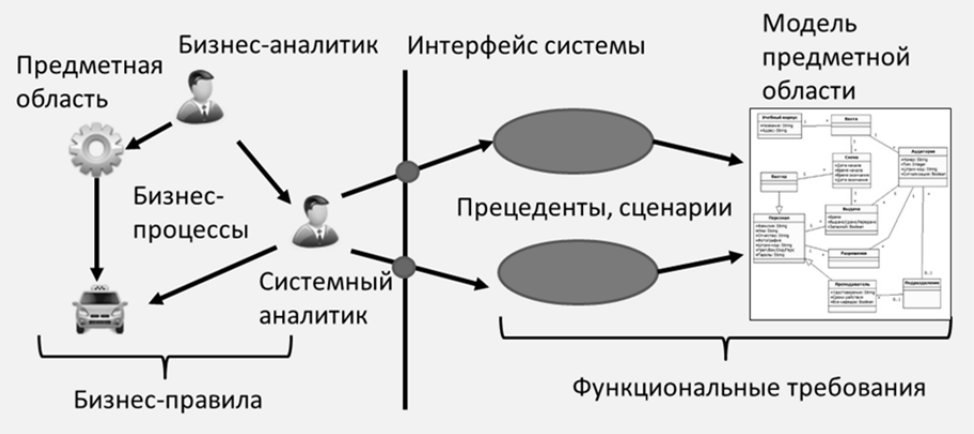


Рис. 6.18. Виды и место требований в описании системы

Технологический процесс управления требованиями привязан в основном к первым двум фазам жизненного цикла: исследованию и развитию. Классикой жанра в этой области является работа К.И. Вигерса [32], где приводится общепринятая классификация требований (рис. 6.19):

- *бизнес-требования* – цели создания системы, получаемый эффект, образ и границы проекта, компоненты бизнес-анализа, создаваемые на этапе исследования проекта (см. раздел 6.1 и 6.2);
- *требования пользователей* – описания возможных действий пользователей и их поведения: варианты использования, сценарии, специфические факты потребительских свойств системы;
- *функциональные требования* – перечень реализуемого функционала с описанием основных параметров и характеристик, факты, касающиеся нескольких прецедентов и сценариев;
- *системные требования* – перечень требований, распространяющихся на всю систему в целом, требования к программной системе, ПО и аппаратным средствам;
- *бизнес-правила* – законы, постановления, юридические нормы, сложившаяся практика, любые формально выраженные ограничения, отношения и зависимости в предметной области;
- *атрибуты качества* – эффективность, надежность, простота использования и т. п., качественные характеристики и их количественная мера оценки;
- *ограничения*, следующие из бизнес-правил, *условия*, при которых невозможно выполнение сценариев.

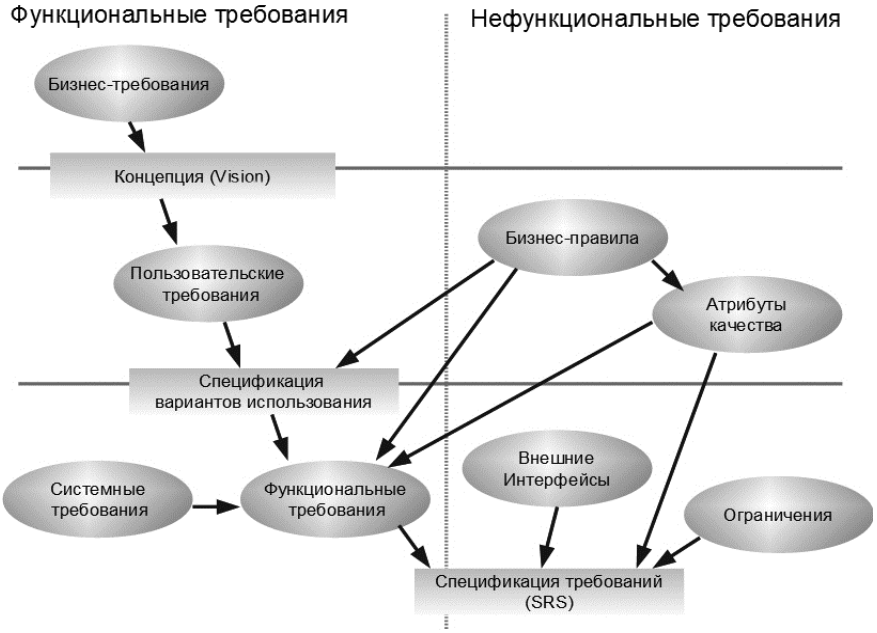


Рис. 6.19. Классификация и документы требований по К.И. Вигерсу

Некоторые виды требований связаны с деятельностями в процессах бизнес-анализа и архитектурного проектирования:

- в какой «экологической нише» будет работать проект, в чем его привлекательность, социальная и экономическая значимость, способы монетизации и сроки окупаемости, перспективы – *бизнес-требования*;
- как видится процесс работы с системой, каковы ее потенциальные пользователи и их квалификация, основные сценарии работы – *требования пользователей*;
- параллельно с проработкой предметной области появляются *бизнес-правила*, определяющие законы, по которым живет предметная область;
- на этапе определения архитектуры, т. е. собственно начала проектирования возникают *системные требования* и *атрибуты качества* – потребительские и эксплуатационные свойства системы, обеспечиваемые этой архитектурой;
- на этапе развития и построения возникают *требования внешних интерфейсов.*

Поскольку основной задачей системной аналитики являются описание и сопровождение функционала системы, то основная группа требований – *функциональные требования*.

Бизнес-правила

Бизнес-правила – это своего рода Бейсик предметной области. Все формализуемые положения или ограничения процессов предметной области формулируются в следующих категориях:

- факты, инварианты – соотношения, справедливые в течение всего времени функционирования системы;
- ограничения – условия, ограничивающие возможность исполнения определенных действий;
- активаторы операций – условия, приводящие к активизации или выполнению действий;
- выводы – аналог условных конструкций «если … то»;
- вычисления – функциональные зависимости, формулы.

Атрибуты качества

Атрибут качества – обобщенное качественное свойство системы. Атрибуты качества не связаны с отдельными элементами функционала, они характеризуют свойства системы в целом как готового продукта. Наиболее распространенные атрибуты качества:

- доступность – процент или время доступа с учетом сбоев, отказов, установки;
- эффективность – затраты на обеспечение производительности;
- гибкость – расширяемость, масштабируемость, легкость конфигурирования под различные условия применения;
- целостность – гарантированная безопасность;
- надежность – гарантии работы без сбоев;
- устойчивость к сбоям, способность к восстановлению;
- удобство использования (useability);
- удобство эксплуатации;
- мобильность, портируемость;
- повторное использование кода и данных.

Атрибуты качества – наиболее сложный для формулировки компонент. Всегда хочется, чтобы система была идеальной, как минимум «приятной во всех отношениях». В реальности же всегда приходится балансировать между качеством и затратами на его обеспечение. Поэтому системный аналитик решает здесь несколько проблем:

- оценка важности и актуальности каждого атрибута качества именно для данного варианта системы;

- определение меры, в которой это качество может быть описано, протестировано и предъявлено;
- оценка архитектурных затрат на реализацию атрибута качества в данном виде, поскольку многие показатели качества обеспечиваются не столько аппаратными средствами, сколько архитектурными решениями.

Основная ошибка при формулировании атрибутов качества – восприятие их именно как *характеристик качества* системы. Отсюда появляются требования, выраженные фразами типа «система должна быть проста в использовании», «система должна быть интуитивно понятной». Коренной порок таких требований – их нельзя проверить из-за именно качественного характера формулируемых свойств. Отсюда главное умение системного аналитика – находить для атрибутов качества *количественную меру*, наиболее соответствующую этому атрибуту, а также оценивать возможные архитектурные затраты на обеспечение качества на указанном уровне меры.

Классификация и оценка требований

Прежде всего требования необходимо оценить и классифицировать по разным категориям. Например, в RUP приняты следующие параметры классификации:

- типы требований:
  - запросы заинтересованных сторон;
  - свойства – требования абстрактного вида;
  - программные требования – системные требования к поведению отдельных компонент;
  - прецеденты;
  - функциональные требования;
  - нефункциональные требования;
- категории нефункциональных требований:
  - практичность (Usability);
  - надежность (Reliability);
  - производительность (Performance);
  - поддержка и среда проектирования (Supportability);
  - безопасность;
- характеристики требований:
  - приоритеты: обязательное, желательное, возможное, перспективное;
  - полезность: критическое, важное, полезное;
  - решаемость: проблематичное, выполнимое, тривиальное;
  - трудоемкость: высокая, средняя, малая.

Как бы ни была хороша систематизация требований, она не гарантирует качество процесса. Некачественный процесс разработки требований может проявить себя следующими пороками:

- недостаточное вовлечение пользователей, пропуск классов пользователей и компонент функционала. Как правило, за бортом оказывается вспомогательный, но необходимый для работы функционал, например отчетность, архивирование, администрирование;
- разрастание требований, неограниченный поток изменений. Необходимо жестко фильтровать требования с точки зрения их полезности, частоты использования, важности и трудоемкости;
- «золочение» продукта – стремление довести до совершенства функционал, сделать его универсальным там, где работают временные решения и заплатки;
- двусмысленность, недостаточная спецификация требований, небрежное планирование – типичные организационные моменты любого планирования.

Известно утверждение, что 80 % пользователей используют 20 % функционала. В то же время функционально ограниченный продукт выглядит непривлекательно, даже если этот функционал является редко используемым. Анализ требований должен давать оценку *важности* требований. Оценивать необходимо не только отношение пользователей к факту присутствия функционала, но и к факту его отсутствия (рис. 6.20).

<table>
<tr><td colspan="2" rowspan="2">Оценка</td><td colspan="5">Отсутствие</td></tr>
<tr><td>нравится</td><td>ожидаю</td><td>все равно</td><td>смирюсь</td><td>не нравится</td></tr>
<tr><td rowspan="5">Наличие</td><td>нравится</td><td>???</td><td colspan="3">привлекательный</td><td>линейный</td></tr>
<tr><td>ожидаю</td><td rowspan="3">лишний</td><td colspan="3" rowspan="2">безразличный</td><td rowspan="2">обязательный</td></tr>
<tr><td>все равно</td></tr>
<tr><td>смирюсь</td><td colspan="3">лишний</td><td>???</td></tr>
<tr><td>не нравится</td></tr>
</table>

Рис. 6.20. Оценка потребительских качеств функционала

Сочетание оценки присутствия и отсутствия функционала дает качественную оценку его важности:

- обязательный – отсутствие недопустимо, реализация выше некоторого предела не влияет на качество системы;
- линейный – пропорциональное добавление функционала дает соответствующее увеличение оценки качества;
- привлекательный – дополнительный, расширяющий применение системы.

Естественно, что для разных групп пользователей этот показатель важности будет различен. Кроме того, для оценки эффективности реализации требования необходимо учитывать частоту его использования, трудоемкость, что условно математически можно изобразить так:

Эффективность = Важность · Частота использования / Трудоемкость.

Известен также набор обязательных характеристик требований – SMART:

- **S**pecific – точность и конкретность, подробность и внятность описания требования;
- **M**easurable – измеримость, возможность сопоставления с требованием количественной оценки (метрики);
- **A**chievable – степень достижимости, сложность реализации, трудности формализации;
- **R**elevant – релевантность, значимость для проекта и для исполнителя;
- **T**ime bound / framed – ограниченность во времени, временные рамки для реализации.

Разработка и управление требованиями

С требованиями связаны два рабочих потока – *разработка и управление*. Под разработкой понимается процесс извлечения требований, их классификация, наполнение содержанием. Управление – это процесс отслеживания продвижения и реализации требования во всех технологических процессах жизненного цикла.

Перечисленные выше свойства требований и способы их оценки относятся, согласно SWEBOK, *к рабочему потоку разработки требований*. Разработка требований включает следующие деятельности:

- извлечение – получение первичных данных о требованиях. Включает опросы пользователей, изучение документации по предметной области, наблюдение за работой пользователей, маркетинговые исследования, работу с прототипами, анализ сценариев работы пользователей, встречи заинтересованных лиц;
- анализ – трансформация требований от формулировки пользователей к виду, готовому к исполнению, в том числе классификация, определение атрибутов, сопоставление с моделями проектирования;
- документирование – разработка документов – спецификаций требований, объединяющих разрозненные требования в единое целое;
- валидация требований – проверка полученных требований на корректность и их утверждение. Включает инспекцию требований, прототипирование, приемочное тестирование.

Анализ и валидация требований не проводятся на пустом месте. Здесь необходима привязка к тем моделям системы и ее окружения, которые имеют место на этапе жизненного цикла, например, к *модели бизнес-процессов, модели предметной области, модели анализа, описанию архитектуры.* Иначе у системного аналитика отсутствуют основания для проведения анализа требований вообще: как оно соотносится с реальными бизнес-процессами, как оно может быть архитектурно обосновано, как оно вписывается в текущие сценарии работы в модели анализа.

Управление жизненным циклом требований

Реализация требований происходит в нескольких технологических процессах: собственно управление требованиями, проектирование, конструирование, тестирование, управление конфигурациями. Поэтому основная задача в этом случае – координация и отслеживание деятельностей исполнителей. Сюда входят:

- взаимодействие с технологическим процессом управления конфигурациями: управление изменениями и контроль версий;
- контроль состояния требования – отслеживание его состояния в процессе реализации;
- отслеживание затрат на реализацию путем введения соответствующих метрик, чтобы иметь статистику влияния функциональной составляющей проекта на его стоимость;
- контроль связей с другими требованиями и дисциплинами разработки.

Процесс прохождения требования описывается простой диаграммой состояний (рис. 6.21).

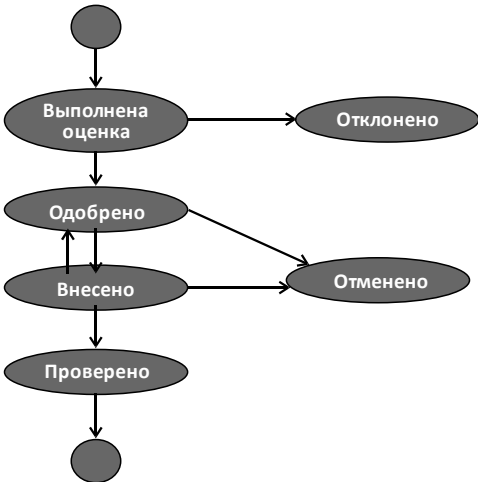


Рис. 6.21. Состояния требования в процессе управления

В реализации требований задействованы исполнители (рис. 6.22), деятельности которых должны быть скоординированы относительно этого требования. Этот процесс называется *трассируемость требований.*

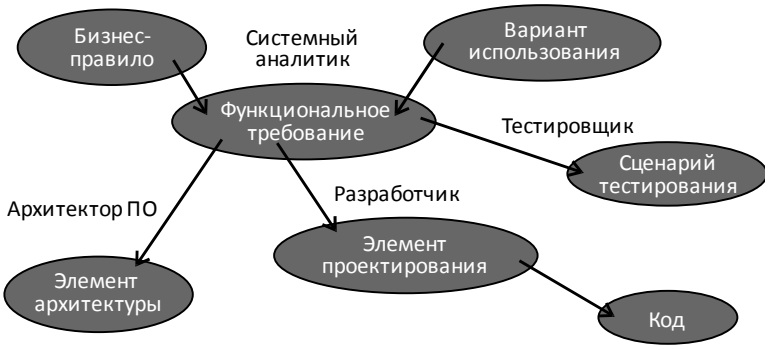


Рис. 6.22. Связь требований с деятельностями, ролями и артефактами ЖЦ

Приведенная схема является идеализированной, в действительности связи требования с другими элементами проекта могут быть вида 1 : 1, 1 : N, N : N. Например, при детализации атрибута качества «устойчивость к сбоям» в виде конкретного требования необходимо внести изменения в код различных компонент, ведомых разными исполнителями.

Требования, как и остальные артефакты проекта, постоянно изменяются в процессе разработки. Поэтому им свойственна версионность. Согласованная версия требований называется **базовой версией (BaseLine).** В процессе работы с требованиями системный аналитик формирует очередную базовую версию, после фиксации которой она становится доступной к использованию другими участниками проекта.

Документирование требований

В разделе 1.1 обсуждались вопросы самоценности и самодокументируемости кода. Каркас кода или код прототипа могут выступать в качестве спецификаций в фазах развития и построения. Требования в любом случае нуждаются в документировании, если только эта процедура не пущена на самотек. Начнем, как всегда, с недостатков спецификаций. К ним относятся:

- терминологическая неопределенность – отсутствие глоссария или привязки к нему;
- отсутствие представления о классификации требований, подмена категорий и смешение требований;

- фокусировка на деталях пользовательского интерфейса;
- излишнее акцентирование внимания на деталях реализации;
- слабая формализация бизнес-процессов.

Само требование должно обладать набором атрибутов, используемых при его классификации и в процессах управления, например:

- дата создания требования, автор, ответственный или список заинтересованных лиц, происхождение или источник;
- состояние требования, обоснование требования;
- затрагиваемые подсистемы, номер версии продукта;
- метод проверки или критерий тестирования приемлемости;
- характеристики: приоритет, важность, полезность, решаемость, трудоемкость;
- стабильность – вероятность изменения в будущем.

Форма документа описания требований должна соответствовать его дальнейшему использованию в принятой методологии. Возможные варианты:

- стандартизованный документ в тяжеловесных технологиях проектирования ПО. Образцом может служить «Спецификации требований к ПО» по стандарту IEEE 830–1998 [32];
- техническое задание на внешнее исполнение для системы в целом или ее части – аутсорсинг. Включает структуру БД, системные требования, бизнес-правила, требования к графическим и внешним программным интерфейсам и форматам, развернутые сценарии. Техническое задание в таком варианте содержит результаты фаз исследования и развития для той компоненты, которую оно описывает;
- иерархический справочник в гибкой технологии разработки ПО. При наличии модели предметной области, модели классов анализа, описания структуры БД, прототипа графического интерфейса, а также взаимопонимания между системным аналитиком и программистами нет особой необходимости расписывать сценарии. В конце концов, программист, изучив все перечисленные документы, способен понять, *в какой последовательности надо давить на кнопки*. В дополнение к этому необходим общий иерархический справочник фактов, касающихся системы. Признаки классификации могут быть разными: особенности системы, варианты использования, режим работы, классы пользователей, стимулы, реакции, классы объектов или уровни функциональной иерархии. Желательно использовать именованные теги вида A.B.C, например *правила.рейтинг.сумма*. Один и тот же факт может присутствовать в разных ветках классификации, в этом случае необходимы взаимные ссылки.

Проект средней руки. Пример фрагмента иерархического справочника для системы контроля рейтинга успеваемости. Перечислены только иерархические теги и наименования требований.

Бизнес-правила

правила.рейтинг.параметры.начало_семестра – текущая неделя отсчитывается от этого параметра.

правила.рейтинг.параметры.штраф_за_пропуск – балл, снимаемый за пропуск занятия **(kp).**

правила.рейтинг.параметры.процент_за_показатель – начисляемый или снимаемый процент за показатель качества исполнения единицы контроля **(dq).**

правила.рейтинг.параметры.процент_за_неделю – процент, снимаемый за одну неделю просрочки. При досрочной сдаче симметрично добавляется **(dw).**

правила.рейтинг.параметры.недель_просрочки – интервал в неделях, в течение которого снимается процент, в дальнейшем он остается на достигнутом уровне.

правила.рейтинг.сумма – сумма баллов рейтинга по обязательным единицам контроля должна составлять 100. Для учета индивидуальных внеплановых заданий и бонусов вводятся единицы контроля с весом 0.

правила.рейтинг.пропуск – за пропуск занятия из общей суммы снимается балл рейтинга, указанный в пункте **параметры.рейтинг.параметры.штраф_за_пропуск**.

правила.рейтинг.показатель_качества_исполнения – устанавливается как логическое значение, предусматривает увеличение и уменьшение балла на фиксированный процент, за сложность, оформление и т. п.

правила.рейтинг.единица_контроля.норматив – каждой единице контроля назначается нормативное количество баллов **(pi).**

правила.рейтинг.единица_контроля.субъективная – балл в единице контроля ставится преподавателем неформально, по собственным оценкам в пределах установленного значения либо по критериям, не включенным в систему.

правила.рейтинг.единица_контроля.формальная – балл в единице контроля выставляется системой по формуле расчета;

правила.рейтинг.единица_контроля.формула:

$$N0 – (w – w0)dw + dq\sum pi.$$

Атрибуты качества

качество.код.повторное использование – весь код, за исключением слоев управления представлением, независим от приложения и может быть повторно использован. Контроллер приложения для преподавателя полностью отделен от представления и используется одновременно в desktop- и android-приложениях.

качество.доступность.автономно – при отсутствии соединения с сервером приложение преподавателя может работать с локальными копиями рейтингов, которые открывались при работе с сетью. При повторном входе и наличии соединения с сервером внесенные изменения автоматически синхронизируются с сервером БД.

качество.доступность.масштабируемость.сервер_БД – приложение имеет настройки на сервер БД. Сервер БД содержит список независимых однотипных БД.

Функциональные требования

функция.оценка.история – БД должна хранить последовательность записей об изменении каждой оценки и обеспечивать просмотр.

функция.оценка.история.очистка – возможна очистка истории по всему рейтингу с оставлением последней оценки.

функция.рейтинг.имя – имя рейтинга в формате <название предмета> <пробел><название группы>.

функция.хранилище.объем – для каждого рейтинга должно выводиться количество файлов отчетов и исходников и их объем.

функция.хранилище.скачивание – преподаватель скачивает все файлы отчетов и исходников для выбранного рейтинга одним действием в выбранный каталог, при скачивании файлы из хранилища удаляются.

функция.хранилище.загрузка.разрешение – загрузку файлов в хранилище могут выполнять преподаватель и студент.

функция.хранилище.скачивание.путь – в выбранном каталоге для скачивания создается каталог с именем рейтинга.

функция.куратор.БД.экспорт – куратор может сохранить содержимое БД рейтингов в файл оригинального формата.

функция.куратор.БД.импорт – куратор может загрузить содержимое БД рейтингов из файла.

функция.куратор.БД.инициализация – куратор инициализирует БД рейтингов, старая БД уничтожается, создается новая структура БД.

функция.преподаватель.разрешения – преподаватель имеет список разрешений на просмотр / редактирование рейтингов, устанавливаемый куратором.

функция.куратор.разрешения – куратор в роли преподавателя имеет доступ ко всему списку рейтингов.

функция.куратор.единица_контроля.порядок – порядок следования единиц контроля при выводе редактируется куратором.

функция.куратор.список_группы – список группы студентов может быть загружен из текстового файла.

функция.вывод.студент – в выпадающих списках и текстовых полях при просмотре и редактировании рейтинга отчество студента не выводится.

функция.единица_контроля.тип – набор типов единиц контроля зашит в приложения и не хранится в БД.

Отчеты

отчеты.формат – необходимо представление всех отчетов в форматах html, pdf и интерактивной форме.

отчеты.интерактивная_форма – вывод отчета в виде экранной формы с возможностью прокрутки и позиционирования к основной форме редактирования через клик по ячейке таблицы с установкой параметров выбранной ячейки в основной форме.

отчеты.виды – для рейтинга отчеты по оценкам должны формироваться по группе в целом, по единице контроля, по студенту, а также по пропускам для всей группы.

Графический интерфейс

GUI.мобильный.отчет – вывод отчета в мобильном приложении должен быть реализован в интерактивной форме. При ограничении размеров экрана прокрутка должна сопровождаться *выделением цветом* выбранных строки и столбца.

GUI.desktop.отчет.интерактивная_форма – прокрутка таблицы отчета должна производиться с сохранением в панели просмотра заголовков строк и столбцов.

GUI.desktop.подсказки – клик правой кнопкой мыши по любой кнопке формы сопровождается выводом подсказки о ее функции.

GUI.форма.title – каждая форма в заголовке содержит основные данные о позиционировании в формате – <адрес сервера>.<имя БД>.<фамилия студента>.<название предмета>.

Несколько замечений по системной аналитике

В заключение отметим несколько проблем, которые возникают при столкновении функционала с реальной жизнью.

«Голь на выдумки хитра». Необходимо учитывать варианты поведения пользователя, начиная от неосознанного заблуждения и заканчивая попытками извлечения мелких выгод и проведения крупных махинаций. Например, ложное сообщение об отмене заказа на определенной стадии его выполнения с целью экономии на оплате его предоставления системой.

Многообразие жизненных ситуаций. Описание бизнес-процессов и их отражение в системной аналитике являются абстрактными моделями, не способными охватить все многообразие жизненных ситуаций. К таковым относятся прежде всего различные форс-мажорные обстоятельства. Да и сами описания могут содержать пробелы – отсутствие прописанных действий в определенных ситуациях. Возможна ситуация, когда невозможно продолжение процесса в системе в необходимом для пользователя направлении имеющимися вариантами функционала. В таких случаях ситуация решается *вручную* вплоть до принудительной коррекции данных системы службой технической поддержки.

Уровень контроля за бизнес-процессом. Система должна адекватно отображать бизнес-процессы, чтобы у пользователя не было необходимости *записывать что-то на бумажке.* В то же время отсутствие гибкости, излишняя регламентированность действий могут приводить к описанным выше блокировкам либо сказываться на потребительских свойствах системы. Здесь необходим компромисс между гибкостью и точностью следования бизнес-процессу.

Ошибки пользователей. Реакция системы на ошибки пользователей относится обычно к ведению графического интерфейса. Здесь же имеются в виду ошибки *несоответствия вводимых данных реальным намерениям пользователей*, например, ввод неверного номера или даты утверждения приказа в систему документооборота. Желательно иметь возможность отмены или коррекции таких действий хотя бы на другом уровне. Например, ошибочная отмена заказа в мобильном приложении водителем может быть скорректирована диспетчером путем его повторного назначения.

6.5. За рамками интуитивно понятного интерфейса

Когда речь заходит о графическом интерфейсе (GUI), часто пользуются расхожим выражением «интуитивно понятный». Попробуем оценить его критически:

- предполагается, что GUI настолько очевиден и соответствует функционалу, что у пользователя не может возникать никаких специфических проблем,

связанных с ним. Справедливости ради следует сказать, что если функционал системы является интуитивно понятным, а графический интерфейс ведет пользователя в соответствии с ним, то он также является интуитивно понятным;

- в унифицированном процессе проектирования разработка GUI является деятельностью второго плана в процессах функционального проектирования. Считается, что если функционал достаточно проработан с точки зрения требований и сценариев, то проблематика GUI ограничена особенностями графического дизайна, эргономики и т. п.;
- требование, сформулированное как «интуитивно понятный», уже само по себе является ересью, так как не может быть протестировано.

Получается, что и говорить не о чем, кроме как о внешнем виде кнопочек или модных трендах в виде плиток. Оставим в стороне внешний вид как таковой и попробуем оценить GUI с более широких позиций: насколько он помогает или мешает пользователю в достижении целей при его работе с системой. Здесь можно выделить факторы, с точки зрения которых следует рассматривать GUI:

- производительность;
- человеческие ошибки;
- обучение;
- субъективное восприятие;
- запоминание, распределение пространства экрана, поиск, визуализация, навигация.

Данный материал представляет собой сжатое свободное изложение [29], подкрепленное собственными дополнениями, выводами и примерами.

Производительность

Производительность GUI или скорость работы с ним является комплексной оценкой всего процесса работы пользователя с системой через GUI и включает следующие этапы:

- осознание или формулирование цели в процессе обдумывания;
- определение последовательности действий в том же процессе;
- исполнение действий;
- восприятие ответа;
- оценка результата.

Для оценки производительности GUI в такой форме используется методика, известная как **GOMS** (**G**oals, **O**perators, **M**ethods and **S**election **R**ules – цели, операторы, методы и правила их выбора).

Способы исполнения действий также влияют на производительность. Сюда относятся меню, горячие клавиши для опытных пользователей, пиктограммы и непосредственное манипулирование элементами GUI (перечислены в порядке увеличения производительности).

В психологии используется термин «фокус внимания». Мы сосредоточимся на технологической стороне этого термина.

Фокус внимания – концентрация внимания на некотором предмете, его поведении и управлении им. Восстановление фокуса внимания требует определенных временных и психологических затрат.

В GUI фокус внимания относится к объектам на экране, которые мы видим и которыми можем манипулировать. Если в процессе работы мы переключаемся на другой объект, то для продолжения работы с первым необходимо не просто его появление на экране, но еще и восстановление в сознании пользователя контекста исполняемого действия в процессе восстановления фокуса внимания. Контекст исполнения может включать:

- исполняемое действие;
- шаг, на котором остановился пользователь;
- введенные параметры;
- текущий фокус ввода – позиция курсора.

Замечание по теме. Здесь напрашивается очевидная аналогия с состоянием процессора при многозадачной или многопоточной работе.

Длительность физических действий. Любое действие может быть либо быстрым, либо точным. Время достижения цели обратно пропорционально размеру цели и дистанции до цели. Сказанное справедливо при манипулировании любыми объектами в GUI. Соответственно чем короче перемещения при манипулировании, тем выше производительность. Соответственно можно расположить элементы GUI в последовательности убывания производительности:

- контекстное меню по месту расположения объекта;
- диалоговое окно по месту элемента управления;
- граница экрана как псевдокнопка. При залипании курсора на границе экрана появляется большой элемент GUI, который дает быстрое позиционирование и не требует точности.

Длительность реакции системы. Если выполняемая системой операция приводит к ощутимым задержкам, то необходима корректная оценка времени,

в течение которого не требуется вмешательства пользователя. Полезные рекомендации:

- перед началом длительного процесса все данные должны быть получены сразу, недопустим запрос дополнительных данных после начала выполнения операции;
- желательна установка тайм-аута на всплывающие окна с подтверждением операции, по истечении интервала времени по умолчанию принимается положительный ответ;
- обеспечение реального прогресс-индикатора. Типичной ошибкой является игнорирование финальных операций, когда индикатор со значением «осталось 0 секунд» висит достаточно долго.

Адаптация. Желательно, чтобы система подстраивалась под контекст, в котором работает пользователь, что позволяет минимизировать число выполняемых в GUI действий. Полезные рекомендации:

- разделение настроек по умолчанию для повторяющихся и эпизодических действий. Например, текущий каталог сохранения файлов меняется при сохранении в новый каталог более двух раз подряд, однократные эпизодические сохранения в другой каталог не изменяют принятого умолчания;
- разделение настроек по умолчанию для разных типов операций. Например, сохранение отчетов и экспорт данных производятся в каждом случае в свой текущий каталог;
- адаптация окон настройки операции в зависимости от истории работы, последовательная развертка окон настроек с различной степенью подробности. Например, первое окно – кнопка OK и ссылка «параметры», второе окно – кнопка OK, последние измененные или часто используемые параметры, третье окно – кнопка OK и все параметры.

Человеческие ошибки

Человеку свойственно ошибаться. Дружественный характер GUI предполагает, что система не является ментором, уязвляющим пользователя, но предлагает варианты исправления. Ошибки могут быть вызваны различными причинами:

- пробелы в знаниях предметной области;
- опечатки;
- моторные ошибки, связанные с неточным манипулированием мышью;
- снижение внимания, пропуск или игнорирование предупреждений.

Замечание по теме. Эта тема перекликается с программными ошибками (см. раздел 8.2) в том смысле, что системе тоже свойственно ошибаться.

И в том и в другом случае ошибки желательно предупреждать, сообщать пользователю не столько о самой ошибке, сколько о способе ее исправления. Возможные меры предотвращения ошибок:

- обучение пользователей в процессе работы;
- снижение требований к внимательности;
- повышение разборчивости и заметности индикаторов;
- снижение чувствительности системы к ошибкам:
  - блокировка потенциально опасных действий пользователя, использование пред- и пост-подтверждений;
  - проверка системой всех действий пользователя перед их принятием, ограничение диапазона и формата данных – выбор вместо ввода;
  - адаптация системы к действиям пользователя – умолчания, запоминание статистики и истории работы.

Способы предотвращения опечаток:

- использование выбора вместо ввода, вывод границ допустимого диапазона, отображение недопустимого ввода сменой цвета / курсивом;
- учет фактора потери бдительности. Например, не стоит делать кнопку OK кнопкой по умолчанию;
- использование команды явного разблокирования вместо подтверждения операции.

Качественный дизайн GUI предполагает исправление ошибок во время совершения действия либо исключает возможность их появления.

Обучение и самообучение

Под обучением понимается не только система подсказок и программной документации. В более широком смысле – это учет всех факторов, позволяющих использовать возможности системы пользователями с различными уровнями квалификации и знаниями предметной области. Для пользователя любая система ценна возможностью решения с ее помощью текущих задач, желательно с минимальными затратами на вхождение. При этом само по себе наличие определенных возможностей еще ни о чем не говорит, воспользоваться ими могут помешать:

- незнание о существующих возможностях – «что можно сделать?»;
- сложность поиска элементов управления – «где это найти?»;
- неумение ими пользоваться – «как это сделать?».

Замечание по теме. Разнообразие уровней знаний о системе и предметной области, в которой она работает, порождает разнообразие вариантов и способов обучения. Например, для обычного видеоредактора можно выделить следующие категории пользователей:

- *случайный пользователь* имеет минимальное представление о предметной области, однократно использует программу для выполнения разовой работы, например, для сведения двух видеофрагментов в один;
- *дилетант* имеет поверхностное представление о предметной области, использует программу для решения ограниченного ряда простых задач, например, создание простого видеоряда с титрами;
- *продвинутый пользователь* использует разнообразный функционал программы, но без вникания в тонкости, например, для производтва заказного любительского видео;
- *профессионал* досконально знает предметную область и систему, желает получить с ее помощью качественный продукт, использует систему как рабочий инструмент, например, для производства профессионального клипа.

Если для случайного пользователя нужен перечень описания решений типовых задач по принципу «какие кнопки давить», то для дилетанта и продвинутого пользователя нужно либо упрощенное описание, либо система подсказок в интуитивно понятном интерфейсе. Для профессионала нужны подборки материалов по художественному дизайну, адаптированные к системе.

Для создания целостного образного представления о системе используются следующие средства:

- ментальная модель – логическое или образное описание организации системы, с которой работает пользователь;
- метафора – представление интерфейса в виде аналога, знакомого пользователю;
- аффорданс – очевидность функционала, исходящая из визуального образа объекта;
- стандарт – общепринятые правила обозначения действий и свойств.

Метафора – интерфейс системы воспроизводит *предметную область,* например полиграфия, звукозапись, типографская верстка. В этом случае графический интерфейс моделирует общеизвестный реальный предмет (рис. 6.23).

Метафоры могут быть различными, охватывать как систему в целом, так и отдельную ее часть, например, красная лампочка как индикатор ошибки или опасности. Предмет метафоры может быть придуман разработчиком и не иметь никакого отношения к предметной области.

Несмотря на свою привлекательность, использование метафор таит в себе следующие проблемы:

- сложность подбора;
- метафора может ограничивать систему;
- со временем метафора может стать архаичной, если сам предмет метафоры перестает использоваться, например печатная машинка;
- метафора должна быть общеизвестной;
- метафора должна покрывать весь функционал, новый функционал сложно вписать в имеющуюся метафору.

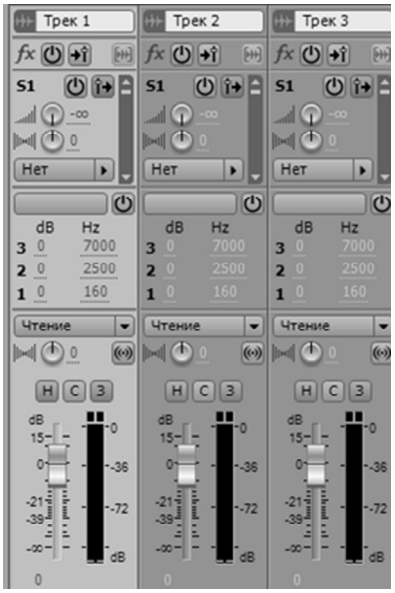


Рис. 6.23. Метафора звукового редактора – режиссерский пульт

«Аффорданс» – термин из области психологии, означает свойство предмета, позволяющее сразу понять, как этим предметом пользоваться, либо получить информацию о его состоянии (рис. 6.24). Как правило, аффорданс ограничен визуальной компонентой, звуковые, тактильные и пространственные ощущения, для которых принцип аффорданса может быть применен, в графическом интерфейсе не используются. С явной натяжкой к аффордансу можно отнести имена файлов типа *ReadMeFirst.txt* или *RunMe.exe*.

Рис. 6.24. Пример аффорданса

В основе аффорданса лежит графическая очевидность. В терминах и названиях, используемых в GUI, может иметь место и терминологическая очевидность – общепринятый смысл термина, основанный на определенном образном представлении. Но в некоторых случаях образное представление может

быть ошибочным, тогда смысл термина меняется на противоположный. Приведем примеры:

- если во главу угла ставить не программу, а файл, то запись данных в файл – это ввод (input), в чтение – вывод данных из файла (output);
- обычно при работе с сервером префикс *down* подразумевает получение данных с сервера, а *up* – передачу данных на сервер. Так обычно и изображается на схемах и рисунках. В то же время в программах обмена данными с мобильными телефонами заливка данных на сервер иногда фигурировала под термином down, а загрузка с него – как up, поскольку телефон мыслился, видимо, как вершина мироздания.

Стандарт. Сюда относятся не только стандарты в общепринятом смысле, но также и общепринятые элементы графической, цветовой и прочей информативности, т. е., по существу, тот же аффорданс. Например:

- стандарт графического дизайна приложений;
- цветовая гамма: красный / желтый / зеленый – недоступен / подключение / доступен, оранжево-черный – обозначение опасности;
- общепринятые пиктограммы.

До сих пор речь шла об априорных знаниях и представлениях о системе. Собственно информационное наполнение справочной системы включает следующие варианты местоположения справочных данных и способов их получения:

- бумажная документация, электронные документы, видеоуроки;
- электронный структурированный документ – система помощи (help);
- отдельные экранные формы, диалоги;
- контекстное меню, всплывающие подсказки;
- управляющие элементы экрана, строка состояния.

В системе документации приняты следующие виды обучающих материалов:

- базовая справка – назначение системы, используется при первом входе;
- обзорная справка – обзор функций системы;
- справка предметной области – сведения о предметной области и профессиональных приемах работы;
- процедурная справка – способы реализации основных функций системы (желательна привязка к графическому интерфейсу, системе помощи);
- контекстная справка – назначение элементов управления;
- справка состояния;
- сообщения об ошибках.

Справочные материалы желательно максимально вписывать в контекст работы.

Запоминание, распределение пространства экрана, поиск, визуализация, навигация

Через графический интерфейс проходит вся информация, которую пользователь получает от системы. И здесь очень важно, как она будет структурирована для представления и в какой форме визуализирована. Формально эти вопросы не относятся к функционалу, но они крайне важны.

Запоминание. Реально человек способен эффективно манипулировать только предметами, непосредственно находящимися в поле зрения. Объем кратковременной памяти ограничен 7 ± 2 единицами неассоциированных данных, т. е. данных, при запоминании которых не возникло образных ассоциаций. При проектировании графического интерфейса необходимо свести к минимуму необходимость такого запоминания. В структуре графического интерфейса объекты разделяются по уровню их доступности и необходимости пользователю задействовать кратковременную память:

- непосредственно видимые;
- прямо доступные через видимые ассоциируемые элементы (закладки, иконки);
- выбираемые через видимые ассоциируемые элементы (выпадающие списки, всплывающие окна);
- находящиеся в известной пользователю цепочке обращений, например, открытие файла в диалоговом окне через меню File;
- находящиеся в неизвестной пользователю цепочке обращений, например, неизвестный пользователю способ настройки или параметр.

Последний пункт имеет отношение к фактору *обучения*. Остальные – к фактору *производительности* и к определенному в нем фокусу внимания.

Распределение пространства экрана. Для многих приложений графическое пространство экрана является критическим ресурсом, который надо умело распределять между отображаемыми данными. Одна из трудно решаемых проблем – *отсеивание лишней информации*. Под лишней может пониматься обнаруженная при поиске ненужная, устаревшая или более не используемая информация. Обычно в процессе работы с программой количество активных элементов (открытых окон, закладок и т. п.) постоянно увеличивается, а закрывать или удалять их приходится вручную. Средств сбора такого интерактивного мусора обычно не предусмотрено.

Интерактивный мусор – созданные в процессе работы, но не используемые более элементы GUI (окна, закладки, иконки).

Так бывает. В системе разработки открытые файлы отображаются в виде закладок со следующими правилами (рис. 6.25):

- количество видимых закладок в пределах десятка, размер закладки зависит от длины имени;
- закладки отображаются в порядке открытия файлов, повторное обращение не меняет их порядка в списке;
- для получения скрытой закладки необходимо промотать список в панели закладок или открыть полный список отдельным кликом.

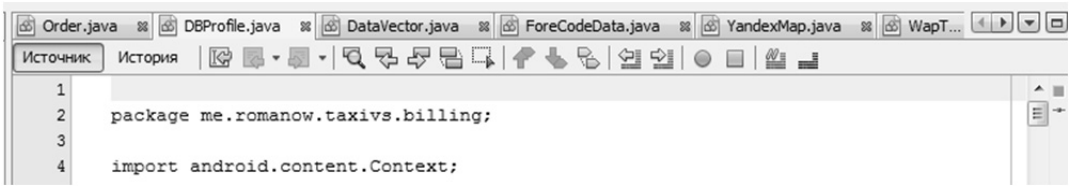


Рис. 6.25. Список открытых файлов проекта в IDE NetBeans

При постоянной работе с большим проектом приходится периодически закрывать ненужные окна. Поэтому логично было бы иметь средства, отслеживающие и закрывающие долго не используемые или однократно используемые окна. В общем случае это касается любого интерактивного мусора – его следует ограничивать и своевременно утилизировать.

Визуализация, поиск, навигация. Визуализация – представление данных пользователю в форме, удобной для восприятия, оценки и манипулирования (рис. 6.26). Пользователю нужны не данные как таковые, а их анализ для достижения некоторой цели, например, для освобождения места на диске путем удаления очень больших файлов или очистки каталогов от устаревших данных.

Стоп-кадр. Популярный кадр 1960–1970-х годов: ученые рассматривают распечатку результатов в виде «простыни» с колонками цифр.

Многообразие способов визуализации лучше всего проявляется при работе с числовыми данными. Зачастую важным является на само значение, а его качественная оценка по каким-то критериям или шкалам. Лучше использовать формат представления, соответствующий целям оценки. Рассмотрим в качестве примера вывод данных о размерах файлов. Возможные варианты:

- относительное значение, в процентах к максимальному значению, общему объему ресурса, к общей сумме значений;
- порядок значения, логарифмическая шкала с ограничением количества значащих цифр (размер файла в виде 376, 12 Кб, 186 Мб);
- превышение значения относительно порога;

- качественная шкала, размер файла изображения – крохотный, маленький, средний, большой, огромный;
- группирование малых значений, например: остальные 1478 файлов – 36 Мб.

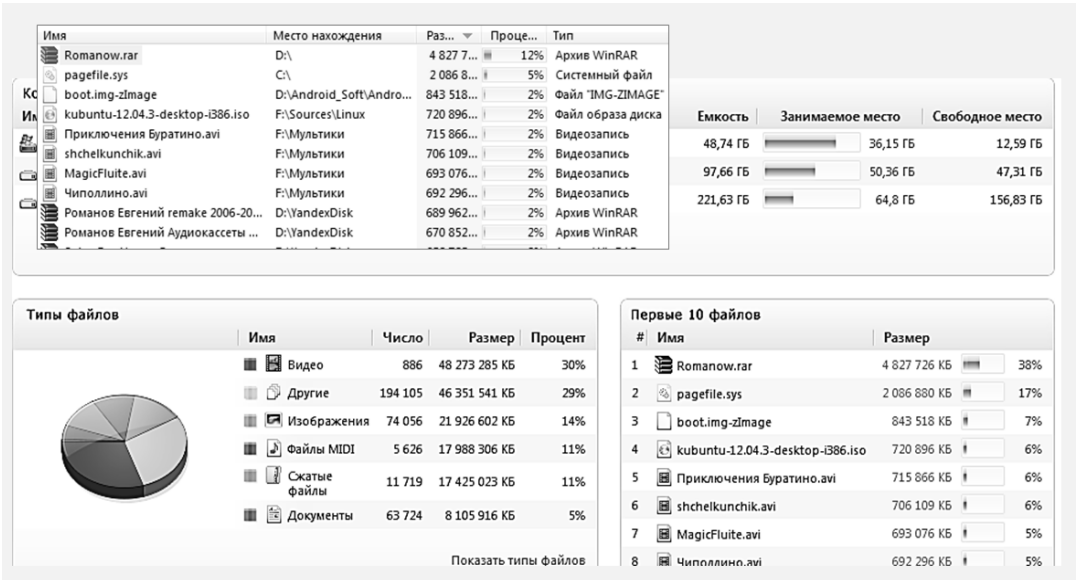


Рис. 6.26. Визуализация в TuneUp Utilities

Способ визуализации значения также может быть отличен от общепринятого:

- линейный индикатор – полоска;
- пиктограмма – количество, цвет, размер, пропорциональный значению;
- точка в системе координат – двумерные данные;
- диаграмма;
- распределение (гистограмма) – группировка для больших наборов данных;
- индикатор цветности – яркость и насыщенность, пропорциональные значению.

Замечание по теме: научная дисциплина *разведочный анализ данных* использует средства визуализации в сочетании со средствами их статистической обработки с целью выявления закономерностей, аномалий, группировки и подобного анализа, активно задействует интуицию исследователя.

Поиск и навигация. По аналогии с визуализацией следует исходить из целей, которые преследует пользователь.

Так бывает. Типичный лог для фиксации ошибок / событий от мобильных приложений (рис. 6.27):

- поле с номерами страниц лога находится внизу, так что для перехода к следующей странице приходится прокручивать страницу до конца;
- обычно поиск производится по конкретной дате или диапазону дат. По номерам страниц сложно судить о датах содержащихся в них записей. Кроме того, не все номера страниц лога выводятся в списке.

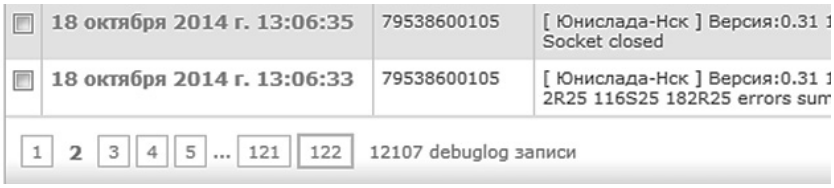


Рис. 6.27. Ошибки поиска-навигации в логе ошибок

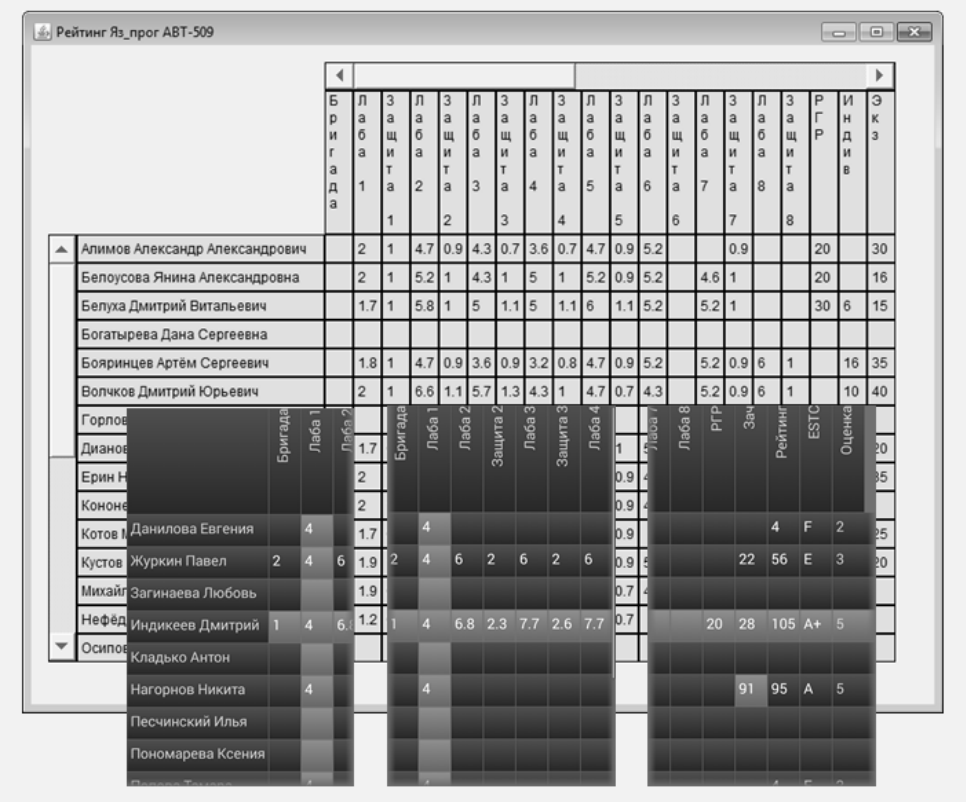


Рис. 6.28. Прокрутка таблиц с сохранением заголовков

Так бывает. При отображении больших таблиц обычно используется прокрутка всего содержимого таблицы. Это создает существенные неудобства, так как заголовки таблиц должны быть видны всегда, поскольку с ними ассоциируется остальное содержимое. В desktop-приложениях логичнее использовать способ отображения таблиц, который планирует двумерный скроллинг только для областей данных, а области заголовков прокручивает синхронно в одном измерении. В мобильных приложениях можно просто отмечать выбранные строку и столбец (рис. 6.28).

Субъективное восприятие

И наконец мы подходим к собственно внешнему виду интерфейса, графическому дизайну и особенностям его субъективного восприятия пользователем. Этот фактор включает:

- социально-психологическое восприятие – мода, «крутизна»;
- технический дизайн – сочетание эстетики с технологичностью;
- психологическое ощущение комфорта при работе.

Социально-психологические аспекты. Графический дизайн приложений сильно подвержен моде, он может быть ориентирован на определенные социальные группы, которые составляют значительную часть пользователей, зачастую должен отражать требования брендов, которые он продвигает, и т. п. Все это имеет весьма отдаленное отношение к программной инженерии, поэтому позволительно дать только одну дилетантскую рекомендацию: необходимы чувство меры и знание истории технологического дизайна. Иначе возможны ситуации, когда внешний вид графического интерфейса будет вызывать у некоторых пользователей непредвиденные ассоциации. Несколько примеров:

- ассоциация по технологическому дизайну (рис. 6.29). Модный ныне минимализм графического дизайна перекликается с технологическим минимализмом 1960-х годов – аналог хрущевок в технологии предметов домашнего обихода;
- ассоциация по цветовой палитре. Графический дизайн в мягких пастельных тонах перекликается с фотографиями 1950-х годов. Особенности последних обусловлены слабой цветовой чувствительностью тогдашней фотопленки;
- чувство меры в цветовой гамме. Многие бренды используют определенные тона или их сочетания, излишне яркие краски отвлекают от содержания и вызывают ненужные ассоциации;
- социальные аспекты. Стремление расширить аудиторию, сделать сайт более привлекательным зачастую лишает сайт информативности, ради которой он создавался (рис. 6.30).

Рис. 6.29. Ассоциация с технологическим дизайном 1960-х годов

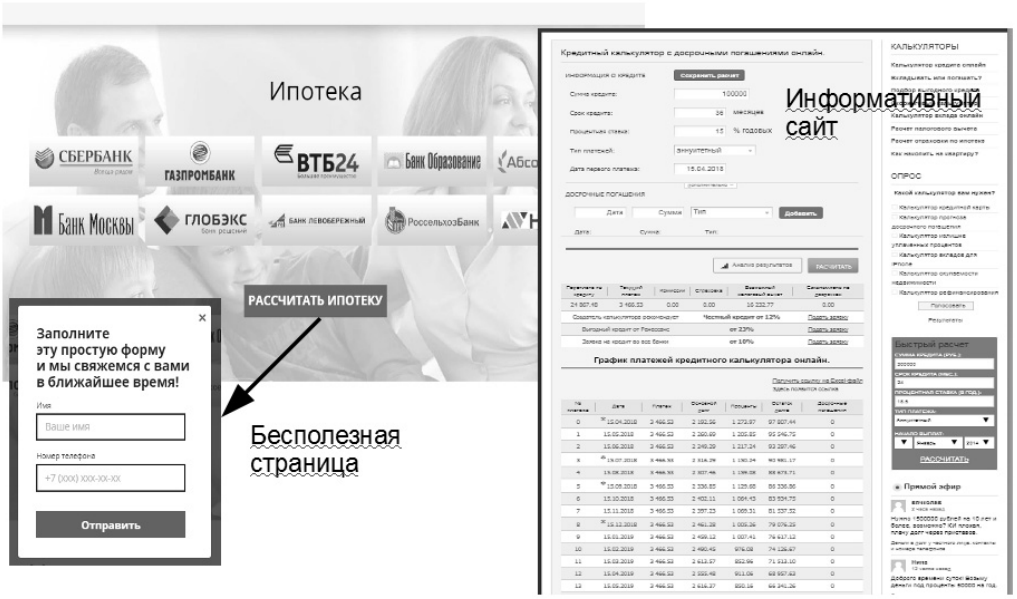


Рис. 6.30. Внешняя привлекательность в ущерб информативности

Принципы технического дизайна интерфейсов. Если социально-психологический фактор использует интерфейс как дополнительный фактор

привлечения внимания, то технический дизайн требует от него в точности обратного – он должен быть максимально незаметен, прозрачен и подчинен функционалу (рис. 6.31), а именно:

- интерфейс не самоцель, он незаметен;
- интерфейс функционален и информативен;
- интерфейс – предмет длительного использования;
- интерфейс технологичен, он обеспечивает производительную работу, минимизирует ошибки;
- интерфейс гармоничен – все элементы соизмеримы, соразмерны, выполнены в едином стиле.

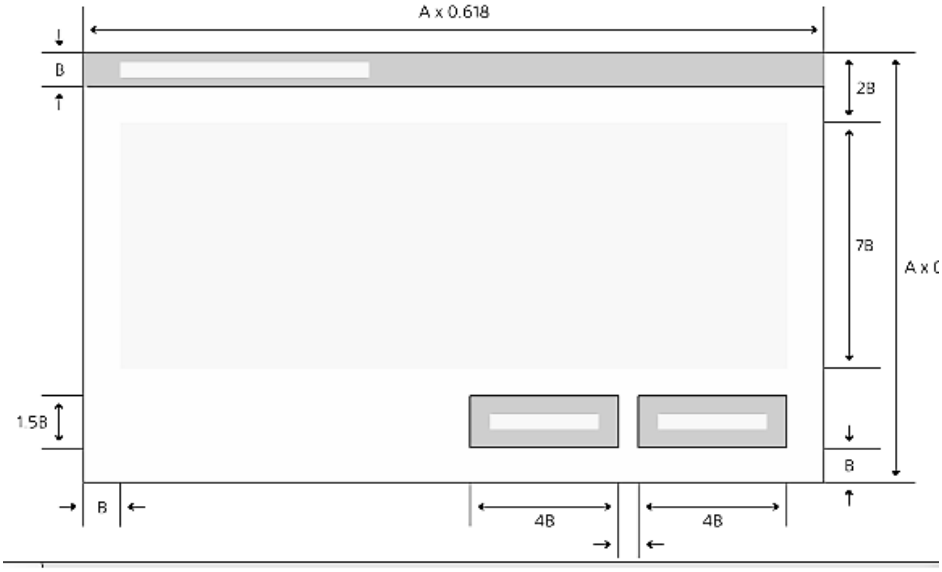


Рис. 6.31. Технический дизайн GUI: прозрачность, гармоничность

Отсюда самые простые рекомендации:

- исключение ярких цветов и острых углов;
- контраст за счет пустых пространств между элементами. По аналогии с музыкой – контраст обеспечиватся не за счет громкости, а за счет паузы и звучания;
- легкость и прозрачность интерфейса;
- исключение крикливости и навязчивости;
- визуальные закономерности: масштабная сетка, золотое сечение в пропорциях 0,618 × 0,382, пропорциональность элементов.

Субъективное ощущение комфорта. Не всегда качество графического интерфейса можно выразить измеряемыми параметрами. Субъективное ощущение дружественности интерфейса складывается в том числе из ряда психологических факторов:

- субъективное ощущение скорости работы: заполнение пауз фоновыми действиями, разбиение действий на более мелкие;
- чувство контроля над системой:
  - ограничение и контроль за потенциально опасными действиями;
  - красная / зеленая лампочка – визуализация состояния соединений, доступности ресурсов;
  - корректная система контроля ошибок: предупреждение ошибок, указание границ действий, выбор вместо ввода, исправление в процессе ввода, сообщения о причине ошибки и способах исправления;
- самовыражение – возможность персональной настройки программы;
- разумная система идентификации и защиты: выбор запомненного логина вместо ввода, выпадающий список логинов, ввод пароля открытым текстом, запоминание пароля на ограниченный срок. Обычно системы защиты вносят значительный дискомфорт, так как действия по предотвращению потенциального вреда не воспринимаются как необходимые, сложные пароли следует запоминать или хранить отдельно и т. п.

Влияние GUI на процесс разработки. Проектирование от GUI

Как уже было сказано, в унифицированном процессе проектирования разработка графическогно интерфейса – деятельность второго плана в процессах функционального проектирования. Первичными являются варианты использования, сценарии и требования как способы описания функционала.

Соответствие между функционалом и графическим интерфейсом не является простым отображением элементов первого на компоненты второго. Например, сценарий может быть реализован как последовательность вызывающих друг друга диалогов либо как действия над списками и кнопками в одном окне в более или менее свободном порядке. Требование визуализации состояния может быть реализовано в виде всплывающего сообщения об изменении состояния либо в виде постоянного индикатора в основном окне.

Рассмотрим несколько типовых вариантов организации графического интерфейса с точки зрения их связи с функционалом. В первом варианте (рис. 6.32) мы имеем дело с набором инструментов, которые можно применять к объекту предметной области, в данном случае – это изображение.

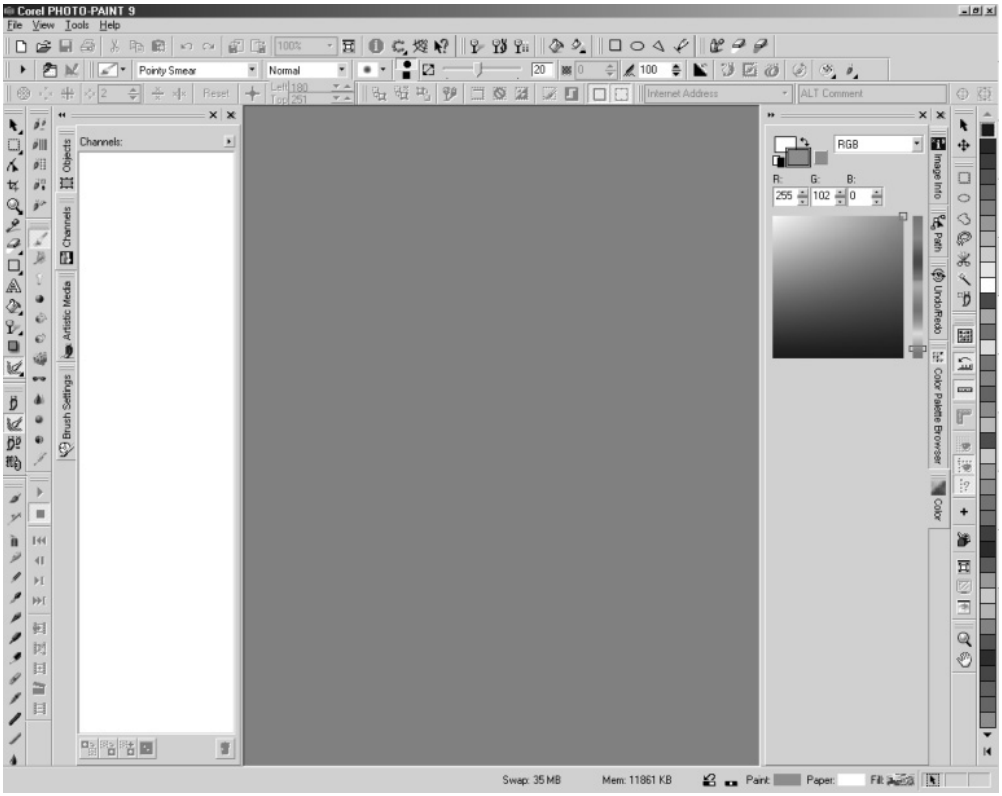


Рис. 6.32. Интерфейс «набор инструментов»

Во втором случае предметная область представлена в виде более сложной модели (рис. 6.33). Например, звуковой редактор работает с треками, в каждый из которых может быть загружен один или несколько звуковых файлов. Для каждого из них устанавливается цепочка эффектов обработки звука и набор параметров – громкость, панорама. Результат микшируется в отдельный трек с соответствующей постобработкой.

Графический интерфейс здесь представляет систему окон, в которые проецируется текущее состояние отдельных компонент предметной области или производится управление ими.

В приложениях, не рассчитанных на широкую публику, и в программных прототипах, где нет смысла в отдельном проектировании графического интерфейса, уместен вариант типа «кабина самолета» (рис. 6.34), где все элементы управления находятся в одном окне и одинаково доступны, а сценарии пользователь держит в голове.

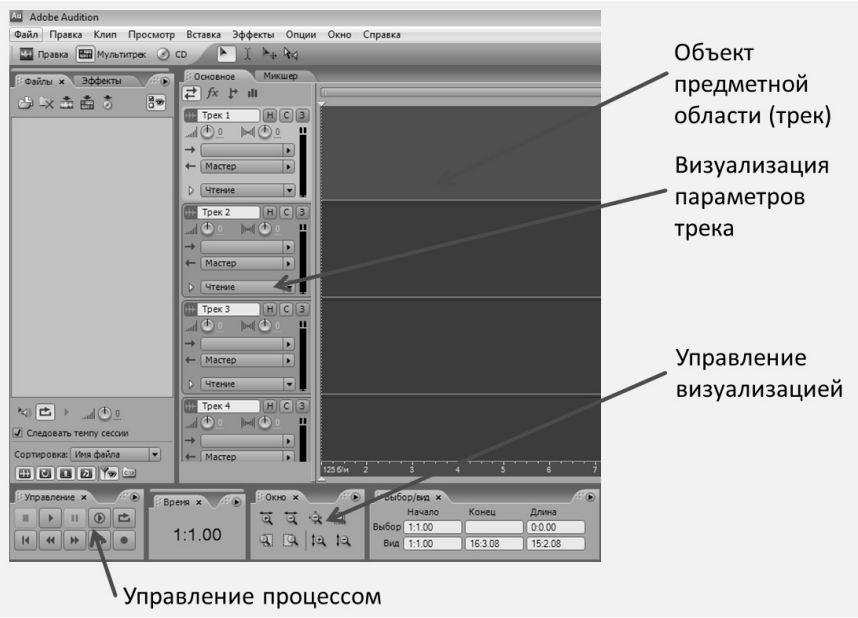


Рис. 6.33. Интерфейс «система окон над предметной областью»

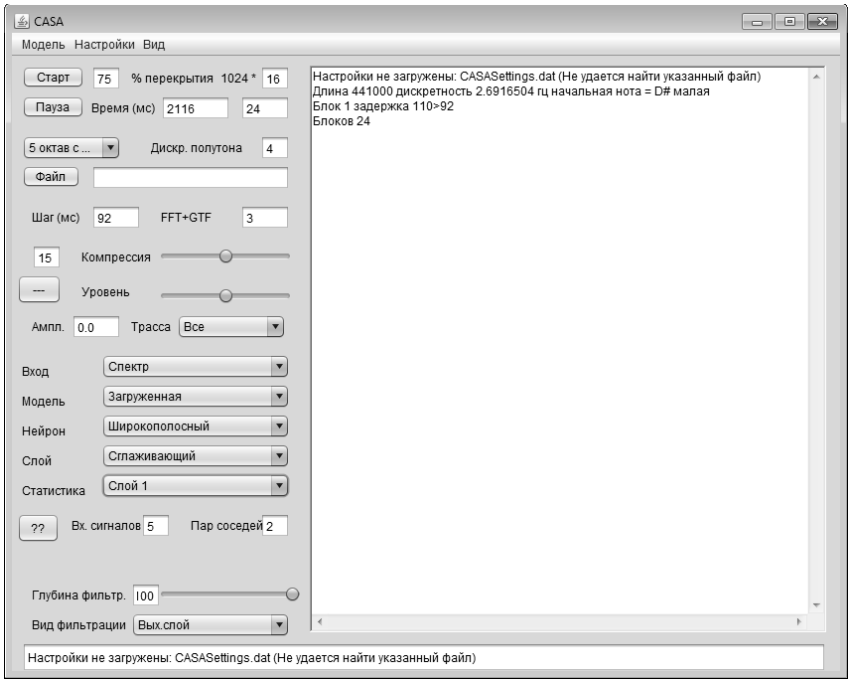


Рис. 6.34. Интерфейс «кабина самолета»

В мобильных приложениях из-за ограниченности размера экрана сценарии, предписанные функционалом, поддерживаются цепочкой окон, в каждом из которых выполняется одно или несколько действий и находится ограниченное количество элементов управления (рис. 6.35).

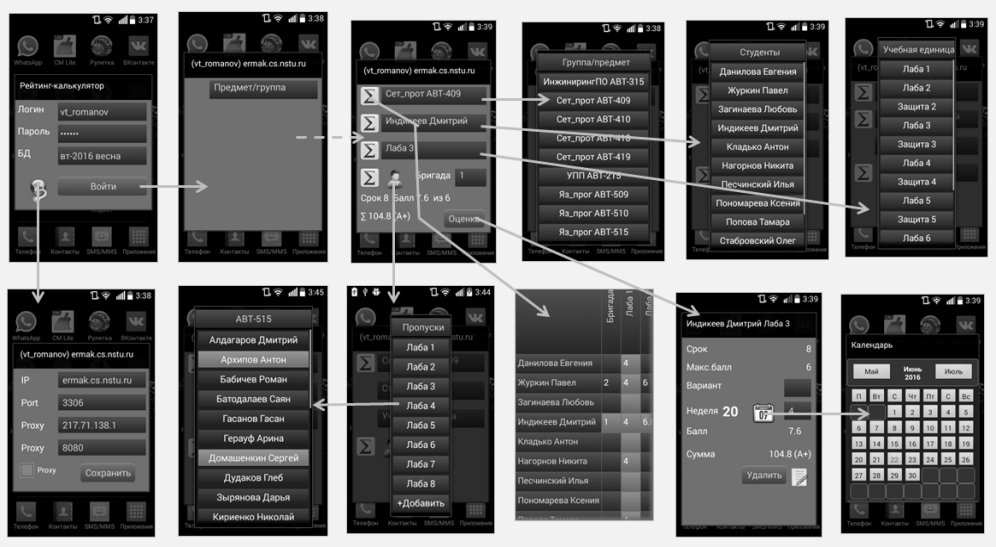


Рис. 6.35. Интерфейс «сценарии, ведомые интерфейсом»

Ввиду таких достаточно запутанных соотношений между графическим интерфейсом и функционалом графический интерфейс не может полностью заменить функциональное описание. Тем не менее есть несколько принципиальных мнений разработчиков по этому поводу:

- прототип графического интерфейса в виде набора экранов и есть описание функционала (заблуждение);
- роль графического интерфейса настолько важна для успеха проекта, что его нельзя прицеплять к готовому функционалу;
- характер взаимоотношений между предметной областью, функционалом и графическим интерфейсом позволяет рассматривать последний как одну из спецификаций функционала. Приведенные выше интерфейсы – *сценарий, ведомый интерфейсом и система окон над предметной областью* – являются иллюстрацией этого положения;
- опытные разработчики графического интерфейса фактически выполняют функциональное проектирование системы. Особенно такое мнение распространено у разработчиков web-приложений и сайтов.

Вариант разработки функционала системы *от графического интерфейса* [29] не отвергает основные элементы бизнес-аналитики и системной аналитики. Первоначально на каждый функциональный модуль системы (единицу поведения) создается отдельная модель. В качестве инструментов описания бизнес-процессов и проектирования могут использоваться различные модели (рис. 6.36): диаграммы потоков данных (DFD), UML-диаграммы деятельности и устойчивости. Собственно на этапе проектирования применяются неканонические диаграммы объектов графического интерфейса – окон, диалогов и переходов между ними или канонические UML-диаграммы оконных классов.

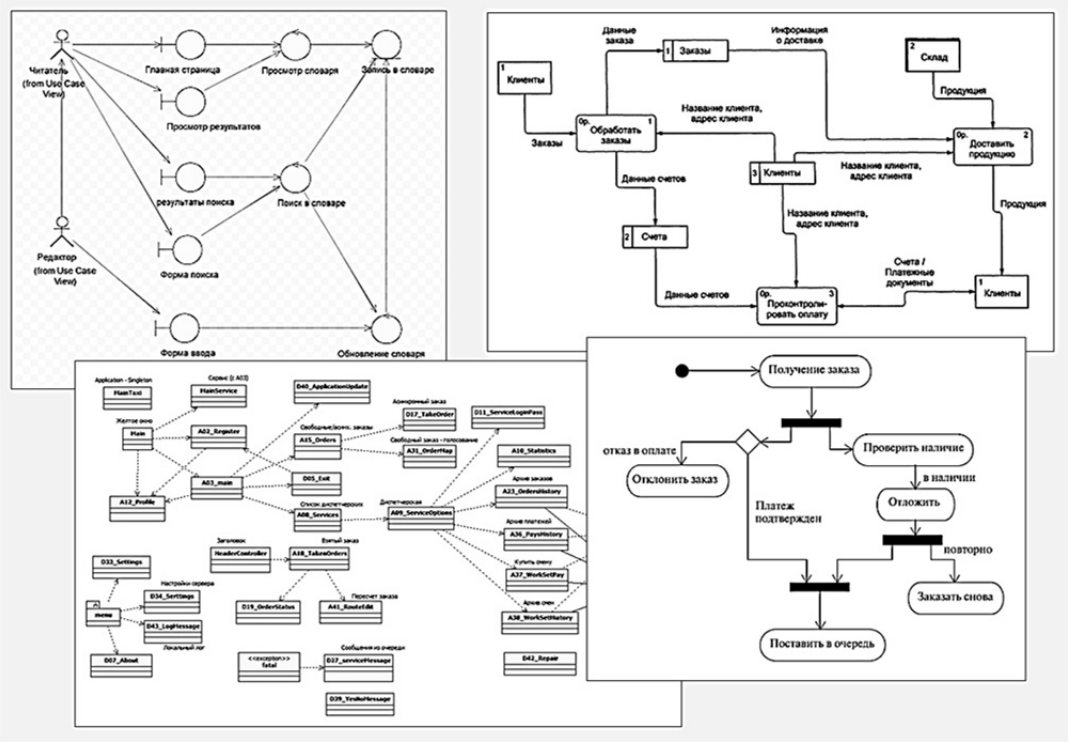


Рис. 6.36. Модели, используемые при проектировании «от графического интерфейса»

Проектирование включает следующие этапы:

- анализ функциональности системы, словесное или формальное описание бизнес-процессов и системной аналитики без привязки к графическому интерфейсу;

- разработка структурной схемы графического интерфейса с привязкой элементов бизнес-процессов к ее компонентам – экранам, всплывающим сообщениям, диалоговым окнам, строкам состояния (рис. 6.37);
- верификация и модификация структурной схемы на основе верификации и анализа бизнес-процессов;
- детальное проектирование отдельных прототипов графического интерфейса, например, с использованием метрики скорости работы (GOMS);
- разработка глоссария и согласование прототипов;
- сведение всех прототипов в общую модель (рис. 6.38).

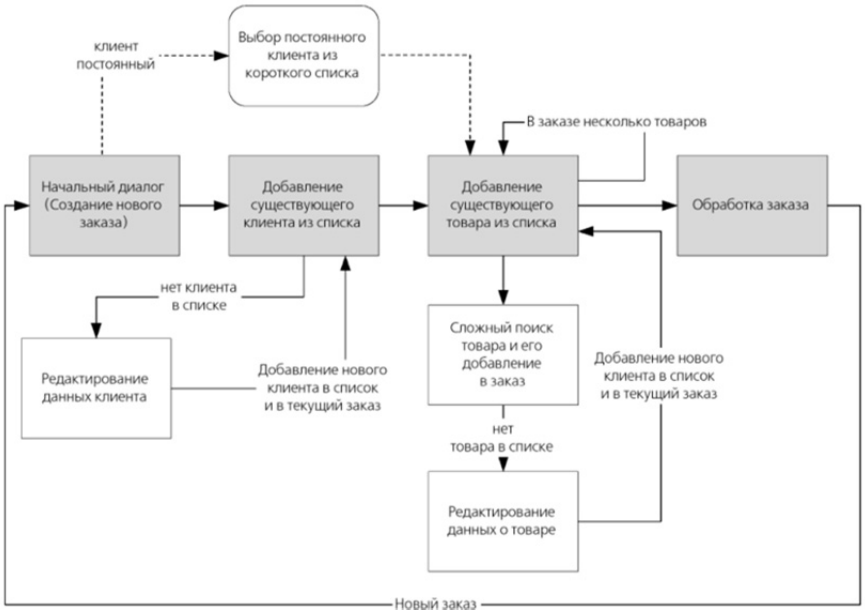


Рис. 6.37. Структурная схема графического интерфейса [29]

Замечание по теме. Проектирование «от графического интерфейса» не касается архитектурных вопросов, т. е. выступает только как замена бизнес-аналитики и системной аналитики.

Заключение. «Приятные мелочи» интерфейса пользователя

Некоторые «приятные мелочи» способны вывести из себя даже весьма флегматичного пользователя. В перечисленном списке кое-что является результатом незнания способов настройки приложений, кое-что представляет собой программные ошибки, кое-что является иллюстрацией несоблюдения

перечисленных выше рекомендаций, кое-что появилось в результате реинжиниринга – не все пункты в старом меню попали в новые пиктограммы, кое-что унаследовано из старой архитектуры приложений:

- при установке курсора в поле ввода переключатель языка клавиатуры оказывается в положении, противоположном необходимому, вы печатаете «по-английски» – ds gtxfnftnt gj fyukbqcrb;

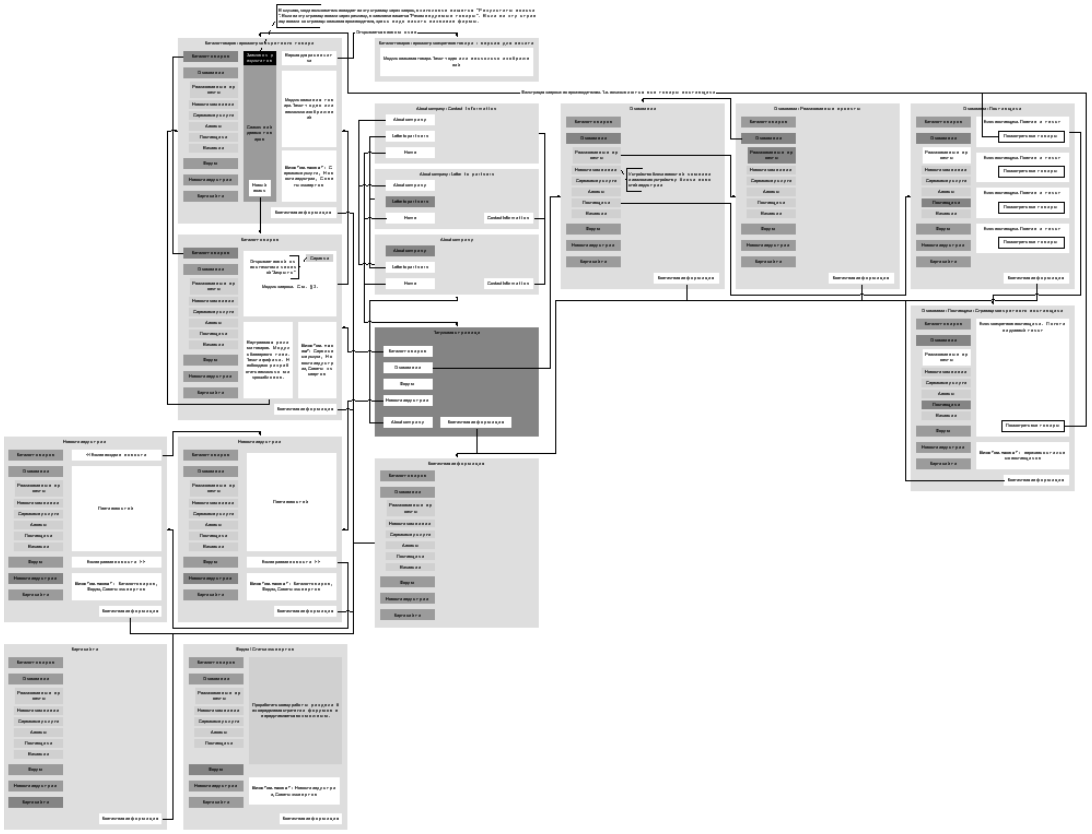


Рис. 6.38. Итоговая модель графического интерфейса [29]

- при запуске приложения-форматера по команде печать выводится запрос на обновление приложения;
- при манипуляциях с блоком текста меняется шрифт или форматирование соседних участков;
- для попадания на нужный элемент в линейке форматирования страницы необходима точная подгонка курсора на несколько пикселов, элементы интерфейса расположенны очень близко;

- при выборе команд download предлагается установить специальное приложение;
- в MS Vilual Studio для продолжения редактирования необходимо остановить режим отладки специальной командой. В других IDE как-то без этого обходятся, новая компиляция и сборка приложения просто отменяют текущий режим отладки;
- в MS Word2010 в меню «разметка страницы» имеется группа «ориентация», две кнопки которого меняют ориентацию документа в целом. Для изменения ориентации одной страницы необходимо в том же меню выбрать в группе «поля» пункт «настраиваемые поля», далее в форме с полным набором параметров для полей и ориентации выбрать пункт списка «применить к выделенному фрагменту».

6.6. Управление конфигурациями и сопровождение

Управление конфигурациями и сопровождение ПО – близкие по духу процессы: оба являются вспомогательными, оба ассоциируются в основном с постпроектным этапом жизненного цикла, оба исходят из того, что код проекта, его артефакты и окружение находятся в процессе эволюции.

Управление конфигурациями

Управление конфигурациями – единый процесс, имеющий три независимые проекции (измерения): конфигурационное управление, управление изменениями и метрика изменений. Используется термин SCCM (Software Configuration and Change Management) (рис. 6.39).

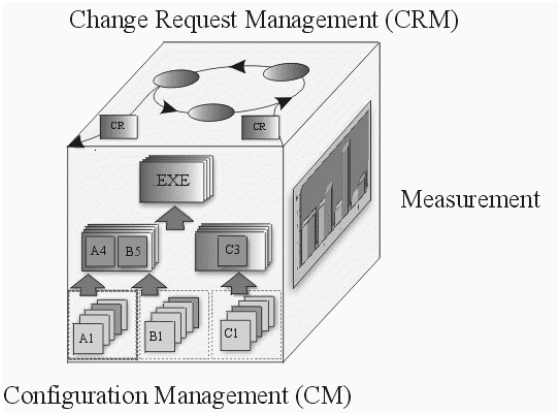


Рис. 6.39. Три измерения SCCM

Управление конфигурацией исходит из того, что любая программная система может собираться из разных версий программных компонент в разных комбинациях. Например, установка системы может производиться в минимальной, стандартной, полной и заказной конфигурациях. Основные принципы конфигурационного управления:

- три уровня артефактов конфигурации – компоненты, подсистемы, система;
- каждый артефакт существует в виде дерева версии;
- между артефактами имеются зависимости, связанные с использованием одними артефактами других;
- артефактами конфигурации являются также документы – артефакты проектирования, версии среды разработки и окружения – библиотек, СУБД, протоколов;
- задача конфигурационного управления – обеспечение целостности системы артефактов и правильности сборки конфигурации из различных версий.

Управление запросами на внесение изменений. Артефакты конфигурации изменяются по различным причинам: исправление дефекта, обнаруженного при тестировании или эксплуатации, улучшение качества, внесение изменений в требования, документирование отличий между версиями. Это создает артефакт конфигурационного управления – *запрос на внесение изменений.* Он имеет свой жизненный цикл, включающий последовательную смену состояний запроса: новый, зафиксированный, утвержденный, принятый, завершенный.

В процессе жизненного цикла запрос накапливает связанные данные: затрагиваемые артефакты, влияние на архитектуру, трудоемкость (стоимость), график реализации. В зависимости от глубины вносимых изменений он влияет на технологические процессы – управление требованиями, анализ и проектирование (изменение классов, структуры БД, архитектуры), конструирование и тестирование).

Замечание по теме. В технологическом процессе управления требованиями существует похожая деятельность – *управление жизненным циклом требований* (см. раздел 6.4). Она относится в большей степени к разработке, а конфигурационное управление в целом – к эксплуатации системы.

Третьей составляющей является **метрика процессов внесения изменений**. Сюда входят метрические характеристики разработки – прогресс проекта относительно изменений, количество изменений в различных состояниях, возраст изменений, распределение изменений по серьезности, приоритету, степени влияния на архитектуру, причинам появления.

Сопровождение

Многие программные системы эксплуатируются как постоянно действующие сервисы в значительном количестве экземпляров. В связи с этим сопровождение становится одним из основных технологических процессов в жизненном цикле. Преимущества проектов с отрытым кодом (Open Source) теряются из-за отсутствия их сопровождения.

Работы по сопровождению проводятся для решения следующих задач:

- устранение дефектов;
- улучшение дизайна;
- реализация дополнительного функционала;
- создание интерфейсов взаимодействия с другими системами;
- адаптация для возможности работы на другой аппаратной платформе, для применения новых системных возможностей, для функционирования в среде обновленной телекоммуникационной инфраструктуры и т. п.;
- миграция унаследованного (legacy) программного обеспечения;
- вывод программного обеспечения из эксплуатации.

Деятельность персонала сопровождения включает четыре ключевых аспекта:

- поддержка управляемости программного обеспечения в течение всего цикла эксплуатации;
- поддержка модификаций программных систем;
- совершенствование существующих функций;
- предотвращение падения производительности программной системы до неприемлемого уровня при росте потоков запросов, масштабирование.

Современное сопровождение связано не столько с устранением дефектов, сколько с *эволюцией программного обеспечения* в процессе эксплуатации и поддержкой среды функционирования для массового пользователя, в том числе конфигурированием, восстановлением, адаптацией.

Существуют четыре категории сопровождения:

- корректирующее сопровождение (corrective maintenance): быстрая модификация программного продукта, выполняемая уже после передачи в эксплуатацию для устранения сбоев;
- адаптирующее сопровождение (adaptive maintenance): модификация программного продукта на этапе эксплуатации для обеспечения продолжения его использования с заданной эффективностью в изменившемся окружении;
- совершенствующее сопровождение (perfective maintenance): модификация программного продукта на этапе эксплуатации для повышения характеристик производительности и удобства сопровождения;

- профилактическое сопровождение (preventive maintenance): модификация программного продукта на этапе эксплуатации для идентификации и предотвращения скрытых дефектов до того, как они приведут к реальным сбоям.

Сопровождением не обязательно занимаются непосредственные разработчики и организации-разработчики. Это вызывает следующие проблемы:

- ограниченное понимание (Limited understanding) – затраты на анализ стороннего кода для его понимания с целью внесения изменений;
- различные вопросы, в том числе и правовые, по поводу доступности исходных текстов и документации сопровождаемого проекта;
- затраты на тестирование внесенных изменений;
- анализ влияния и возможных последствий изменений, вносимых в существующую систему.

Так бывает. Для небольших и средних проектов типична ситуация прекращения сопровождения в связи с увольнением разработчиков, отсутствием документации, прекращением деятельности организации-разработчика, большими затратами на обратный инжиниринг проекта и т. п.

6.7. Управление программным проектом

«Такую личную неприязнь я испытываю к потерпевшему, что кушать не могу…»

Из кинофильма «Мимино»

Меньше всего хотелось бы упоминать в управлении программными проектами о руководстве и эффективном менеджменте, помятуя известную народную мудрость – не умеешь работать, иди руководить. Поэтому возьмем за основу термин, который имеет идентичный перевод в данном контексте: *management* и *control* переводятся как управление, только первый подразумевает больше организационные меры, а второй акцентируется на технологическом аспекте. Мы будем рассматривать проблему с позиций «ситуация под контролем».

Своды знаний и стандарты

Начать придется с менеджмента. Управление программным проектом – это проектная деятельность, а управление проектами – **Project Management** (**PM**) – стандартизованная категория менеджмента. В ней, аналогично

программной инженерии, существует свой «Свод знаний по управлению проектами» (Project Management Body of Knowledge – PMBOK) [37], охватывающий следующий перечень деятельностей:

- управление интеграцией проекта (project integration management);
- управление содержанием проекта (project scope management);
- управление сроками проекта (project time management);
- управление стоимостью проекта (project cost management);
- управление качеством проекта (project quality management);
- управление человеческими ресурсами (project human resource management);
- управление коммуникациями проекта (project communication management);
- управление рисками проекта (project risk management);
- управление поставками проекта (project procurement management).

Баня, музыка, кино и мост. Проектная деятельность в разных областях имеет свою специфику. Для программной инженерии наиболее удачные аналогии и метафоры можно найти там, где результат имеет нематериальный характер, например в кинопроизводстве или постановке спектакля.

В программной инженерии вопросы управления программными проектами отражены в отдельном документе SWEBOK «Управление программной инженерией» (Software Engineering Management). Основные разделы идентичны стандарту IEEE (ISO/IEC) 12207 в части «Процесс управления» (Management Process):

- инициирование и определение содержания;
- планирование программного проекта;
- выполнение программного проекта;
- обзор и оценка;
- закрытие проекта;
- измерения в программной инженерии.

Кроме того, отдельные вопросы управления проектом обсуждаются в других документах SWEBOK:

- требования к программному обеспечению (Software Requirements);
- конфигурационное управление (Software Configuration Management);
- процесс программной инженерии (Software Engineering Process);
- качество ПО (Software Quality).

Специфика управления программным проектом

Если же отвлечься от сухих стандартов, то для поддержания ситуации под контролем в любом проекте необходимы:

- организация команды исполнителей;
- планирование работ;
- оценка сроков исполнения, стоимости, объема работ для будущего периода, учет для прошедшего периода – метрика проекта;
- оценка потенциальных угроз проекту – рисков.

Этим, собственно, и занимается Project Management. В программной инженерии необходимы поправки на специфику отрасли:

- специфические риски, связанные с качеством персонала, оценками трудозатрат, адекватностью представлений и требований;
- различные принципы организации и самоорганизации исполнителей, широкий спектр методологий, в каждой из которых имеется собственная структура менеджмента;
- особенности метрик программного продукта, метрик качества кода;
- особенности и структура жизненного цикла ПО.

Особенности программной инженерии как проектной деятельности были отмечены в разделе 1.2. Все это имеет прямое отношение к управлению проектом. Существует также связь методологии разработки с основными метрическими характеристиками проекта, учитываемыми в управлении [41] (рис. 6.40).

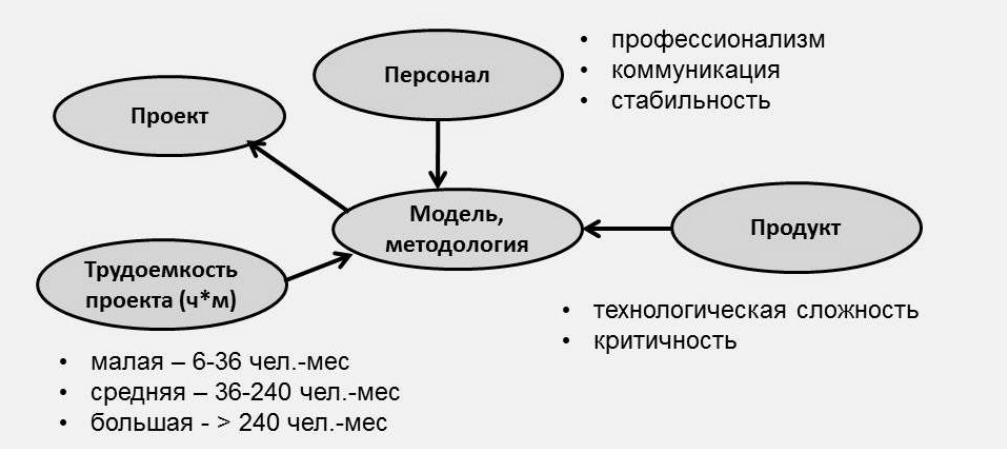


Рис. 6.40. Факторы, влияющие на выбор методологии разработки проекта

Под *тяжестью* методологии понимается количество элементов управления – артефактов, деятельностей, под *плотностью* – степень их детализации и связности с другими элементами. Выделены следующие зависимости:

- чем больше команда, тем тяжелее используемая методология;
- плотность методологии пропорциональна критичности проекта;
- чем тяжелее методология, тем выше стоимость проекта;
- самая эффективная форма коммуникации – непосредственное общение.

Методология и организационная структура компании

Программный проект и коллектив исполнителей существуют в рамках компании, в организационную структуру которой они вписываются. Традиционно наиболее распространенной формой является *функциональная структура*, т. е. подразделения, ориентированные на функции или на исполняемые виды работ (руководство, финансовое планирование, ресурсное обеспечение, снабжение, инфраструктура, web-разработка, клиентские приложения (рис. 6.41)).

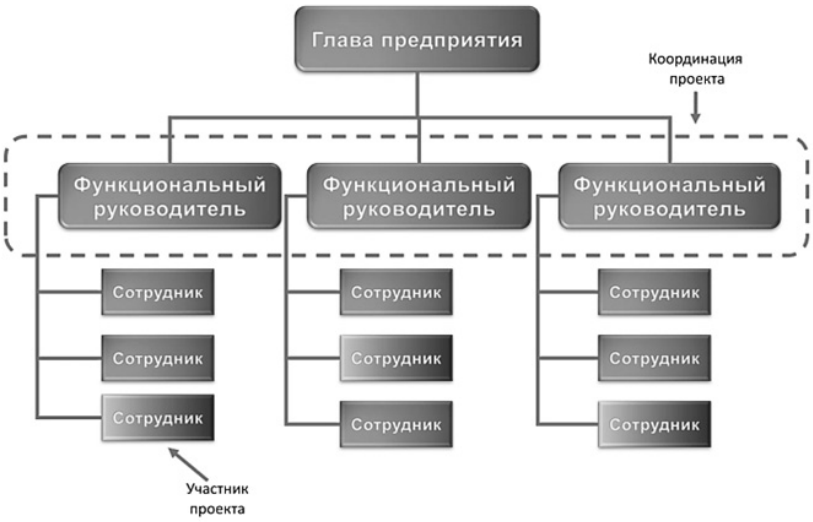


Рис. 6.41. Функциональная структура компании [35]

Такая структура обеспечивает преемственность опыта разработчиков, рассчитана на стабильные отношения, однако не способствует коммуникациям в проекте и его управляемости. Альтернативой является *проектная структура*, в которой структурные подразделения создаются под проект (рис. 6.42).

Рис. 6.42. Проектная структура компании [35]

Она в большей степени соответствует требованиям отдельного проекта в части коммуникаций и управления, но не обеспечивает стабильности организации и преемственности опыта разработки.

Как у других. Большинство репертуарных театров имеет функциональную структуру: в оперном театре – это дирекция, отдел распространения, бутафорский и костюмерный цеха, хор, оркестр, балетная труппа, оперная труппа. Проект (спектакль) не имеет собственной самостоятельной организационной структуры, все его финансовые и имиджевые показатели включаются в общий котел. Проектной формой театральной деятельности является *антреприза,* в которой актеры набираются под отдельный проект, имеющий самостоятельное финансирование и организационную структуру.

Матричная структура пытается в разных вариантах совместить интересы проекта и предприятия. Функциональная структура предприятия сохраняется, но степень независимости проекта может быть разной:

- слабая матрица – координатор проекта выполняет вспомогательные функции мониторинга, а управление проектом закреплено за функциональным руководителем;
- сбалансированная матрица – менеджер проекта и функциональный руководитель имеют примерно равные полномочия, возникает двойное подчинение исполнителя в рамках проекта и отдела;
- сильная матрица – при сохранении функциональной организационной структуры исполнители организованы в проектные команды, имеется отдельное подразделение менеджеров проекта.

Гибкие методологии требуют отдельных организационных решений. В разделе 5.2 уже рассматривалось масштабирование Scrum для крупных проектов в двух аспектах:

- масштабирование структуры исполнения проекта – объединение лидеров команд;
- масштабирование планирования проекта – объединение бэклогов и владельцов продуктов.

Для гибких технологий наиболее приемлемы:

- проектная структура организации;
- отдельная команда как самостоятельная хозяйственная единица;
- матричная структура организации – сильная матрица SCRUM of SCRUM;
- специфичные приемы планирования и самоорганизации для команды, метрик и планирования.

Фаза исследования. Скорость принятия решения

> Скорость принятия решения – наибольшая скорость разбега многодвигательного самолета, при которой в случае отказа двигателя возможно как безопасное прекращение, так и безопасное продолжение взлета.
>
> *Энциклопедия «Авиация»*

В какой-то момент обсуждения проекта должно быть принято ключевое решение «взлетаем». Естественно, оно принимается руководством на основании проверенных данных. В унифицированном процессе для этих целей выделена первая фаза исследования, результатом которой является набор артефактов, на их основе принимается указанное решение (см. рис. 5.3).

Главными технологическими процессами этой фазы являются *управление требованиями* и *управление проектом*. Выделим в структуре фазы исследования (раздел 5.1) компоненты, имеющие прямое или косвенное отношение к управлению проектом.

Цели фазы исследования:

- область применимости, граничные условия, видение проекта;
- предварительная оценка стоимости;
- предварительная оценка рисков.

Артефакты фазы исследования:

- документ видения;

- начальный бизнес-план;
- начальная оценка рисков;
- план проекта – фазы и итерации.

Деятельности фазы исследования:

- формулировка важнейших требований – области действия проекта и ограничений, т. е. рамок проекта;
- разработка бизнес-плана – оценка рисков, кадровое обеспечение, цена, график работ, рентабельность.

Риск – не благородное дело, а фактор разработки

Риск – неопределенное событие или условие, наступление которого отрицательно или положительно сказывается на целях проекта. Как правило, в случае возникновения негативного риска почти всегда стоимость проекта увеличивается и происходит задержка в выполнении расписания проекта.

Приведенное определение ничего не говорит о сущности самих событий. Здесь сразу надо определить жесткую границу. Если негативным событием нельзя управлять, то оно относится к *непрогнозируемым рискам,* единственной защитой от которых является создание резерва проекта или его страхование. Управляемые (прогнозируемые) риски также ограничиваются технологической и организационной сферами проекта. Таким образом, рисками становятся обычные повседневные недостатки организации проекта, если с ними не мириться, а пытаться их контролировать. В доказательство перечислим наиболее распространенные риски по мнению авторов:

Риски программного проекта по Б. Боэму [5]:

- дефицит специалистов;
- нереалистичные сроки и бюджет;
- реализация несоответствующей функциональности;
- разработка неправильного пользовательского интерфейса;
- золотая сервировка, перфекционизм, ненужная оптимизация;
- непрекращающийся поток изменений;
- нехватка информации о внешних компонентах окружения системы;
- недостатки в работах, выполняемых сторонними организациями;
- недостаточная производительность получаемой системы;
- разрыв в квалификации специалистов разных областей знаний.

Наиболее существенные риски по Т. де Демарко и Т. Листеру [45]:

- изъяны календарного планирования;
- текучесть кадров;

- раздувание требований;
- нарушение спецификаций;
- низкая производительность.

Наиболее существенные риски по С. Архипенкову [35]:

- требования заказчика отсутствуют, подвержены частым изменениям;
- отсутствие необходимых ресурсов и опыта;
- отсутствие рабочего взаимодействия с заказчиком;
- неполнота планирования, забытые работы;
- ошибки в оценках трудоемкостей и сроков работ.

Обычно при первоначальной оценке объемов и сроков проекта обращают внимание на основную функциональность, в результате за бортом оказываются важный функционал и работы, связанные со вспомогательными технологическими процессами. Наиболее часто выпадают из первоначальной оценки:

- функциональные требования: программы установки и настройки, конфигурирование, миграция данных, интерфейсы с внешними системами, справочная система и документация;
- общесистемные требования: обеспечение производительности, надежности, открытости, масштабируемости, безопасности, кроссплатформенности, эргономичности;
- деятельности: обучение, координация работ, уточнение требований, управление конфигурациями, управление версиями, автосборка, разработка автотестов, создание тестовых данных, обработка запросов на изменения;
- накладные расходы: сопровождение действующих систем, повышение квалификации, участие в подготовке технико-коммерческих предложений, участие в презентациях, административная работа, отпуска, праздники, больничные.

Резюмируя перечисленное многообразие, можно определить типичные места в процессе разработки, являющиеся источниками рисков:

- управление требованиями, функционал системы;
- планирование и оценка работ, объемов и сроков;
- связь с внешними системами и информация о них;
- квалификация, опыт и взаимодействие исполнителей.

Для того чтобы риском управлять, его необходимо определить *как артефакт жизненного цикла* со своим набором характеристик, например:

- причина или источник;
- симптомы риска;
- вероятность наступления риска;

- последствия риска, его тяжесть – влияние риска на возможность достижения целей проекта и на его основные параметры (стоимость, график, технические характеристики продукта).

Качественный анализ рисков

Поскольку большинство характеристик риска сложно определить количественно, остается экспертный качественный подход по шкале «низкий–средний–высокий». По этой шкале можно оценить две основные характеристики риска – *вероятность наступления* и *тяжесть последствий* (*воздействий*). Обобщенный *показатель риска* можно определить как произведение *вероятности на воздействие*, переведя качественные оценки в числовые значения. Тогда получим:

- исходные характеристики: 1 – низкий, 2 – средний, 3 – высокий;
- показатель риска: 1–2 – низкий, 3–4 – средний, 6–9 – высокий.

Управление рисками

Управление рисками является отдельной деятельностью в управлении программным проектом. Как и любая деятельность, она имеет свой план, в котором определяются:

- подходы (методология), инструменты и источники данных – метрика проекта;
- персоналии и ответственность;
- ресурсное обеспечение деятельности – учет требуемых ресурсов в базовом плане по стоимости проекта;
- процесс – виды операций, их частота и привязка к расписанию проекта;
- категории и классификация рисков, процедура идентификации;
- оценка вероятностей и степени влияния на параметры проекта – стоимость, сроки исполнения;
- источники данных о рисках: внутренние – статистика организации, экспертные оценки; внешние – отраслевая статистика, аналитика.

Результатом является определение варианта реагирования на каждый идентифицированный риск:

- уклонение от риска – изменение плана работ с целью исключения риска, например, приобретение стороннего ПО, привлечение специалиста;
- передача риска – передача ответственности за наступление риска на другую сторону, например, страхование ответственности или передача заказчику по условиям договора;

- снижение рисков – снижение вероятности наступления риска или его возможных последствий, попытки снижения имеющейся неопределенности на более ранних стадиях проекта;
- принятие риска – пресловутое авось.

В целом стратегия проста: если риск не удается снизить административным путем, переложив ответственность, то необходимо попытаться уклониться от него. Если не получается, то как можно раньше снизить связанную с ним неопределенность, чтобы внести необходимые коррективы в оценку проекта.

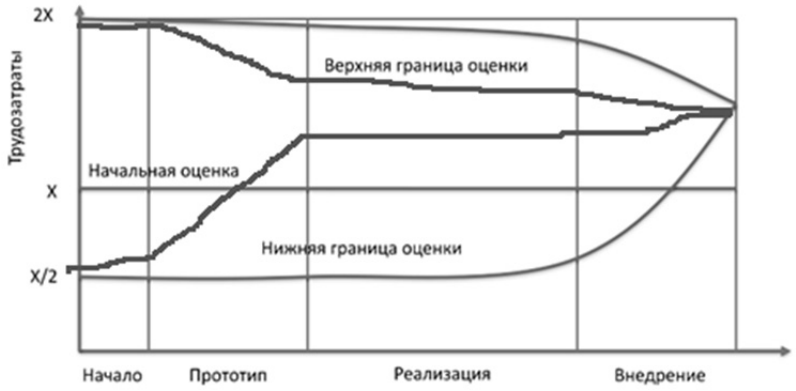


Рис. 6.44. Влияние прототипирования на оценку затрат [35]

Инструментом снижения рисков является прототипирование (рис.6.45), которое обеспечивает:

- снижение рисков по неопределенности требований путем использования функционального прототипа или макета;
- снижение архитектурных рисков и идентификация узких мест путем использования прототипа базовой архитектуры.

Планирование проекта

Начальный этап планирования проекта включает:

- уточнение содержания и состава работ. Результирующий артефакт – развернутое содержание в виде *иерархической структуры работ* (ИСР) (Work Breakdown Structure – WBS);
- планирование организационной структуры, формирование команды;
- базовое расписание проекта – компоненты ИСР расписываются по срокам с учетом их взаимосвязей, занятости исполнителей и прочих внешних и внутренних факторов, например сроков поставки оборудования.

Для внутреннего планирования проекта необходимо определить организационные структуры и регламенты для поддержки следующих видов деятельности:

- планирование управления содержанием – определение процедур работы с изменениями, вносимыми в проект в технологическом процессе управления требованиями:
  - классификация объектов изменений: требования, архитектура, форматы и структуры данных, исходные коды, сценарии тестирования, документация;
  - определение источников запросов на изменение;
  - разработка порядка анализа, оценки и утверждения или отклонения изменения содержания;
  - определение порядка документирования изменений содержания;
- планирование управления конфигурациями:
  - хранение исходного кода;
  - управление версиями;
  - поддержание инфраструктуры;
  - администрирование БД;
  - управление документами;
- планирование управления качеством:
  - принятие стандартов качества программного кода и документации – стилистические и технологические требования;
  - мониторинг качества: определение отклонений по качеству, причин и мер по их устранению, контроль исполнения;
  - аудит качества: независимая информация о несоответствиях, не устраняемых на уровне проекта.

Средством визуализации базового расписания проекта является диаграмма Ганта (рис. 6.45) – графическое представление последовательности работ с учетом их причинно-следственной связи и распределением по исполнителям. Срок исполнения проекта по диаграмме Ганта ограничивает *критический путь* – самая длинная цепочка зависимых работ.

При организации взаимосвязанных работ нужно учитывать вероятностный характер планирования [48] (рис. 6.46). Если срок исполнения работы является случайной величиной, то срок исполнения проекта не будет равен простой сумме средних значений времени исполнения работ.

Обычно досрочное исполнение работы все равно может потребовать завершения работ, синхронизированных с ней, а затягивание сроков автоматически сдвинет сроки работ, зависимых от нее.

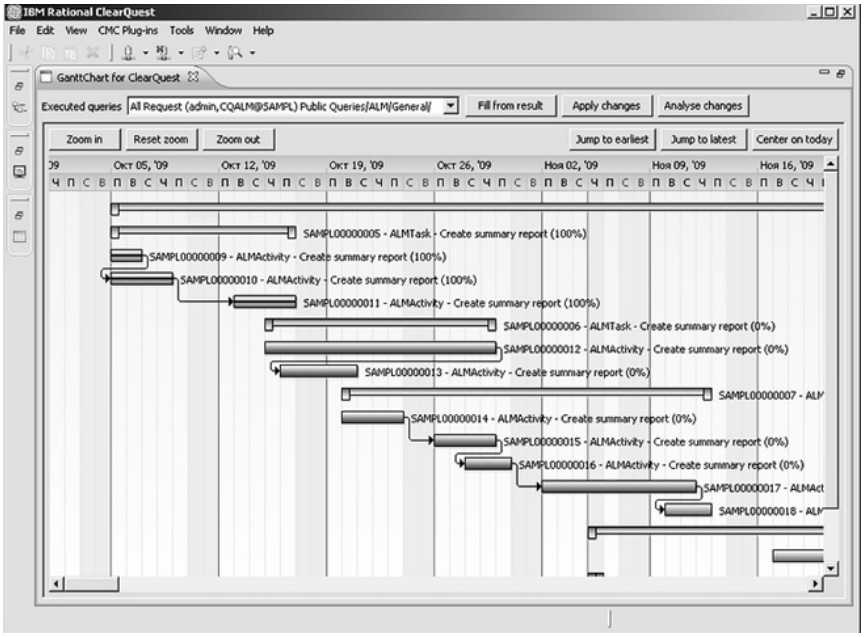


Рис. 6.45. Диаграмма Ганта для взаимосвязанных работ

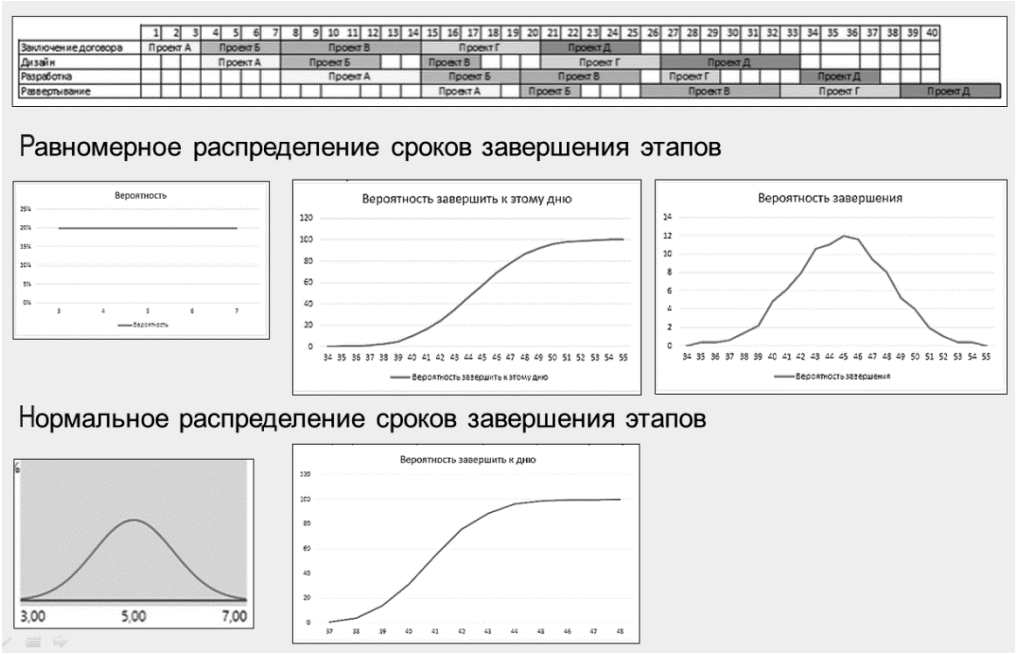


Рис. 6.46. Распределение сроков завершения взаимосвязанных работ

Стоимость и сроки

> Потом знакомые рекомендуют некоего Вавилыча, кристального старика. Кристальный старик приходит, с хватающей за душу медлительностью осматривает комнаты и берется сделать работу со своей олифой и кроном – все за двенадцать рублей. Тут же выясняется, что почтенный старец смертельно пьян и за свои слова отвечать не может. Его с трудом выводят.
>
> *Ильф И., Петров Е. На купоросном фронте*

Ключевые вопросы в момент принятия решения – сколько это будет стоить и в какие выльется сроки. Для этого необходимо определить следующие характеристики:

- трудоемкость проекта – на уровне проекта обычно измеряется в человеко-месяцах (N);
- срок разработки в месяцах;
- количество исполнителей.

Результаты оценки являются определяющими для других видов деятельности в управлении проектом: планирования работ, подбора команды. Основной параметр проекта – его трудоемкость, остальные параметры выводятся из него. Так, оптимальный *срок исполнения проекта* в месяцах при отсутствии других оснований рассчитывается из трудоемкости по формуле Боэма (см. методику COCOMO) как $T = 2{,}5\ N^{1/3}$.

То, что для проекта определенного объема существует оптимальный срок его разработки, объясняется практическими и житейскими доводами:

- увеличение сроков приводит к увеличению трудоемкости: работа занимает все отведенное для нее время (законы Мэрфи);
- попытка форсирования сроков за счет увеличения числа исполнителей дает результаты только в ограниченных пределах. Подключение исполнителей сверх некоторого предела, наоборот, приводит к резкому увеличению трудоемкости за счет затрат на вхождение в проект и усложнения коммуникаций между участниками. Сроки при этом значительно не сокращаются, а могут и, наоборот, вырасти.

Сметный подход – PERT

Смета – способ подсчета затрат как суммы произведений объемов отдельных работ, ресурсов и материалов на стоимость единицы. В своем натуральном виде этот способ не пригоден. Но если в качестве единиц работ использо-

вать *количественные параметры* кода или функционала проекта, то такую условную смету можно использовать для оценки при определенных допущениях:

- проект однороден, содержит значительное количество идентичных компонент – бизнес-объектов, форм, таблиц БД, событий, отчетов;
- выбранные параметры более или менее равномерно перекрывают программный код, т. е. затраты на реализацию одинаковых компонент приблизительно совпадают;
- проект идентичен по сложности и структуре с выполняемыми ранее этой же командой, имеется достоверная метрика по трудоемкости отдельных компонент.

Методика PERT (Program / Project Evaluation and Review Technique) учитывает вероятностный характер оценки на основе вероятностной оценки трудоемкости компонент каждого вида, но ничего не говорит о выборе этих компонент, оставляя это на совести оценщика. Сущность методики:

- определяется N видов функциональных параметров, имеющих количественную оценку;
- для каждого i-го типа вводится три вида трудоемкости на единицу параметра: пессимистическая, средняя и оптимистическая – P_i, M_i, O_i соответственно;
- вероятностная оценка – среднее и среднее квадратичное отклонение по i-й компоненте складывается из оптимистичной, средней и пессимистичной оценок:

$$E_i = (P_i + 4M_i + O_i)/6;$$

$$\text{СКО}_i = (P_i - O_i)/6;$$

- полная оценка получается как сумма по всему набору компонент:

$$E = \sum E_i;$$

$$\text{СКО} = \left(\sum \text{СКО}_i^2\right)^{1/2};$$

- оценка того, что трудоемкость с вероятностью 95 % не превысит расчетную, определяется по формуле

$$E_{95\,\%} = E + 2\text{СКО}.$$

Замечание по теме. Обычно трудоемкость функциональных компонент оценивается по затратам на их кодирование с учетом работы программиста, так как это легче всего сделать в метриках проекта. Тогда для полной оценки по всем технологическим процессам нужно учесть долю затрат на конструирование, которая статистически составляет около 25 %, т. е. окончательный результат умножается на 4.

Проект средней руки. Для однородного проекта, состоящего из набора приложений на основе клиент-серверной архитектуры с общим кодом для нижних уровней, массовые компоненты реализации могут считаться типовыми при имеющейся метрике трудоемкости (рис. 6.47).

Затраты чел/час	кол-во	P	M	O	Ei	СКОi	E	СКО
Приложений	5	30	20	10	20,0	3,33	100	17
Экранов (форм)	21	10	6	4	6,3	1,00	133	21
Элем. Управления	190	5	3	1	3,0	0,67	570	127
Отчетов	12	20	10	4	10,7	2,67	128	32
Событий	130	5	3	1	3,0	0,67	390	87
Таблиц БД	17	12	5	3	5,8	1,50	99	26
Полей БД	90	3	2	1	2,0	0,33	180	30
Бизнес-объектов	15	15	5	3	6,3	2,00	95	30
Форматов файлов	7	10	8	5	7,8	0,83	55	6
						Итого	**1750**	**167**
						E 95%	**2083**	
						E ч/мес	**51**	
		=H26/165/0,25				**Срок**	**9,2**	
						Исп	**5,5**	

Рис. 6.47. Оценка по методике PERT, человеко-часы

Основным недостатком сметного подхода является то, что он ориентирован на метрики конкретного технологического процесса в команде разработчиков. Метод функциональных точек [35] пытается охарактеризовать продукт с более общих позиций. *Функциональная точка* (Functional Point – FP) – обобщенная единица функциональности продукта. Различные компоненты функциональности нормируются, затем учитываются различные факторы, влияющие на проект аналогично методике COCOMO. Трудоемкость в выровненных функциональных точках перечитывается в строки кода (объем проекта) в соответствии с языком программирования.

Отраслевая статистика – COCOMO

Другой способ определения трудоемкости основан на анализе отраслевой статистики. Имея статистику основных параметров проектов – объемные пока-

затели, трудоемкость, сроки, а также данные о дополнительных факторах, оказывавших влияние на проекты, можно построить статистические зависимости трудоемкости и сроков от объемных показателей проекта. Наиболее известная **методика COCOMO (COnstructive COst Model)** [46] создана на основе анализа данных 161 проекта в области ИТ в 1997 г. Несмотря на свой солидный математический базис, она также технологически непродуктивна, так как опирается на единственный входной объемный показатель SLOC – количество строк кода в тысячах килострок. Это значение легко получить для готового проекта задним числом, а на этапе исследования придется опять использовать оценку объема проекта в SLOC разными способами:

- на основе общих оценок масштаба проекта;
- используя тот же сметный подход, только вместо трудоемкости оценивать объемные показатели кода SLOC в среднем на единицу компоненты каждого вида.

Итак, методика COCOMO использует модель, полученную на основе анализа статистики трудоемкости проектов методом *регрессионного анализа*. *Регрессия* – это интерполяция статистических данных линейной или экспоненциальной зависимостью. В данном случае используется экспоненциальная регрессия вида $Y = aX^b$. Аргумент функции регрессии – SLOC – объем проекта в килостроках кода.

Существуют три варианта моделей и три уровня оценки – *базовый, средний* и *детальный.* Базовый уровень выглядит так:

- трудоемкость $V = a \cdot \text{SLOC}^b$ [человеко-месяцев];
- срок разработки или длительность $T = cV^d$ [месяцев];
- число разработчиков $N = V / T$ [человек].

Коэффициенты a, b, c, d определены для разных классов проектов – *органический, полуразделенный, встроенный* – в зависимости от уровня и жесткости требований (рис. 6.48).

Труд	Срок	Исп		a	b	c	d
43	10	4	однородный	2,4	1,05	2,5	0,38
64	11	6	разнородный	3	1,12	2,5	0,35
96	11	9	встроенный	3,6	1,2	2,5	0,32
ч/мес	мес	чел					

Рис. 6.48. Базовый уровень оценки COCOMO II

Отсюда следует знаменитая формула Боэма для расчета сроков проекта

$$T = 2,5V^{1/3}.$$

Модель среднего уровня учитывает поправки, вносимые различными факторами, одни из которых являются линейными множителями – *множители трудоемкости* (F_i), а другие добавляются к показателю степени – *факторы масштаба* (M_j):

$$V = a \cdot \mathrm{KLOC}^b \prod F_i;$$

$$a = 2,94 b = 0,91 + 0,01 \sum M_j .$$

Значения коэффициентов заданы в таблицах в зависимости от *качественного состояния фактора*.

Параметры, учитываемые как множители трудоемкости:

1) PERS – квалификация персонала и текучесть кадров;
2) RCPX – сложность и надежность продукта;
3) RUSE – разработка для повторного использования;
4) PDIF – сложность платформы разработки;
5) PREX – опыт персонала;
6) FCIL – оборудование;
7) SCED – сжатие расписания. Диапазон изменения фактора от 75 до 160 % от номинальной длительности.

Параметры, учитываемые как факторы масштаба:

1) PREC – прецедентность, наличие опыта аналогичных разработок;
2) FLEX – гибкость процесса разработки;
3) RESL – архитектура и разрешение рисков;
4) TEAM – сработанность команды;
5) PMAT – зрелость процессов.

Модель *детального уровня,* выполненная в виде приложения «калькулятор» [49], использует расчетные формулы для многокомпонентного проекта. Она подсчитывает вероятностные оценки трудоемкости – оптимистическую, пессимистическую, наиболее вероятную (рис. 6.49), а также трудоемкость по модулям, фазам и технологическим процессам (рис. 6.50).

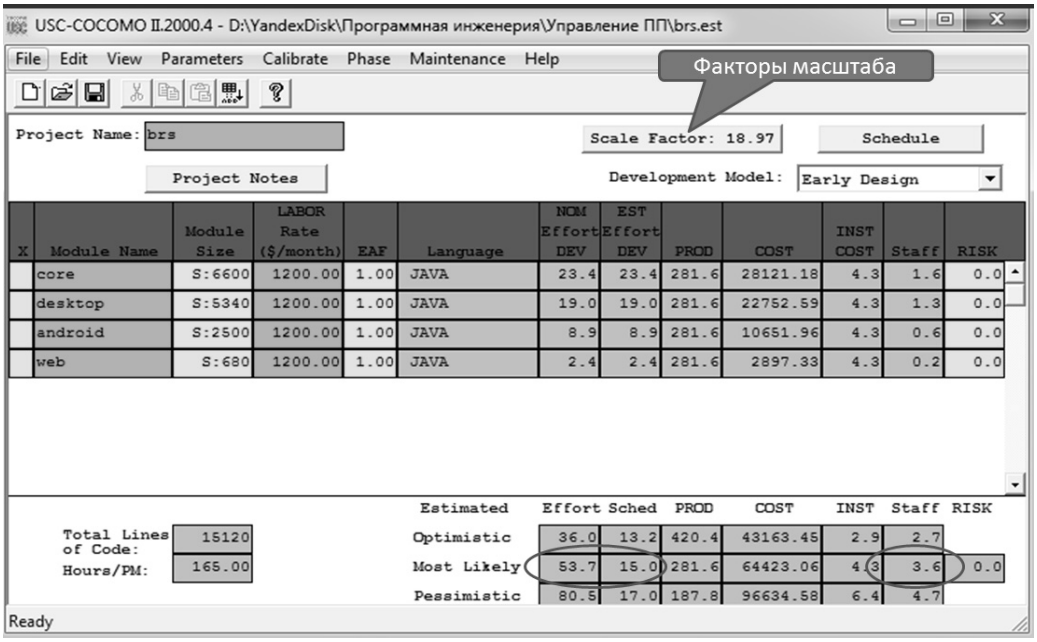


Рис. 6.49. Калькулятор трудоемкости COCOMO II для многокомпонентного проекта

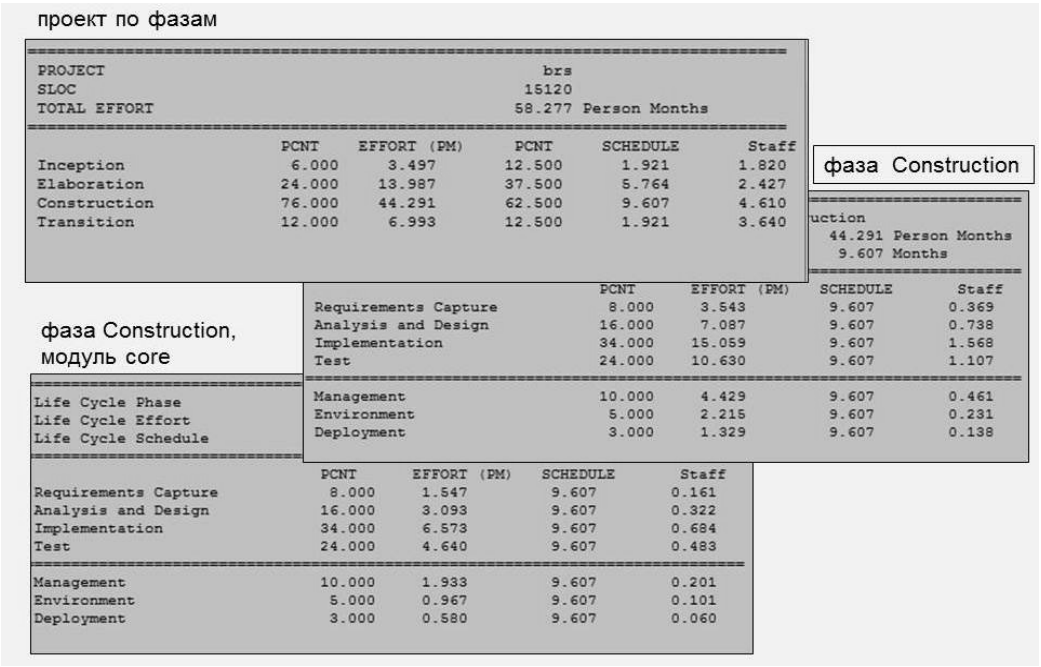


Рис. 6.50. Расчет детального уровня COCOMO II по фазам, дисциплинам и модулям

Как у других. Сметный подход широко используется в *градостроительстве*, где он регламентируется СНИПами (Строительные нормы и правила). На их основании производятся расчеты материальных затрат на строительство, исходя из основных параметров зданий, сооружений, автодорог и пр. При этом используются поправочные коэффициенты для различных внешних факторов и условий эксплуатации. Отраслевая статистика не может быть основанием для расчета, так как при соблюдении СНИПов она является их следствием.

Формирование команды

Структура команды исполнителей проекта определяется в соответствии с деятельностями и дисциплинами унифицированного процесса. В самом общем случае команда состоит из нескольких групп.

1. Анализ. Исполнители фазы исследования и дисциплины управления требованиями:

- бизнес-аналитик. Построение модели предметной области;
- бизнес-архитектор. Разработка бизнес-концепции – видения системы;
- системный аналитик. Перевод требований к продукту в функциональные требования к ПО;
- специалист по требованиям. Документирование и сопровождение требований к продукту;
- менеджер продукта (функциональный заказчик). Представление в проекте интересов пользователей продукта.

2. Управление проектом и технологическая административная верхушка:

- руководитель проекта;
- куратор проекта. Оценка планов и исполнения проекта, выделение ресурсов;
- системный архитектор, архитектор ПО. Разработка технической концепции системы, принятие ключевых проектных решений;
- руководитель группы тестирования;
- ответственный за управление изменениями, конфигурациями, за сборку и поставку программного продукта.

3. Производство. Проектирование, конструирование, сопровождение ПО:

- проектировщик. Проектирование компонентов и подсистем в соответствии с общей архитектурой, разработка архитектурно значимых модулей;

- проектировщик БД;
- проектировщик графического интерфейса;
- разработчик.

4. Тестирование:

- проектировщик тестов. Разработка тестовых сценариев;
- разработчик автоматизированных тестов;
- тестировщик. Тестирование продукта. Анализ и документирование результатов.

5. Обеспечение. Инструментальная поддержка. Вспомогательные процессы:

- технический писатель;
- переводчик;
- дизайнер графического интерфейса;
- разработчик учебных курсов, тренер;
- участник рецензирования;
- специалист по продажам, маркетолог;
- системный администратор, технолог;
- специалист по инструментальным средствам.

Приведенный список содержит максимальный перечень исполнителей. В зависимости от размера проекта и его содержания некоторые деятельности могут быть исключены или совмещаться одним исполнителем. Основания для совмещения могут быть разными:

- системный архитектор, архитектор ПО с разработчиком – удачное совмещение, поскольку архитектор ПО, по сути, вырастает из разработчика;
- разработчик с дизайнером графического интерфейса – неудачное совмещение: от дизайнера требуются специфические знания художественного дизайна, а разработчик будет превносить в интерфейс пользователя свои внутренние представления о реализации.

Структура и команды исполнителей и график их включения в проект (рис. 6.51) разрабатываются на основе следующих данных:

- общей трудоемкости проекта и сроков разработки;
- базового расписания проекта;

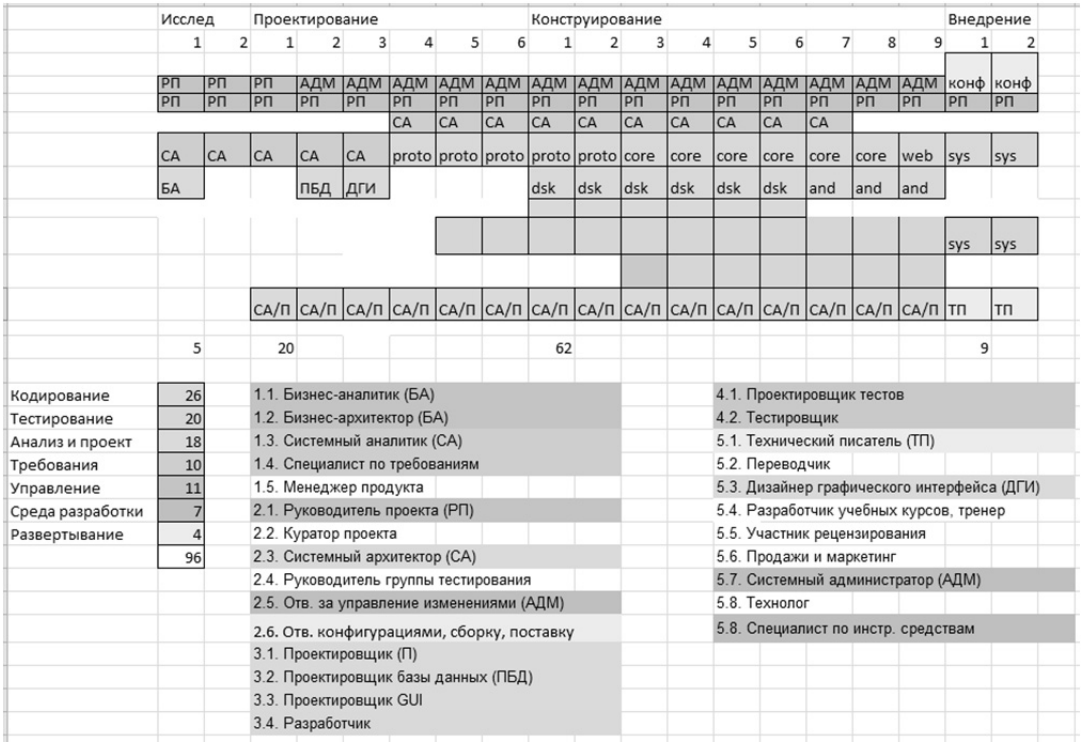


Рис. 6.51. Структура команды проекта

- распределения трудоемкости и сроков исполнения проекта по этапам и технологическим дисциплинам. Соответствующие данные можно получить, например, при помощи калькулятора трудоемкости из методики COCOMO-2 [49].

ГЛАВА 7
АРХИТЕКТУРА И ПРОЕКТИРОВАНИЕ

7.1. Параллельные прямые функционала и реализации

Прямые называются параллельными, если они лежат в одной плоскости и не пересекаются, сколько бы их ни продолжали.

Аксиома параллельных прямых

В предыдущих рассуждениях о бизнес-аналитике и системной аналитике все ясно: мы стремимся дать более или менее адекватное описание того, как программная система будет представлять окружающий мир и как она будет вписываться в происходящие в нем процессы. Процесс реализации (кодирование) – тоже вполне понятен: берешь и пишешь код, как умеешь. Между этими «что должна делать система» и «как это можно сделать» лежит архитектурная пропасть (рис. 7.1), преодолеть которую – задача архитектуры. В унифицированном процессе для этой цели предусмотрена отдельная технологическая дисциплина – **проектирование** или **программный дизайн** (design, не путать с графическим дизайном), концентрированным выражением которого является **архитектура.**

Наиболее распространенное определение архитектуры [55] включает следующие элементы:

- значимые решения по поводу организации ПС;
- структурные элементы и их интерфейсы, при помощи которых компонуется система;
- поведение – взаимодействие между этими элементами;
- компоновка элементов в иерархию подсистем;
- стиль архитектуры, который направляет эту организацию.

Что говорят классики

Казалось бы, все просто и понятно. Опиши систему на основе приведенных правил, и дело с концом. Однако на практике приходится видеть и такие описания (рис 7.2).

Рис. 7.1. Архитектурная пропасть между функционалом и реализацией

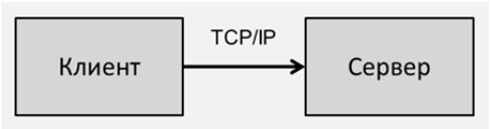


Рис. 7.2. «Архитектура» клиент-серверного приложения

Кто скажет, что это не архитектура, пусть первым бросит в меня камень. По крайней мере, первые два пункта формально выполнены. Так все-таки чего же в ней не хватает? И почему разработчики не в состоянии двинуться дальше?

Принципы: разумная достаточность, существенность, множественность представления

Если взять любую сложную техническую систему, например самолет, то можно нарисовать картинку, подобную приведенной выше, например контур с крыльями, хвостовым оперением и иллюминаторами. Можно, наоборот, дать

полную спецификацию всех деталей и узлов, которых насчитывается несколько сотен тысяч. И то и другое не будет архитектурой. Первое – по недостаточности, второе – по той же причине, по которой полный исходный текст программы не дает понимания того, как она работает. Для архитектуры важны другие аспекты.

Сущность системы, разумная достаточность – описание, достаточное для понимания ее сущности и процессов функционирования, излишние подробности столь же вредны, как и пробелы.

Множественность представления. Система не имеет единственного представления, ее описание состоит из нескольких представлений (аспектов, проекций, срезов), которые только в реализации создадут единое целое. Классический перечень представлений для ПС, именуемый «4 + 1», содержит [55] следующие компоненты:

- функциональное представление – сценарии, прецеденты, требования, предметная область;
- логическое представление – представление системы с точки зрения конечного пользователя, внешний вид, логическая структура;
- процедурное представление – параллелизм, потоки, взаимодействие, масштабируемость, производительность;
- представление развертывания – компоновка, топология, коммуникации, администрирование, настройка;
- представление разработки – структура программного кода, библиотеки, интерфейсы, абстракции.

Ключевым и объединяющим представлением является функциональное (рис. 7.3). Вопросы возникают только относительно логического представления. Представление конечного пользователя не является решающим, он может иметь какие угодно предубеждения относительно внутреннего устройства системы. Под логическим представлением разумнее понимать **согласованное для всех заинтересованных сторон представление о реализации функциональности на архитектурной модели проекта,** т. е. понимание как функциональное представление системы реализуется в архитектуре, доступное всем, имеющим отношение к проекту.

Архитектура – минимально избыточное, согласованное описание системы, включающее структуру, поведение, компоновку, стили разработки и значимые решения, создающее адекватное представление о ключевых моментах ее организации для всех заинтересованных сторон.

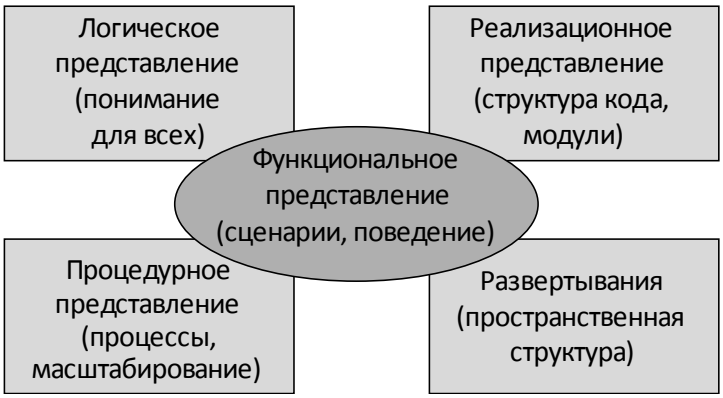


Рис. 7.3. Представления архитектуры в модели «4 + 1»

Термин «архитектура» в таком толковании относится к ПО системы в целом. Но он применим также в бизнес-аналитике и системной аналитике, где также необходимо собственное согласованное описание, поэтому бывают различные архитектуры и архитекторы со своими видами деятельности:

- **бизнес-архитектура** – бизнес-цели проекта, обоснование его успешности, экономические, социальные и прочие аспекты разработки ПС, доказательства того, что проект «выстрелит»;
- **системная архитектура** – описание системы, создаваемой преимущественно из готовых компонент, интеграция, адаптация под бизнес-процессы;
- **архитектура ПО** – описание системы со значительной долей разрабатываемого кода, взгляд на программный продукт изнутри.

Требования к архитекторам ПО

Так почему же разработка архитектуры не продвигается дальше представленной на рис. 7.2? Ответ прост. Требуется знание «матчасти» на соответствующем уровне, т. е. минимально избыточном, но необходимо подробном. Сюда входят: известные архитектурные компоненты, решения и стандарты, фреймворки, шаблоны проектирования, архитектурные стили, внутренние проблемы функционирования программного кода и способы их решения. Иначе архитектуру не из чего собирать, а ее проблемы неизвестны (рис. 7.4).

С другой стороны, архитектор не должен изобретать велосипед, он, как правило, использует уже известные идеи, его деятельность можно охарактеризовать как *искусство создания оригинального продукта с максимально возможным использованием доступных компонент и решений.*

Рис. 7.4. Программная архитектура и представление дилетанта

И последнее. Профессия архитектора, как и профессия руководителя, опирается на обобщенный практический опыт. Образно говоря, архитекторами не рождаются, ими становятся.

Ниже приведена выборка из описания трудовых функций профессионального стандарта «Архитектор программного обеспечения» [1]:

- разработка решений для повторного использования компонентов;
- определение перечня возможных слоев программных компонентов;
- определение перечня возможных протоколов взаимодействия компонентов;
- определение перечня возможных шаблонов (стилей) проектирования для каждого слоя или компонента;
- анализ качества кода: анализ зависимостей, статический анализ кода;
- оценка и выбор модели управления исключениями.

Специалист без навыков и опыта работы с кодом не может принимать перечисленные решения. В то же время ему свойственны функции руководителя

и управленца. В [56] перечислены требования к архитектору в области информационных технологий:

- технические знания – пресловутая «матчасть»: IT-стратегия, Enterprise Architecture, методологии проектирования и ведения проектов (UML, TOGAF, MSF, OUM), интеграционная архитектура (SOA и EDA), управление бизнес-процессами (BPM), процессы управления IT-инфраструктурой (ITIL / ITSM), архитектура и функциональность приложений (1C, Oracle, SAP, IBM WebSphere), архитектура инфраструктурных сервисов (LDAP, MS Exchange, DNS, DHCP, …), архитектура сети (TCP / IP, VOIP), архитектура центров обработки данных (электропитание, кондиционирование, пожаротушение, оборудование), облачные вычисления и виртуализация, фреймворки, методологии, стандарты и лучшие практики, языки и среды разработки: Java, .NET, SQL, PL / SQL, Delphi, операционные системы Windows, Linux, IBM AIX, Solaris;
- технические навыки – деятельности, связанные с управлением требованиями, проектированием и управлением программными проектами: сбор и управление требованиями, умение оценивать время и стоимость решения, проектирование систем, умение писать документы, умение работать на высоком уровне абстракции, управление техническими рисками, следование методологиям, умение создавать решения без личных предпочтений в технологиях;
- бизнес и менеджмент. Архитектор должен понимать, по каким законам функционирует бизнес, на чем компании зарабатывают деньги, как при помощи ИТ создавать ценность для бизнеса, чем отличаются OPEX от CAPEX и что предпочтительнее для конкретного бизнеса, как управлять проектами, как управлять рисками, как управлять людьми, как оценивать время и стоимость решения, отраслевые стандарты, предметные области, в которых ведутся проекты;
- личностные навыки: переговоры, манипуляции, английский язык, управление конфликтами, продажи, проектная деятельность, дипломатия (политика и интриги), ораторское искусство, умение брать ответственность, умение передавать ответственность, умение отстаивать свою позицию, тайм-менеджмент, самопрезентация.

Архитектурные аспекты проектирования

Согласно SWEBOK [2, 6] большинство аспектов проектирования имеет прямое отношение к архитектуре, так как представляет собой *значимые решения,* принятые в организации системы:

- ключевые идеи и концепции – абстрагирование, связность и соединение, модульность и декомпозиция, инкапсуляция, разделение интерфейса и реализации;

- параллелизм;
- контроль и обработка событий;
- распределение компонентов;
- обработка ошибок и исключительных ситуаций, обеспечение отказоустойчивости;
- взаимодействие и представление (шаблон MVC);
- сохраняемость данных – доступность «долгоживущих» данных;
- виды представления архитектуры – структурное, поведенческое, логическое, физическое, реализация кода;
- архитектурные стили;
- шаблоны проектирования;
- методы проектирования: нисходящее проектирование, модульное, абстрагирование, итеративность, функционально-ориентированное или структурное проектирование, объектно-ориентированное проектирование, проектирование на основе структур данных, компонентное проектирование.

Замечание по теме. Что будет, если какой-либо ключевой момент, например устойчивость как способность к восстановлению, не будет отражен в архитектуре? В этом случае в процессе кодирования возникающие проблемы будут решаться по принципу «как получится», что может существенно ухудшить характеристики системы, а при накоплении таких решений поставить предел ее развитию. Выходом из такой ситуации будет только глубокий реинжиниринг кода – «развалить» систему и собрать ее заново.

7.2. Проектирование, плавно переходящее в конструирование

В качестве примера попробуем описать процесс детального проектирования системы учета рейтинга успеваемости. В разделе 4.1 ее архитектура была показана без погружения в подробности, на уровне слоев, компонент и их связей. Дальнейшее продвижение содержит определенное противоречие. С одной стороны, уровень проработки не позволяет писать код, с другой – проработка, оторванная от реализации, должна вестись на уровне документов.

Выходом является проектирование от кода (см. раздел 1.1), которому здесь самое место. Детальное проектирование сопровождается тремя видами артефактов:

- каркас программного проекта – интерфейсы, абстракции, заготовки классов (свойства, структуры данных, заголовки ключевых методов, наследование, функциональные спецификации классов в комментариях);

- минимальный набор формальных документов – диаграммы архитектурных классов, диаграммы для ключевых взаимодействий архитектурных компонент;
- словесные формулировки ключевых моментов, структур и взаимодействий, применяемых паттернов.

Результатом дальнейшего наполнения каркаса кодом является появление архитектурного прототипа либо прототипов отдельных подсистем или компонент.

Уровень доступа к данным. Классы DAO

SQL-запросы не используются на уровне бизнес-логики. Единственным пользователем коннектора к БД являются объекты уровня доступа к данным – DAO (Data Access Objects). Сервис доступа данным реализован в классе DBItem с помощью рефлексии:

- любому классу, производному от DBItem, ставится в соответствие таблица в присоединенной БД с именем, совпадающим с именем класса без префикса DB;
- методы выбора, добавления или обновления DAO-объекта в таблице БД по его ключу-идентификатору, а также создание таблицы обеспечивают однозначное соответствие имен и типов данных класса и его предков с именами полей таблицы;
- поддерживаются соглашения для имен ссылочных полей со связью типа «один ко многим» и получение вектора DAO-объектов по значению связываемого поля;
- модификатор *transient* запрещает связывать свойство DAO-объектов с полем БД.

Классы табличных объектов соответствуют основным классам-сущностям модели предметной области, а ссылочные поля – отношениям между классами (рис. 7.5).

Структура БД, изображенная на рис. 7.6, генерируется автоматически по списку классов DAO-объектов приложения.

Для создания таблицы в БД достаточно определить класс-наследник для *DBItem* со списком необходимых свойств – примитивных типов данных и свойств – ссылочных полей.

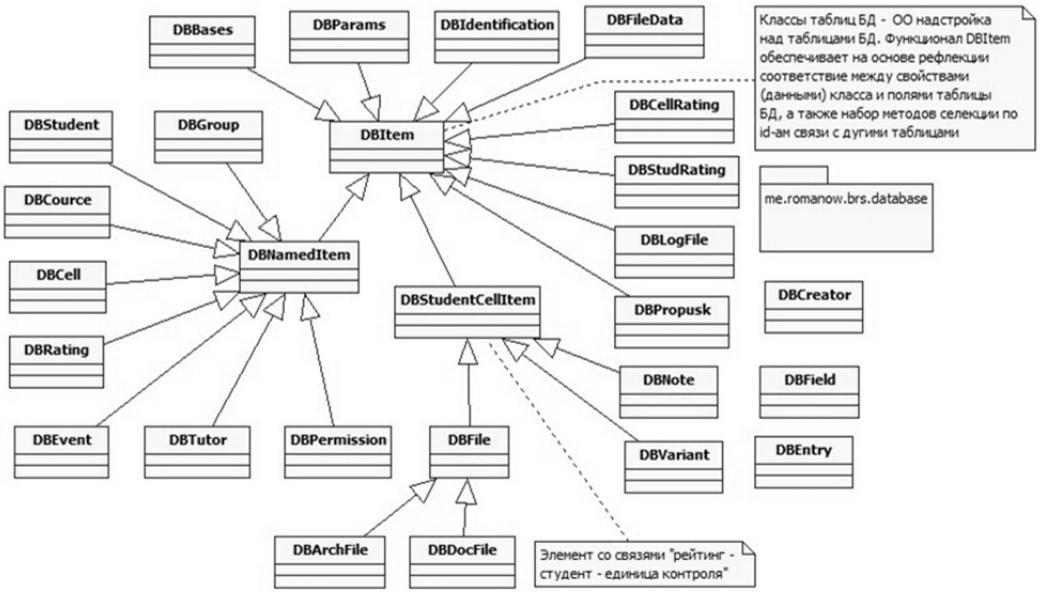


Рис. 7.5. Диаграмма классов таблиц БД (DAO)

Рис. 7.6. Структура БД

Классы бизнес-объектов

Классы бизнес-объектов (см. раздел 4.1) составляются из связанных объектов уровня DAO таким образом, чтобы следующий уровень бизнес-процессов получал уже готовые сборки бизнес-сущностей с необходимыми, часто используемыми связями. Принципы создания бизнес-объектов в данной реализации:

- классы бизнес-сущностей наследуют соответствующие DAO-объекты, например, MDStudent – DBStudent (рис. 7.7). Бизнес-сущности могут содержать поля, загружаемые из других источников, например, из связанных таблиц;

```
]public class MDStudent extends DBStudent{
    transient public boolean studRatingChanged=false;
    public int brigade=0;      // Данные из связанной таблицы StudRating
    public int cDate=0;
    public boolean second=false;
```

Рис. 7.7. Бизнес-объект, унаследованный от DAO

- бизнес-объекты могут содержать векторы связанных бизнес-сущностей. Для их загрузки соответствующему методу необходимо передать класс связываемых DAO-объектов и DAO-объект, на который они ссылаются. Последний содержит требуемое значение ключа. Например, для получения вектора объектов – студентов группы – необходимо передать объект DBGroup со значением ключа и описатель класса DBStudent.class. Генерируется SQL-запрос в соответствии с принятыми соглашениями по именованиям связываемых полей;
- система бизнес-объектов построена по иерархическому принципу (рис. 7.8). Основной бизнес-объект «рейтинг» (MDRating) содержит бизнес-объекты «группа» (MDGropup) и «предмет» (MDCource), которые имеют векторы связанных бизнес-объектов: «студент» (MDStudent) и «единица контроля» (MDCell) соответственно (рис. 7.9);
- бизнес-объект «рейтинг» работает с множествами данных, существующих в двумерных системах координат – *предмет / единица контроля* и *студент / занятие*, т. е. табличных данных с соответствующими координатными осями. По этому принципу организованы сущности – *оценки, пропуски, файлы отчетов и архивов*. Для моделирования такого представления существуют специальные классы, использующие линейные векторы DAO-объектов с соответствующими парами ключей;

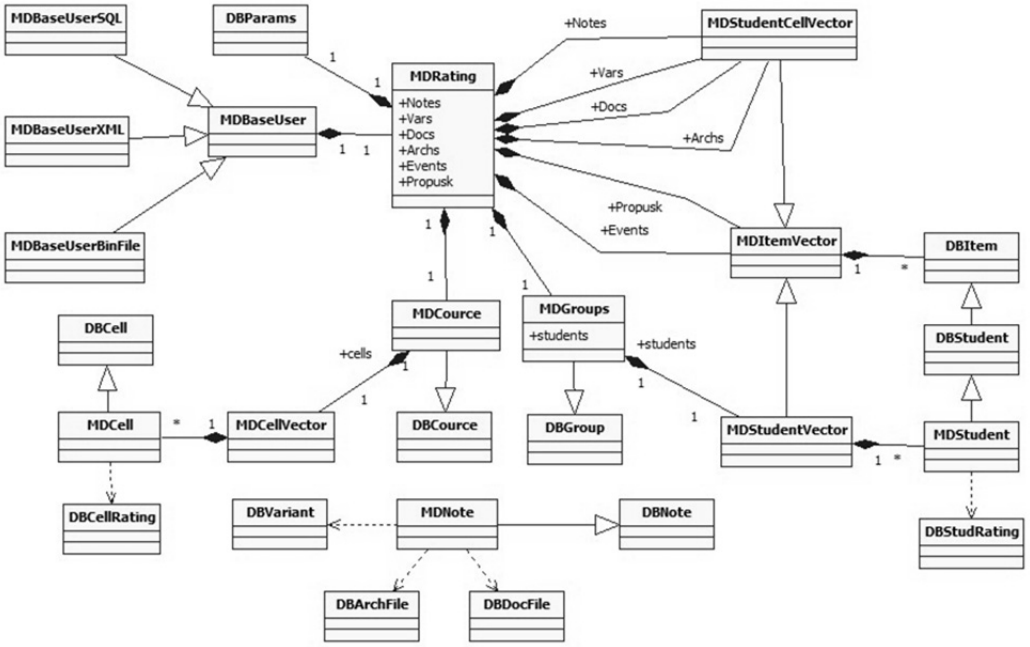


Рис. 7.8. Диаграмма классов бизнес-модели

```
public class MDRating extends DBRating{
    public MDGroups groups=null;
    public MDCourse course=null;
    public DBParams params=null;
    public MDItemVector events=new MDItemVector();
    public MDItemVector propusk=new MDItemVector();
    public MDStudentCellVector notes=new MDStudentCellVector();
    public MDStudentCellVector vars=new MDStudentCellVector();
    public MDStudentCellVector docs=new MDStudentCellVector();
    public MDStudentCellVector archs=new MDStudentCellVector();
```

Рис. 7.9. Заголовок класса MDRating

- для бизнес-объекта *рейтинг* имеется три метода загрузки данных: *минимальный, стандартный* и *полный* с соответствующей глубиной связывания данных по иерархии;
- бизнес-модель реализована в классе MDBaseUser с интерфейсом MDBaseFace, который является базовым для расширений – источников данных модели (рис. 7.10 и 7.11).

```
public interface MDBaseFace {
    public DBIdentification getDBIdentification() throws Throwable;
    public void connect() throws Throwable;          // подключение к источнику данных
    public void reconnect() throws Throwable;        // переподключение к источнику данных
    public void close() throws Throwable;            // закрыть источник данных
    public boolean isConnected() throws Throwable;   // проверка подключени к источнику
    public void loadRating(int id) throws Throwable; // загрузка рейтинга в бизнес-модель
    public void loadPermissionList() throws Throwable;// загрузка списка разрешений
    public void loadStudentList() throws Throwable;  // загрузка списка студентов тек. рейтинга
    public void loadCellList() throws Throwable;     // загрузка списка дисциплин тек. рейтинга
    public void loadEventList() throws Throwable;    // загрузка списка контроля посещаемости
    public void loadFullRating(int id) throws Throwable;// загрузка рейтинга с оценками и пропусками
    public void loadStudentRating(int id,int sid) throws Throwable;
    public void loadCellRating(int id,int cid) throws Throwable;
    public void testEdit() throws Throwable;         // загрузка оценок по студенту и ед.контроля
    public MDNote getNote() throws Throwable;        // получить оценки тек. студента и ед. контроля
    public DBItem []getNoteHistory() throws Throwable; // получить историю изменения оценки
```

Рис. 7.10. Интерфейс MDUserFace

```
public abstract class MDBaseUser extends DBNamedItem implements MDBaseFace{
    //------------- Компоненты данных ----------------------------------------
    public MDRating rating=null;          // бизнес-объект - рейтинг
    public DBTutor tutor=null;            // бизнес-объект - преподаватель
    public DBProfile entry=null;          // бизнес-объект - данные подключения
    //------------- Компоненты редактирования --------------------------------
    public DBTutor []permission=null;     // массив id-ов разрешений для преподавателя
    public DBItem []ratingList=null;      // массив id-ов списка рейтингов
    public boolean editEnabled=false;     // разрешение редактирования рейтинга
    public MDStudent studentItem=null;    // бизнес-объект - выбранный студент
    public MDCell cellItem=null;          // бизнес-объект - выбранная единица контроля
    public DBEvent eventItem=null;        // бизнес-объект - котроля посещаемости (занятие)
    public boolean wasChanged=false;      // флаг изменения рейтинга
    public DBConnect getConnect(){ return null; }
```

Рис. 7.11. Класс MDBaseUser

Источники данных для бизнес-модели

Производные классы бизнес-модели (рис. 7.12) обеспечивают реализацию толстого и тонкого клиентов (WebAPI). Базовый класс бизнес-логики *MDBaseUser* имеет три расширения:

- *MDBaseUserSQL* загружает бизнес-объекты непосредственно через соединение с БД, используется в толстом клиенте;
- *MDBaseUserBinFile* сохраняет бизнес-объект «рейтинг» в двоичном файле, используется для работы с локальными копиями рейтингов;

• *MDBaseUserXML* реализует режим тонкого клиента с созданием WebAPI. Используется ряд классов, сериализуемых в XML-формат, для поддержки команд и ответов протокола (XMLCmd, XMLAnswer, XMLArray). В протоколе также сериализуются бизнес-объекты, например MDNote. MDBaseUserXML является клиентской частью протокола, а MDXMLCommand – серверной. Последний создает на сервере объект MDBaseUserSQL и выполняет в нем необходимые команды от имени клиента.

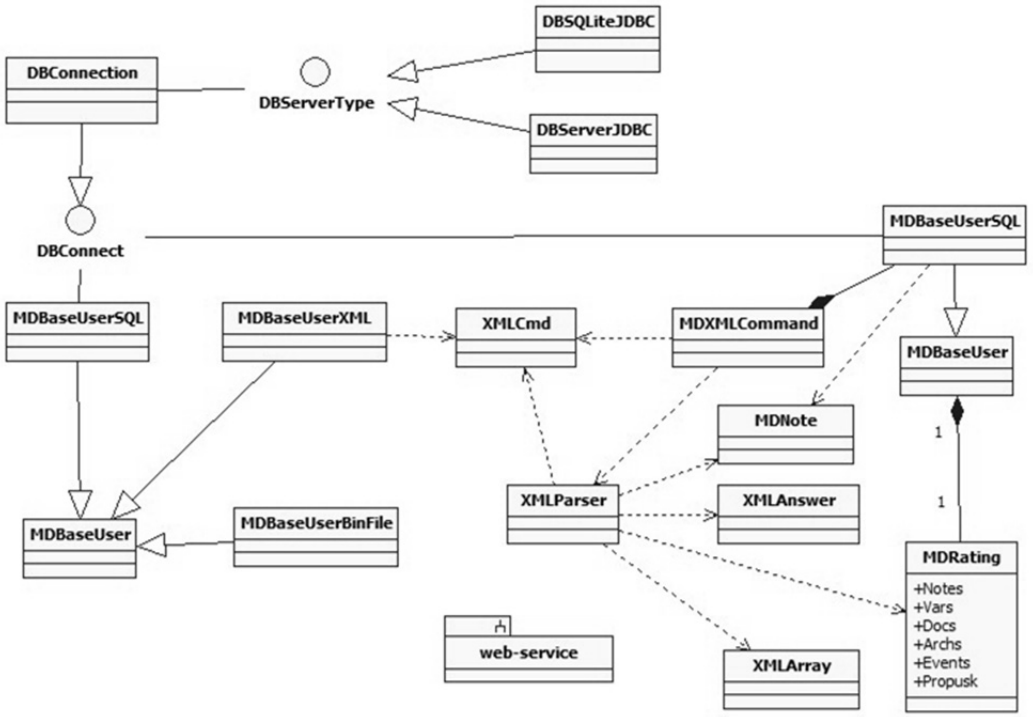


Рис. 7.12. Производные классы для источников данных

Реализация толстого клиента

В толстом клиенте сетевые коммуникации реализованы непосредственно в драйвере доступа к БД JDBC, поэтому все слои приложения локализованы в клиенте. Производный класс бизнес-модели MDBaseUserSQL (рис. 7.13) доопределяет абстрактные методы загрузки бизнес-объектов через прямое обращение к объекту соединения с БД DBConnect, поддерживающему сервис DAO.

```
public class MDBaseUserSQL extends MDBaseUser{
    public final static int blockSize=32000;
    private DBConnect conn=null;
    public DBConnect getConnect(){ return conn; }

@Override
public void loadFullRating(int id) throws Throwable{
    loadShortRating(id);
    loadStudentList();
    loadCellList();
    loadEventList();
    loadPermissionList();
    //-----------------------------------------------------------------------
    DBItem xx[]=conn.getList(DBNote.class, rating);
    rating.notes.clear();
    for(int i=0;i<xx.length;i++){
        rating.notes.add(xx[i]);
        }
    xx=conn.getList(DBVariant.class, rating);
    rating.vars.clear();
    for(int i=0;i<xx.length;i++){
        rating.vars.add(xx[i]);
        }
```

Рис. 7.13. Производный класс MDBaseUserSQL толстого клиента

Протокол и коммуникации в тонком клиенте с WebAPI

На рис.7.12 изображены также классы коммуникаций для тонкого клиента. Прикладной протокол тонкого клиента строится на базе HTTP-протокола и содержит набор синхронных запросов к серверу, соответствующих абстрактным методам загрузки и обновления данных бизнес-модели в MDBaseUser.

Далее процесс фактически переходит от детального проектирования к конструированию, дальнейшая разработка протокола элементарно вписывается в технологические средства поддержки протокола классами URLConnection и Servlet:

- клиент создает http-соединение на основе объекта класса URLConnection;
- часть параметров команды клиента может быть передана в заголовке http-запроса;
- после передачи заголовка клиент и сервер договариваются, в каком формате (текстовом или двоичном), а также в какой кодировке они будут передавать дополнительные данные запроса и ответ. Аналогичные параметры имеются в самом заголовке http-запроса, поэтому их необходимо корректно устанавливать;

- обычно в качестве передаваемых данных используются объекты, сериализованные в двоичном или общепринятом текстовом формате XML или JSON;
- описание протокола состоит из описания форматов запросов клиента и возможных ответов сервера.

Решения, принятые далее, являются технологическими: форматы представления потоков данных, проблемы размерностей стандартных типов данных (строка в формате UTF), кодирование команд и ответов, способ сериализации объектов, прикрепление объектов данных в ответе сервера.

Единственным принципиальным решением является создание на сервере класса для бизнес-модели MSBaseSQL с локальным источником данных, что позволяет использовать общий код слоя доступа к данным в тонком и толстом клиентах.

Клиентская часть протокола

От клиента к серверу не передаются значительные объемы данных, поэтому большинство запросов оформляется в виде объекта класса XMLCmd (рис. 7.14), при необходимости вслед за ним в потоке передается дополнительный объект.

```
public class XMLCmd {
    public int code=0;
    public int id=0,id2=0,id3=0,id4=0;
    public String str="";
```

Рис. 7.14. Класс «запрос клиента»

В протоколе используется XML-сериализация объектов, полученые строки пишутся в *двоичный поток данных* (рис. 7.15) методом *writeUTF*.

Ответ сервера выглядит аналогично. Вначале следует объект XMLAnswer с основными параметрами ответа. В протоколе предусмотрены три варианта в соответствии с кодом ответа:

- ответ без дополнительного объекта данных;
- ответ с дополнительным объектом данных, возвращаемый в качестве результата метода;
- сообщение об ошибке, приводящее к генерации исключения в клиенте.

В большинстве случаев реализация переопределяемых методов получения и изменения бизнес-объектов на клиентской стороне заключается в вызове единственного метода sendCmd с передачей ему параметров и сохранением объекта-результата (рис. 7.16).

```
Object sendCmd(int code, int id, int id2, int id3, int id4, String w, DBItem obj) thro
Object res=null;
try {
    if (binary){                          // Двоичный поток туда-обратно
        httpConnect();
        XMLCmd cmd=new XMLCmd(code,id,id2,id3,id4,w);
        DataOutputStream out=new DataOutputStream(conn.getOutputStream());
        out.writeUTF(pars.toXML(cmd));  // Сериализованный XMLCmd в двоичном формате
        if (obj!=null) {
            out.writeUTF(pars.toXML(obj));    // Дополнительный объект в двоичном формате
            }
        out.flush();                      // Протолкнуть данные в поток
        is=conn.getInputStream();      // Открыт двоичный поток приема
        DataInputStream dis=new DataInputStream(is);
        ans=(XMLAnswer)pars.fromXML(dis.readUTF()); /
        if (ans.code>0) throw new BRSException(BRSException.serv,ans.message);
        if (ans.code==XMLAnswer.noobj) res=null;
        else{                             // Есть присоединенный объект
            int sz=dis.readInt();         // Счетчик длины (int)
            char cc[]=new char[sz];  // Массив принимаемых символов
            for(int i=0;i<sz;i++) cc[i]=dis.readChar();
            String ss=new String(cc);     // Строка на основе принятого массива
            res=pars.fromXML(ss);         // Парсить XML-ответ из строки
            if (res instanceof XMLAnswer)
                throw new BRSException(BRSException.serv,((XMLAnswer)res).message);
            }
```

Рис. 7.15. Передача команды клиента и обработка ответа

```
@Override
public void loadRating(int id) throws Throwable{
    rating=(MDRating)sendCmd(XMLCmd.loadRating,id);
    }
@Override
public void loadFullRating(int id) throws Throwable {
    rating=(MDRating)sendCmd(XMLCmd.loadFullRating,id);
}
```

Рис. 7.16. Получение бизнес-объектов на стороне клиента

Серверная часть протокола

Из описания протокола в клиенте понятно, как аналогичные действия будут выглядеть на стороне сервера (рис. 7.17). Сервлет читает объект XMLCmd, создает парсер и локальную бизнес-модель, после чего переключается на обработку команды с соответствующим кодом. Перехватываемые исключения отправляются в виде объекта XMLAnswer с кодом и текстом ошибки.

```
protected void processRequestBinary(HttpServletRequest request, HttpServletResponse response){
    out=null;
    XMLParser pars=null;
    XMLCmd cmd=null;
    try {
        pars=getParser();
        response.setContentType("text/html;charset=UTF-8");
        out=new DataOutputStream(response.getOutputStream());
        DataInputStream is=new DataInputStream(request.getInputStream());
        cmd=(XMLCmd)new XMLParser().fromXML(is.readUTF());
        MDBaseUserSQL conn=getJDBC();
        switch (cmd.code){
default:    throw new BRSException(BRSException.serv,
            "Функция код:"+cmd.code+" не поддерживается");
.....
    } catch(Throwable ee){
        try {              // исключения – объект XMLAnswer с кодом ошибки и сообщением
            BRSException e2=new BRSException(ee);
            XMLAnswer ans=new XMLAnswer(e2.getCode(),e2.getMessage());
            out.writeUTF(pars.toXML(ans));    // передать объект в качестве ответа
            } catch(Throwable e3){}
        }}
```

Рис. 7.17. Серверная часть протокола для двоичного потока

Типовая реализация серверной компоненты предусматривает наличие *агента*, в функции которого входит поддержание необходимого *контекста*, в котором исполняются локальные запросы.

В нашем случае никакого постоянного окружения запросы не имеют, так как являются независимыми друг от друга. Поэтому сервер просто вызывает соответствующий метод в полученной локальной бизнес-модели MDBaseUserSQL, а затем сериализует объект ответа XMLAnswer, а вслед за ним необходимую компоненту, извлеченную из бизнес-модели. Если в команде присутствует дополнительный объект, то он читается непосредственно обработчиком команды (рис. 7.18).

С приведенным кодом связан один из *типовых дефектов диапазона представления данных* (см. раздел 8.2), который вручную нейтрализован с помощью метода *writeChars*.

```
private void writeChars(String ss)throws Throwable{  // Передача строки неограниченной длины
    int sz=ss.length();                               // в двоичном формате
    out.writeInt(sz);                                 // Передать счетчик длины
    char cc[]=ss.toCharArray();                       // Конвертировать строку в массив символов
    for(int i=0;i<sz;i++)
        out.writeChar(cc[i]);                         // Передать символы
    }
.....
case XMLCmd.loadRating:                               // Читать рейтинг по id
                conn.loadRating(cmd.id);              // Загрузить в локальную бизнес-модель
                out.writeUTF(pars.toXML(new XMLAnswer(true)));
                writeChars(pars.toXML(conn.rating));// Передать заголовок ответа XMLAnswer
                break;                                // с отметкой, что следует объект данных
case XMLCmd.loadFullRating:                           // Сериализовать бизнес-объект MDRating
                conn.loadFullRating(cmd.id);
                out.writeUTF(pars.toXML(new XMLAnswer(true)));
                writeChars(pars.toXML(conn.rating));
                break;
case XMLCmd.changeNote:                                       // Редактировать оценку
                conn.loadShortRating(cmd.id);                 // Загрузить «компактную» бизнес-модель
                conn.setStudentItem(cmd.id2);                 // Установить студента и ед.контроля
                conn.setCellItem(cmd.id3);                    // по переданным id-ам
                // Читать дополнительный объект напрямую из потока
                MDNote mm=(MDNote)new XMLParser().fromXML(is.readUTF());
                conn.changeNote(mm);                          // Вызвать метод в модели с параметром
                out.writeUTF(pars.toXML(new XMLAnswer()));    // - полученным объектом
                break;
```

Рис. 7.18. Непосредственная обработка запросов сервером

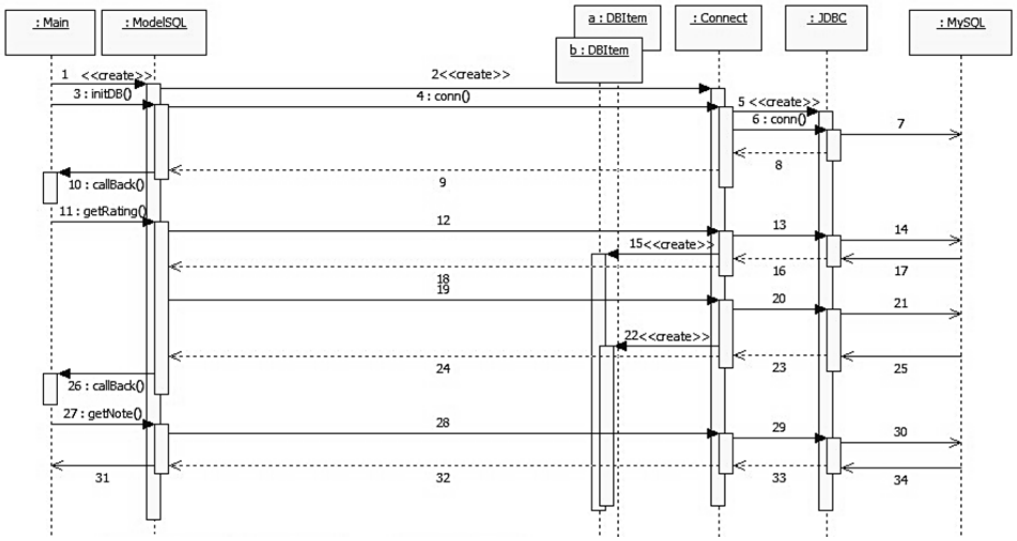


Рис. 7.19. Толстый клиент. Сетевое соединение в библиотеке доступа к БД

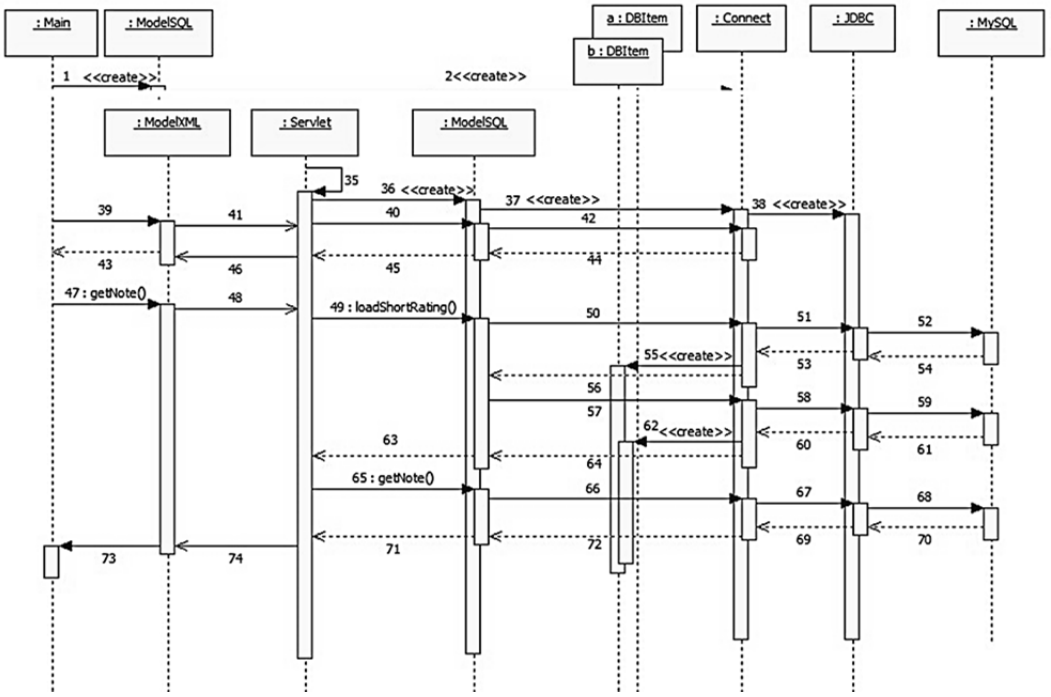


Рис. 7.20. Тонкий клиент. Сетевой протокол в бизнес-логике

В качестве иллюстрации приведем диаграммы последовательности для толстого и тонкого клиентов. Даже поверхностное визуальное сравнение позволяет увидеть идентичность поведения на стороне клиента и на стороне сервера в первом и втором случае (рис. 7.19 и 7.20).

Архитектурный MVC

В разделе 4.2 были описаны особенности паттерна архитектурного MVC, в котором контроллер определяет долгосрочное поведение представления в бизнес-процессе. Одним из преимуществ такого подхода является возможность повторного использования кода контроллера на разных платформах. В нашем примере бизнес-процесс определяется возможностью управления рейтингом и просмотра его составляющих путем выбора соответствующих полей из списков и нажатия кнопок.

Контроллер реализует варианты поведения внешнего представления, определяет возможные последовательности нажатия кнопок и генерирует события для изменения состояния элементов отображения. Он реализует модель поведения на основе диаграммы состояний (рис. 7.21).

Контроллер является имитатором поведения абстрактного представления. В идеальном случае представление – тонкий клиент, лишенный модели поведения. Контроллер и представление связаны двумя интерфейсами – передачи *событий* от представления к контроллеру и *команд* контроллера по вводу / выводу содержимого и отображению элементов управления. Бизнес-модель (MDBaseUser и его производные) создается и управляется контроллером, хотя некоторые ее данные могут быть получены представлением по прямой ссылке (рис. 7.22).

Рис. 7.21. Диаграмма состояний внешнего представления в контроллере

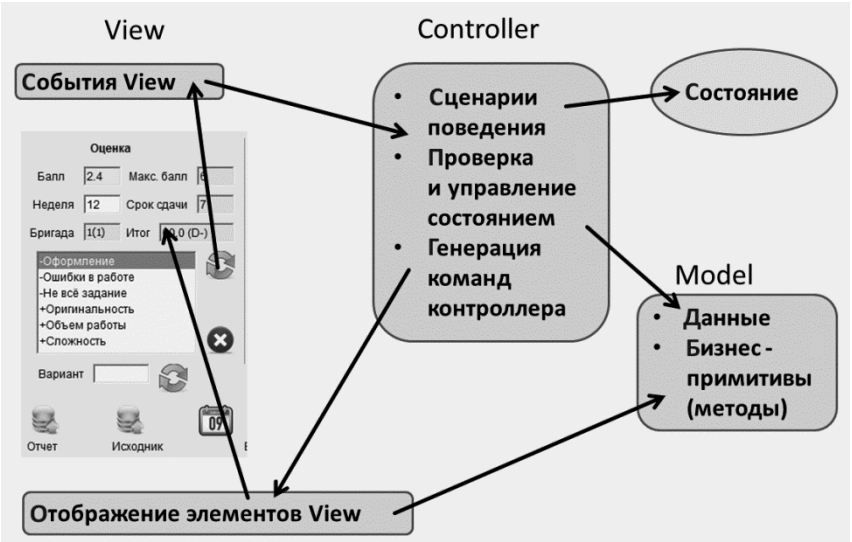


Рис. 7.22. Взаимодействие представления (View) и контроллера

Интерфейс *ViewControllerCommands* определяет набор событий и команд, передаваемых контроллеру со стороны представления, инициируемых такими действиями, как нажатие кнопок или выбор элементов списков (рис. 7.23).

```
public interface ViewControllerCommands {
    public double getSum();
    public MDNote getNoteItem();              // Получить текущую оценку от модели
    public void setState();
    public void retryState();
    public void close();
    public void updateAllLocalRatings();      // Синхронизировать локальные копии
    public void loadRating(int id) throws Throwable; // Загрузить рейтинг из модели
    public void loadFullRating(int id) throws Throwable;
    public void loadFullRatingSoft(int id) throws Throwable;
    public void setState(int newState);       // Попытка перевести контроллер (модель) в состояние
    public void callBacks();                  // «Провоцирование» контроллера на возможные события
```

Рис. 7.23. Интерфейс событий и команд представления

Интерфейс *ViewControllerListener* определяет набор команд контроллера для выполнения представлением передаваемых действий над элементами отображения (рис. 7.24).

```
public interface ViewControllerListener {
    public String getInputFileName(String title, String defName) throws Throwable;
                                              // получение имени файла контроллером
    public String getOutputFileName(String title, String defName) throws Throwable;
    public void fatal(Throwable ee);          // исключение, перехваченное контроллером
    public void stateChanged(int state);      // изменение состояния контроллера
    public void setRatingVisible(boolean enb); // «видимость» полей данных рейтинга
    public void setCellVisible(boolean enb);  // «видимость» полей данных единицы контроля
    public void setStudentVisible(boolean enb); // «видимость» полей данных студента
    public void setNoteVisible(boolean enb);  // «видимость» полей данных оценки
```

Рис. 7.24. Интерфейс команд контроллера

Контроллер управляет всем бизнес-процессом приложения, поэтому он создается главным окном и передается по ссылке между окнами (формами).

Для управления поведением контроллер имеет методы повторной генерации текущих событий, проверки текущего состояния бизнес-процесса, попытки его перевода в указанное состояние, а также пропускает через себя команды от представления по изменению компонент модели, проверяя возможность исполнения соответствующих действий.

Помимо основных функций, контроллер в таком виде может выполнять другие общесистемные функции:

- кэширование данных – создание дополнительной бизнес-модели, настроенной на локальный источник данных. Эта бизнес-модель используется при отсутствии соединения с сервером, невидима для представления. При восстановлении соединения с сервером контроллер синхронизирует накопленные в ней изменения с основной моделью;

- перехват исключений, возникающих в модели, представлении, а также генерируемых самим контроллером, информирование о них представления через интерфейс событий контроллера в текстовом формате.

Рис. 7.25. Реализация интерфейсов контроллера в desktop и Android

На рис. 7.25 прказано, что программный код представлений является *чистым* с точки зрения поведения, т. е. представляет собой формальную переадресацию событий и связанных с ними данных от элементов управления формы к контроллеру и обратно с использованием платформенно-зависимых методов работы с графическим интерфейсом в данном приложении.

Тонкий web-клиент

Рассмотрим еще один вариант тонкого клиента – клиент как набор html-страниц web-сервера (web-клиент). Для эффективной реализации необходимо подобрать технологию, максимально сохраняющую преемственность с теку-

щей архитектурой и структурой кода. В качестве такой технологии можно использовать Ajax (рис. 7.26) [90].

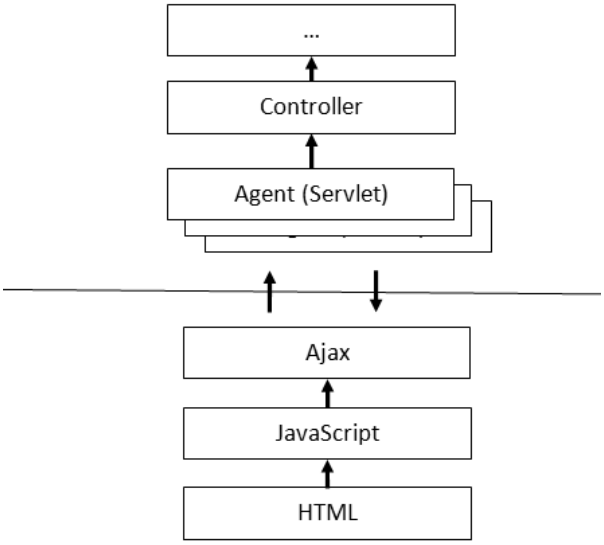


Рис. 7.26. Технология Ajax в web-клиенте

Ajax позволяется посылать отдельные запросы к серверу, не привязываясь к процессу перерисовки web-страницы. Клиентская компонента протокола посылает собственный запрос к серверу, указывая необходимый активный ресурс, например сервлет. Активный ресурс возвращает ответ в формате XML, из которого клиентская компонента извлекает необходимые данные. Применительно к описываемой архитектуре этот процесс будет выглядеть так:

- существующая разметка формы тонкого клиента преобразуется в разметку web-страницы (HTML);
- на активные элементы страницы (кнопки, поля со списком) навешиваются события, обработчики которых реализуются на JavaScript;
- каждый обработчик посылает Ajax-запрос к сервлету с кодом команды и параметрами;
- сервлет по коду команды выполняет соответствующее обращение к контроллеру на стороне сервера, полученные ответные данные передаются клиенту сообщением в формате XML;
- поскольку контроллер запоминает текущее состояние представления, то на стороне сервера необходимо использовать механизм сессий для сохранения контекста соединения [91].

В оригинале интерфейс «представление–контроллер» является двунаправленным, так как ответы контроллера формально выглядят как *асинхронные*. На самом деле ответы контроллер может посылать только после соответствующего запроса, но их может быть несколько. Чтобы вписать соответствующую схему в полностью синхронный протокол Ajax, контроллеру необходимо в интерфейс *ViewControllerListener* посылать еще одну команду – *окончание ответа*. Тогда сервлет будет накапливать ответы и посылать их единым сообщением.

Использование фреймворка DWR [92] скрывает излишние подробности Ajax при программировании на Java:

- на сервере пишется и объявляется класс с определенным именем, методы которого напрямую вызываются на клиентской стороне из JavaScript;
- DWR компилирует интерфейсы этих методов в вызовы аналогичных методов в JavaScript с передачей параметров. Например, если на сервере есть класс *XXX* с методом *AAA(int b, String c)*, то его можно вызвать напрямую из JavaScript в виде *XXX.AAA(5,"fff")*;
- для обработки ответа в вызове может быть указано имя функции, которая асинхронно вызывается при получении ответа и содержит параметр-ответ, например строку.

На рис. 7.27 изображено решение, позволяющее web-клиенту максимально использовать код толстого и тонкого клиентов с WebAPI.

Клиент строится и работает следующим образом:

- разметка и имена элементов управления копируются из формы, создаваемой в оконном классе на Java в html-код (1);
- по событию в форме (2) планируется вызов функции (7) в коде JavaScript (3). Функция средствами DWR вызывает метод в классе сервлета *DWRServer* (4) через одноименный метод, сгенерированый в JavaScript (8). Ему передаются параметры в формате строки, в том числе идентификаторы объектов;
- метод сервера получает сессию и объект-контекст (5), с которым связаны контроллер и слушатель его событий;
- код метода сервера (4) при обращении к контроллеру (9) изоморфен коду сервлета тонкого клиента с WebAPI, т. е. формально откорректирован с учетом изменения окружения, его исполняющего;
- объект-контекст является одновременно слушателем событий контроллера. Обрабатывая их, он пишет результаты в формате текстовой строки в виде текста скрипта для клиента на JavaScript;

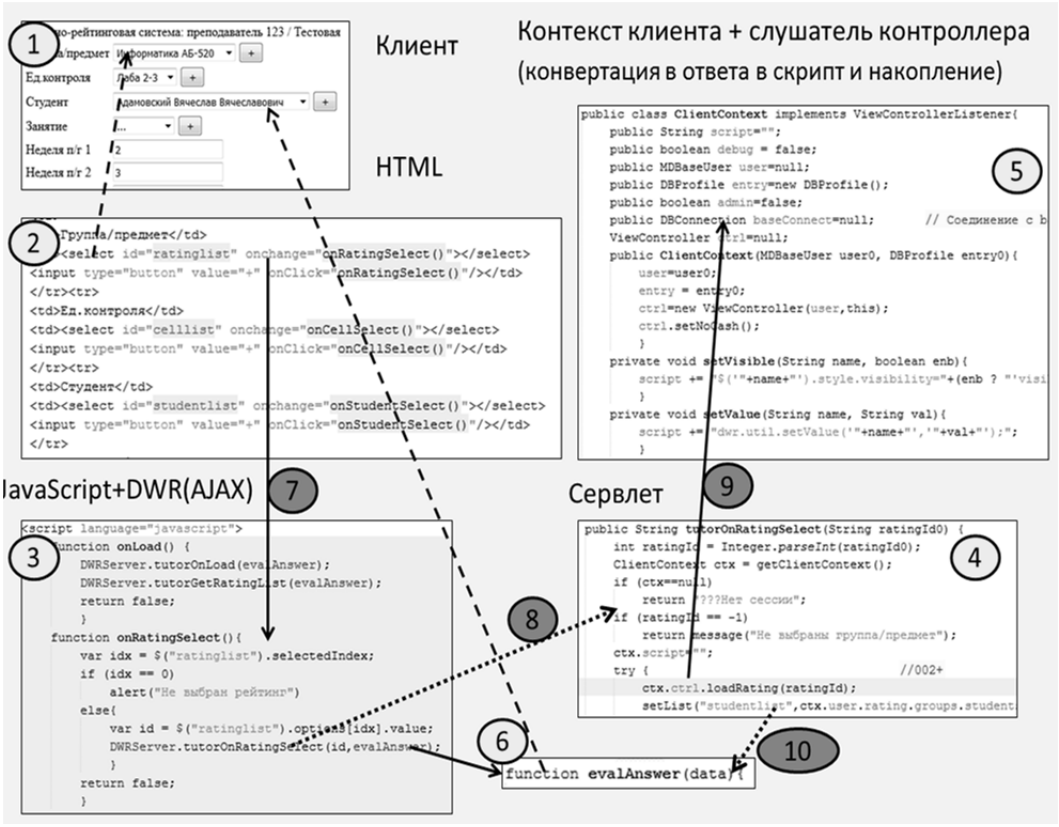


Рис. 7.27. Web-клиент с повторным использованием кода

- по окончании обращения к контроллеру накопленный текст скрипта возвращается клиенту (10);
- клиент, получая строку ответа, интерпретирует содержащийся в нем скрипт методом *eval* (6), который вызывается в функции асинхронного завершения запроса.

Параллелизм. Синхронизация

Специальных требований к параллелизму не предъявляется. В элементах графического интерфейса исполнение продолжительных операций происходит в фоновом режиме, при этом элементы управления не блокированы, но и не позволяют выполнять действия над данными, обрабатываемыми в фоне (мягкая блокировка).

Структура кода и объемные показатели

Для удобства повторного использования кода значительная часть классов выделена в ядро (core) [97]: контроллер для приложения преподавателя, бизнес-логика, классы DAO, коммуникации, коннекторы к БД, интерфейсы, что составляет 37 % кода в строках исходного текста [98–101] (SLOC) (рис. 7.28).

Компонента	SLOC	%	Пакет	Классов	Строк	Назначение
Core (ядро)	7816	37%	brs	2	325	Параметры настройки приложений
			brs.connect	4	828	Коннекторы к БД
			brs.model	29	3412	Бизнес-объекты
			brs.database	34	1207	Табличные объекты БД
			brs.ftpclient	4	880	ftp-клиент хранилища - не используется
			brs.controller	3	646	Контроллер бизнес-логики "преподаватель"
			brs.xml	8	244	Команды-ответы протокола WebAPI
			brs.interfaces	5	143	Интерфейсы компонент
			brs.cui	5	131	Объекты протокола ЦИУ
Android	4044	19%	brs.view	26	3755	Уровень представления (экраны)
			brs.sqlite	2	289	Коннектор БД SQLite (Android) - не используется
Desktop (все приложения)	7980	38%	brs.javaview	40	7748	Уровени представления и бизнес-логики (экраны) - все приложения, "преподаватель" - только представление
			view	1	232	Таблица с прокруткой
WebAPI	684	3%	brs.web	1	684	Сервлет WebAPI для тонкого клиента
Web	541	3%	brs.web	1	541	Java - серверная часть
					400	Клиентская часть (HTML+JavaScript)
Всего	21065					

Рис. 7.28. Объемные показатели проекта в SLOC

7.3. Архитектурное проектирование и прикладные протоколы

Преамбула. Существуют ли нерешенные проблемы?

Любая система, состоящая из клиентской и серверной части, использует протокол прикладного уровня. Он может быть построен на основе универсального сетевого соединения (TCP/IP) либо встроен в один из прикладных протоколов, например http, webSocket [89].

Если полагать, что надежность базовой сети достаточна, то разработчику остается только определить форматы сообщений и набор примитивов взаимодействия типа «запрос–ответ». Что обычно и делается. Тем не менее чисто протокольные проблемы имеют место и на прикладном уровне. Приведем несколько примеров:

- программные ошибки могут приводить к нарушению форматов сообщений и потере синхронизации;
- реальная сеть может быть ненадежна, например, имеют место мертвые зоны в мобильной связи. На прикладном уровне необходимо предусмотреть процедуры восстановления не только самих соединений, но и последователь-

ности принимаемых и передаваемых сообщений, а также средства отложенной передачи в состоянии «временно недоступен»;

- при длительном или предполагаемом отсутствии трафика соединение можно закрывать с сохранением контекста у клиента и сервера, а при появлении трафика – восстанавливать без повторной авторизации;
- обычной практикой является собственный *keep-alive*, поскольку при отсутствии трафика в соединении аварийное завершение одного из участников или промежуточного звена не будет замечено остальными достаточно долго;
- клиент может посылать сообщения, не дожидаясь ответа на предыдущее (требуется установление соответствия запросов и ответов);
- сервер также может являться инициатором взаимодействия, например, обновлять или запрашивать данные клиента (требуется распознавание сообщений во встречных потоках).

Специфика программирования протокольных процессов заключается в самой природе протокола. Напомним его определение. Протокол – описание правил взаимодействия параллельных независимых (асинхронных) процессов путем обмена сообщениями через ненадежную инертную пространственную среду.

Принципиальным моментом является то, что любой протокол не обеспечивает 100 %-ной надежной доставки сообщений с учетом перечисленных выше факторов, поэтому дублирование функций протоколов низших уровней вполне оправданно.

Аспекты описания протокола и его реализации

Программная реализация протоколов содержит много специфических моментов, связанных с особенностями протокольных процессов. Прикладной протокол и реализующие его процессы можно рассматривать в нескольких аспектах:

- **функциональный аспект** – функциональность протокола: виды взаимодействия, наличие механизмов восстановления, повышения надежности, синхронизации состояний;
- **структурный аспект** – все, что связано с передаваемыми данными: форматы сообщений, программные интерфейсы;
- **алгоритмический аспект** – принципы и алгоритмы реализации: общие механизмы взаимодействия, процедуры обмена, состояния протокола, автоматные модели, процедуры синхронизации состояний, восстановления, поддержки сессий;
- **процессный аспект** – особенности реализации протокола как системы процессов или потоков: параллелизм, потоки, синхронизация, производительность, задержки, очереди сообщений, буферизация данных.

Прикладной протокол как элемент взаимодействия слоев клиент-серверной архитектуры

Стандартным способом реализации прикладного протокола в клиент-серверной архитектуре является его встраивание в интерфейс между функциональными уровнями. Интерфейс «расшивается» следующим образом (см. рис. 4.13):

- функциональный слой клиента соединяется через интерфейс с клиентской компонентой протокола;
- клиент протокола взаимодействует через сеть с серверной компонентой протокола;
- в клиентской части протокола создается клиент протокола, а в серверной компоненте протокола – агент, который от имени клиента выполняет необходимые действия в следующем слое через расшитый интерфейс;
- наличие агента как представителя клиента, локально воспроизводящего его действия, обусловливает необходимость создавать, передавать от клиента или постоянно поддерживать необходимое окружение (контекст), в котором выполняются эти запросы.

Данный вариант соответствует наиболее простой стратегии взаимодействия – *синхронные запросы клиента к серверу* (см. раздел 4.3).

Параллелизм, синхронизация

Протокольный процесс обычно включает несколько параллельно исполняемых компонент. Необходимость их параллельного исполнения вызвана следующими причинами:

- в компоненте происходит ожидание каких-либо событий, например приема данных с блокировкой потока;
- по самой природе протокола компоненты работают одновременно;
- исполнение компоненты достаточно продолжительно, и в основном потоке это приведет к зависанию графического интерфейса приложения или других работ;
- компонента может иметь более высокий или более низкий приоритет исполнения.

Перечислим часто встречающиеся варианты реализации протокольного процесса:

- поток приема является необходимым, если используется синхронный прием данных, который сопровождается ожиданием с блокировкой текущего потока, например в классах *Socket* (протокол TCP/IP) или *URLConnection* (протокол HTTP);

- дополнительный поток приема необходим при использовании механизма отложенного опроса сервера (см. раздел 4.3 – *long polling*);
- поток используется для поддержки дополнительных соединений, например в FTP для передачи / приема файла вне основного потока сообщений протокола;
- поток передачи необходим, если время блокирования при синхронной передаче данных является существенным. Кроме того, чтобы исключить паузы в самом процессе передачи, данные сообщения сначала пишутся в байтный буфер *ByteArrayOutputStream*, а затем извлекаются единым массивом;
- для реализации тайм-аутов могут использоваться как обычные потоки, так и специальные классы, поддерживающие отложенные по времени события (*Timer, Handler*). В протокольном процессе тайм-ауты могут иметь место везде, где возможны зависания и блокировки:
  - тайм-аут замыкания петли «запрос–ответ» или получения подтверждения приема – «тайм-аут передачи»;
  - тайм-аут попытки восстановления соединения при его временном пропадании;
  - тайм-аут проверки активности соединения. На клиентской стороне периодический тайм-аут используется для передачи сообщения *keep-alive*. Сервер использует более продолжительный тайм-аут для контроля поступления сообщений;
  - тайм-аут закрытия соединения *shutdown*;
  - тайм-аут ожидания приема исключает зависание на приеме сообщения после начала приема при обрыве его передачи.

Замечание по теме. Указанные тайм-ауты не являются чем-то вычурным. Скорее наоборот, следует исходить из того, что все, что может зависнуть, зависает. Аналогичный эффект производят ошибки программирования, когда ожидаемая последовательность данных может нарушаться либо могут возникать взаимные блокировки – клинчи.

Буферизация, очереди

Очереди сообщений и соответственно необходимость буферизации возникают в двух случаях:

- клиент поддерживает несколько потоков, создающих запросы к серверу, либо имеет несколько независимых источников запросов, вызываемых событиями. В одно соединение мультиплексируются запросы от разных источников;
- клиент может посылать запрос, не дождавшись завершения предыдущего.

Замечание по теме. Иногда явная буферизация отсутствует, но в потоке приема имеется синхронизация к основному потоку, которая на каждое принятое сообщение вызывает метод синхронизации, а он фактически ставит дальнейший исполняемый код (*Runnable*) во внутреннюю очередь, а уже с ней работает основной поток. В любом случае желательно, чтобы данные принимались в динамически создаваемые объекты-буферы.

Диспетчеризация, планирование

Иногда вместо обычной буферизации сообщений применяется *буферизация кода* «запроса–ответа» в целом. Она заключается в последовательном исполнении в одном потоке кода всех поступающих запросов и реализуется шаблоном проектирования «планировщик» (scheduler) (см. раздел 3.3). Интерфейс объекта-запроса на исполнение планировщиком имеет методы, в которых прописываются:

- код, исполняемый планировщиком в собственном потоке;
- код, исполняемый в основном потоке и вызываемый при завершении предыдущего;
- код обработки исключений, возникших при исполнении указанных выше кодов;
- код реакции на отмену планирования.

Диаграмма состояний протокола

Для того чтобы логически связать воедино все параллельно исполняемые компоненты протокольного процесса и исключить ошибки их взаимодействия, необходима единая модель описания поведения протокольного процесса. В этом качестве часто используются конечные автоматы и их графическое представление в виде диаграмм состояний (см. разделы 2.2, 4.4).

С технологической точки зрения все события, которые изменяют или проверяют состояние конечного автомата (команды верхнего уровня, тайм-ауты, принятые сообщения), должны быть синхронизированы между собой. Объектом синхронизации может быть сам автомат.

В простейших случаях диаграмма состояний отдельно описывает поведение клиентской или серверной компоненты, например диаграмма состояний соединения в прикладном протоколе со стороны клиента, изображенная на рис. 7.29. По диаграмме состояний легко анализируется и тестируется корректность поведения автомата при различных последовательностях событий, определяются возможные блокировки.

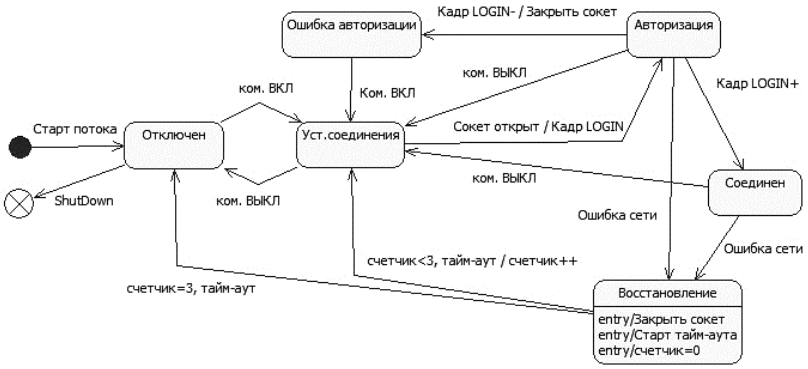


Рис. 7.29. Диаграмма состояния соединения с сервером (прикладной уровень)

В случае если конечный автомат используется для описания состояния обоих сторон соединения, то фактически речь идет о взаимодействии двух идентичных автоматов, которое может приводить к их взаимной блокировке, условия ее возникновения не очевидны из самой диаграммы. В качестве примера можно привести диаграмму состояний TCP-соединения [77]. Верификация таких моделей – тема отдельного обсуждения.

Установление, восстановление и закрытие соединения, сессия, авторизация

В протокольном процессе могут использоваться собственные средства повышения надежности и эффективности соединения, например:

- при частых разрывах соединения в протоколе создается состояние «временно недоступен» с отложенной передачей запросов серверу. При этом, с точки зрения клиента, соединение продолжает быть работоспособным;
- при длительном или предполагаемом отсутствии трафика соединение закрывается с сохранением контекста у клиента и сервера, а при появлении трафика – восстанавливается без повторной авторизации;
- передача собственного *keep-alive* для обнаружения краха или неработоспособности одного из участников соединения или среды передачи.

Процедура авторизации клиента на сервере также может быть составной частью протокольного процесса, а не вышележащих уровней. Тогда состояние доступности сервера для протокола определяется не только созданием соединения, но и успешно проведенной авторизацией.

В протокольный процесс также может быть включен механизм сессий (см. разделы 3.3, 4.4 – восстановление долгоживущих соединений).

Канальная и прикладная компоненты протокольного процесса

Прикладной протокол поддерживает множество примитивов взаимодействия «клиент–сервер». Если в протоколе принимается *единый формат* представления всех сообщений и единый способ их передачи, то функции оформления упаковки запросов / ответов в сообщения, их буферизация и непосредственная передача в сетевое соединение могут быть реализованы **канальной компонентой – линейным драйвером**. Это позволяет разделить содержательную и транспортную компоненты протокола.

Стандартным способом унификации с использованием технологии ООП является создание отдельного класса сообщений для линейного драйвера:

- сообщение содержит код, соответствующий команде или ответу;
- специальные коды сообщений могут использоваться для общепринятых ответов: положительное завершение без параметров, ошибка, исключение в серверной компоненте, а также сообщения для собственных нужд протокола;
- сообщение может содержать несколько полей для наиболее часто используемых типов параметров;
- преобразование запросов / ответов в сообщение может происходить поразному:
  - параметры запросов / ответов передаются как данные класса сообщений;
  - запросы / ответы являются производными классами, которые сериализуются при передаче и приеме;
  - объекты запросов-ответов ссылаются на объект с дополнительными данными, в протоколе прикрепленный объект следует за основным как отдельное сообщение.

Процедурная и объектно-ориентированная реализация протокола

Обычная реализация набора примитивов взаимодействия выглядит следующим образом:

- команды и ответы примитивов кодируются уникальными кодами;
- клиентская компонента протокола для каждой команды создает сообщение с соответствующим кодом, заполняет параметры и передает его линейному драйверу;
- серверная компонента, приняв сообщение, выполняет переключатель (*switch*) по коду сообщения, в каждой ветке переключателя пишется код обработки;

- в зависимости от результата исполнения запроса серверная компонента создает сообщение с соответствующим кодом;
- клиентская компонента при приеме ответа выполняет ветвление или переключение по коду ответного сообщения, в каждой ветке пишется свой код завершения.

Такая реализация является немодульной, поскольку код обработки всех запросов «свален в одну кучу», то же самое касается обработки ответов. Альтернативным решением является использование технологии ООП:

- все сообщения имеют общего предка – базовый класс, содержащий данные протокола, необходимые для работы с потоком сообщений;
- выполняется принцип «для каждого типа запроса свой производный класс»;
- в классе сообщения имеются полиморфные методы *onServer* и *onClient*, которым передается необходимый контекст для обработки запроса на сервере и ответа на клиенте, в них прописывается соответствующий код обработки.

Таким образом, работа прикладной компоненты протокола состоит в передаче клиентом объекта класса, соответствующего запросу, в его методе *onServer* прописана процедура обработки, создания и передачи объекта-ответа. В методе *onClient* того же класса прописана процедура обработки ответа на клиенте, причем в качестве параметра он получает и ответное сообщение. Код становится максимально модульным, сам протокол необходимо снабдить диаграммой классов с указанием зависимостей порождения одних классов другими.

Итоговый вопросник

Прикладной протокол, являясь связующим элементом архитектурных компонент, должен соответствовать требованиям, которые предъявляются к их взаимодействию. Поэтому те вопросы, которые необходимо задавать по структуре протокола, следует сначала задать на архитектурном уровне. Все они уже обсуждались, теперь соберем их воедино.

1. Какие компоненты, слои, подсистемы связывает протокол?
2. Какие элементы функционала определяют форматы передаваемых сообщений протокола (классы предметной области, бизнес-объекты)?
3. Какие локальные интерфейсы «расшиваются» для удаленного доступа через данный протокол? Требуется ли прозрачная реализация этих интерфейсов? Когда для локальной и удаленной работы используется один интерфейс с разными реализациями?
4. Какие средства синхронизации данных используют связываемые компоненты? Изменяются ли данные прикладного уровня одним или обоими участниками взаимодействия?

5. Какие примитивы используются для поддержания целостности данных (получение данных по запросу клиента, контроль и передача обновлений сервером и т. д.)?

6. В какой из компонент происходят события, инициирующие взаимодействие? Кто преимущественно играет роль клиента?

7. Необходимо однонаправленное или двунаправленное, синхронное или асинхронное взаимодействие?

8. Какой стандартный вид линейного драйвера передачи сообщений технологически лучше всего подходит для реализации протокола (сокеты, http-соединение, webSocket)?

9. Необходимо ли создавать собственные средства повышения надежности передачи, контроля состояния и восстановления соединений?

10. Взаимодействие происходит в виде постоянных соединений, восстанавливаемых соединений или транзакций без сохранения контекста?

11. В каком виде осуществляется поддержка контекста соединения (объекты, привязанные к соединению, внешняя поддержка сессий средствами окружения)?

12. Необходимо ли включать в протокол сообщения о сбоях / исключениях, происходящих при обработке запросов на прикладном уровне на сервере, транслировать исключения клиенту?

13. Необходимы ли средства трассировки протокола и сбора статистики при его отладке и эксплуатации?

14. Какие форматы сообщений использует протокол (собственные форматы, двоичная сериализация, XML- или JSON-сериализация объектов-сообщений)?

15. Какие элементы параллелизма необходимы в реализации протокола (потоки, синхронизация, буферизация, планирование)?

16. Предполагает ли протокол открытое использование, например, в виде webAPI? В каком виде необходимы его описания и спецификации?

17. Какие стандартные технологии будут применены в реализации его компонент?

18. Какие стандартные средства защиты данных из нижележащих уровней будут использованы и требуются ли собственные средства защиты?

На большинство вопросов нельзя ответить, не имея полного представления о принятых архитектурных решениях и общих требований к разработке.

ГЛАВА 8
ТЕСТИРОВАНИЕ

8.1. Тестирование, валидация, верификация

Сомневайся во всем.
Рене Декарт

Тестирование выглядит как формальный и нетворческий вид деятельности, сопровождающий процесс разработки ПС, с которым поневоле приходится мириться. Тестировщики считаются низшей кастой программистов, не способных к написанию собственного кода. Однако с усложнением программного проекта роль тестирования возрастает, также увеличивается число видов деятельности, именуемых тестированием, и число подлежащих тестированию артефактов.

Особую роль тестирования в программной инженерии придают сложность объекта проектирования и отсутствие рамочных законов верификации проекта в целом (см. разделы 1.1, 1.2).

Тестирование – весьма многоаспектная тема. В широком смысле к ней можно отнести все виды деятельности, связанные с *испытанием* артефактов процесса проектирования.

Тестирование (в широком смысле) – целенаправленная деятельность, направленная на выявление ошибок в различных артефактах проекта (цели, требования, модель предметной области или анализа, архитектура, код, сборка, релиз, документация) и в проекте в целом.

Тестирование как соответствие требованиям

Одно из положений о структуре тестировании – тезис о соответствии иерархии требований к системе и проверяющих эти требования тестов (рис. 8.1).

Рис. 8.1. V-образная модель тестирования

Здесь во главу угла ставятся требования, на что можно резонно возразить: «А что гарантирует нам качество (валидность, непротиворечивость, обоснованность) самих требований?»

Тестирование на различных этапах жизненного цикла программы

В V-образной модели речь идет об исполнении тестов на готовом продукте – релизе или прототипе. Но тестированию может подлежать и любой другой артефакт процесса разработки, например, видение проекта, модель предметной области или классов анализа. Тесты в этом случае будут выглядеть как умозрительные испытания при разнообразных внешних воздействиях. Например, модель классов анализа можно тестировать с помощью сценариев исполнения функционала: при выполнении шагов сценария проверяется, могут ли быть извлечены из модели связанные данные, необходимые для его исполнения.

Второй аспект тестирования – его привязка к жизненному циклу программы и процессу проектирования. Здесь все просто: берем любой артефакт и добавляем к нему слово «тестирование»:

- тестирование модели предметной области (фаза исследования);
- тестирование требований (фазы исследования и развития);
- тестирование прецедентов и сценариев (фаза развития);
- тестирование модели классов анализа (фаза развития);
- тестирование архитектурного прототипа (фаза развития);

- тестирование модели графического интерфейса пользователя (фаза развития);
- тестирование документации (фаза развертывания).

И наконец, готовый программный продукт как самый важный артефакт подлежит многообразному тестированию в зависимости от проверяемых характеристик или устанавливаемых фактов на этапе *системного тестирования*:

- приемочное тестирование при передаче заказчику;
- установочное тестирование для проверки правильности инсталляции;
- альфа- и бета-тестирование при пробной эксплуатации продукта внутри организации или на группе пользователей;
- тестирование на соответствие требованиям спецификаций;
- тестирование надежности и устойчивости к сбоям;
- регрессионное тестирование – повторное тестирование после внесения изменений;
- тестирование производительности;
- нагрузочное тестирование (стресс-тестирование) – тестирование при повышенной рабочей нагрузке;
- сравнительное тестирование различных версий;
- восстановительное тестирование для проверки работоспособности после восстановления;
- конфигурационное тестирование;
- тестирование Useability (пользовательских характеристик системы).

Тестирование, валидация, верификация

Тестирование – один из инструментов *контроля качества ПО* наряду с **валидацией** и **верификацией.** Взаимоотношения между ними выглядят поразному в зависимости от тонкостей определения терминов:

- термин «тестирование» в широком смысле понимается как деятельность;
- термин «тестирование» в узком смысле трактуют как испытание артефакта на наборе входных данных. В еще более узком смысле он относится к тестированию программного кода;
- термин «верификация» как проверка на соответствие требованиям включает тестирование в узком смысле как вариант динамической верификации.

В самом простом случае отмечается связь терминов с уровнями проверяемых артефактов проекта (рис. 8.2).

Приведенная схема не раскрывает сущности деятельности, которая лежит в их основе. Рассмотрим ее подробнее.

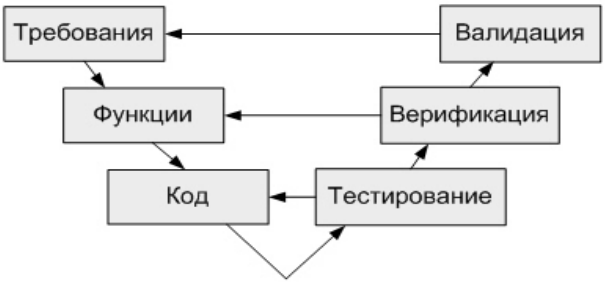


Рис. 8.2. Валидация, верификация, тестирование как уровни испытаний

В основе контроля качества ПО лежат верификация и валидация. **Верификация** – это проверка системы на соответствие требованиям, а **валидация** – проверка того, насколько сами требования и система соответствуют целям, поставленным при ее создании. Валидация относится в основном к артефактам бизнес-анализа и фазы исследования (бизнес-требования, видение проекта) и касается полезности проекта в целом.

Структуру верификации удобно рассматривать в системе координат «анализ–испытание» и «код–остальные артефакты» (рис. 8.3). Тогда можно определить принципиальную разницу между тестированием и остальными методами верификации следующим образом:

- **тестирование** – испытание экземпляра артефакта на заданных входных данных (сценарий, тестовый набор), анализ поведения, проверка на ошибки;
- **верификация** – формальный или содержательный анализ артефакта как сущности.

В каком-то смысле это соответствует принятым в UML понятиям «классификатор» и «экземпляр».

Валидация – образно-содержательный анализ соответствия бизнес-требований системы целям ее создания.
Верификация – проверка на соответствие разработки функциональным требованиям, стандартам качества, анализ на предмет непротиворечивости.
Формальная верификация – способ формального доказательства правильности программного кода путем анализа утверждений о состоянии данных и их преобразовании операциями и операторами программы.
Тестирование (в узком смысле) – воспроизведение поведения артефакта при заданном входе (испытание).
Тестирование кода – исполнение готового кода (прототипа или релиза) на тестовых данных или рабочей нагрузке.

Рис. 8.3. Взаимоотношение валидации, верификации и тестирования

Основные методы верификации [64] представлены на рис. 8.4. При этом все они применимы к программному коду, остальные артефакты жизненного цикла могут быть проверены только с помощью экспертизы:

- *экспертиза* – анализ и оценка артефактов специалистами:
  - *техническая экспертиза* – систематический анализ артефактов проекта квалифицированными специалистами для оценки их внутренней согласованности, точности, полноты, соответствия стандартам и принятым в организации процессам, а также соответствия друг другу и общим задачам проекта;
  - *сквозной контроль* – метод экспертизы, в рамках которого один из членов команды проверки представляет ее участникам последовательно все характеристики проверяемого артефакта, а они анализируют его, задавая вопросы, внося замечания, отмечая возможные ошибки, нарушения стандартов и другие дефекты;
  - *инспекция* – последовательное изучение характеристик артефакта по определенному плану, с целью обнаружения в нем ошибок и дефектов;
  - *аудит* – сторонний анализ артефактов с целью исключения субъективных оценок или привлечения опытных специалистов;
- *статический анализ* – структурный анализ кода на предмет поиска потенциальных угроз и ошибок. Элементы статического анализа кода могут включаться в среды разработки и компиляторы, а также в семантику языка. В Java к таким средствам относятся: проверка переменных на инициализируе-

мость, проверка на достижимость участков кода, контроль перехвата исключений и т. п.;

- *формальные методы* – способы формального доказательства правильности программного кода путем анализа утверждений о состоянии данных (предикаты) и их преобразовании операциями и операторами программы;
- *динамические методы* – основаны на исполнении программного кода и включают:
  - *мониторинг* – непосредственное наблюдение за исполнением;
  - *профилирование* – сбор данных об используемых ресурсах, состоянии программы и окружения во время ее исполнения;
  - *тестирование* – исполнение программы на тестовых данных (по формальным признакам также относится к динамическим методам верификации).

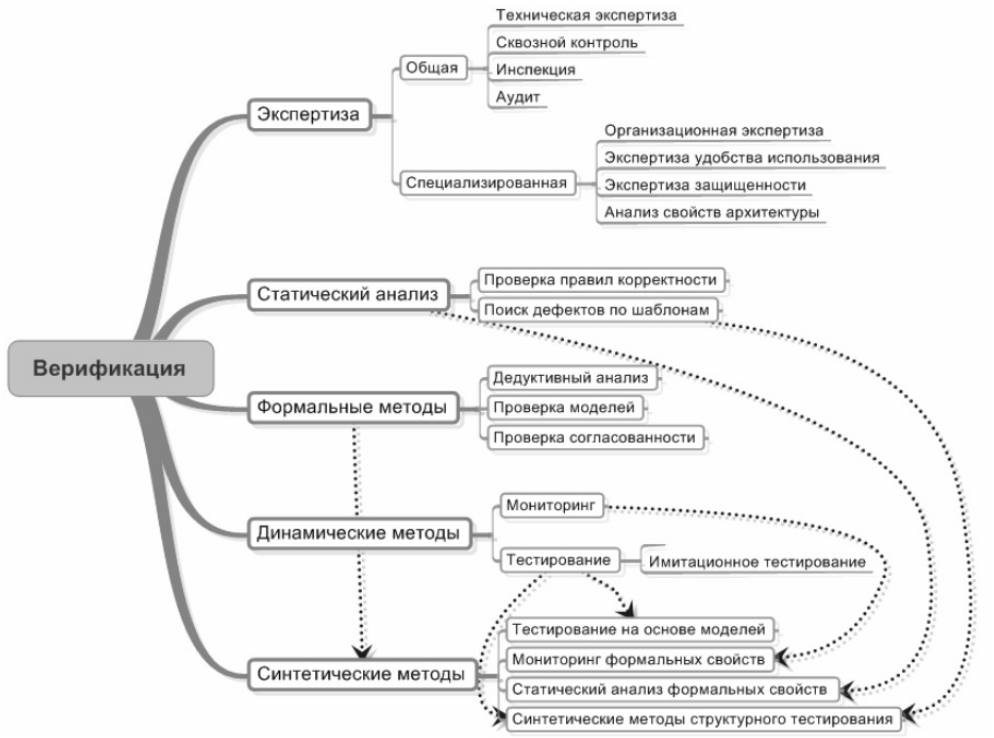


Рис. 8.4. Методы верификации программного обеспечения

На житейском уровне к верификации кода можно отнести проверку утверждений, формулируемых для *соглашений по данным* и *инвариантам*:

- утверждения о состоянии структур данных – *соглашения по данным*. При проектировании структур данных оговариваются допустимые конфигурации элементов и их значений, которые обязаны соблюдаться всеми компонентами программного кода, работающими с ними, например всеми методами класса. Соглашения по данным должны минимизировать возможные сочетания представлений данных;
- утверждения об условиях, сохраняемых на последовательных шагах алгоритма – *инварианты*. При конструировании и тестировании циклов и рекурсивных алгоритмов весьма продуктивна формулировка условий, сохраняемых на каждом шаге цикла или рекурсивного алгоритма. Обычно это согласованный набор значений переменных или формальных параметров.

Эмоции и ничего личного. В основе тестирования лежит сомнение или занудство: действительно ли артефакт является идеальным, работает без ошибок. Настрой разработчика и тестировщика прямо противоположен: первый уверен в своей непогрешимости, второй стремится ее развенчать. Это противоречие решается путем вовлечения разработчика в процесс тестирования, чтобы сделать тестирование составной частью процесса разработки кода.

Рис. 8.5. Виды тестирования

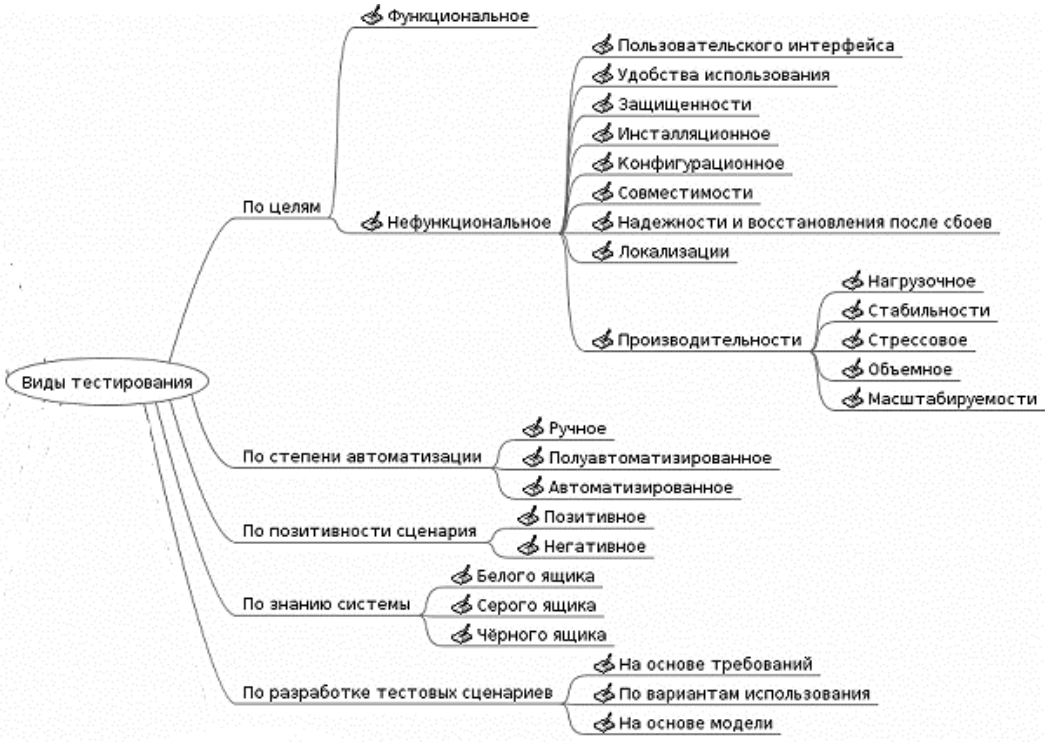


Рис. 8.6. Виды тестирования (продолжение)

В заключение перечислим [63] перечень видов тестирования (рис. 8.5, 8.6).

8.2. Программные ошибки

Ошибки надо не признавать. Их надо смывать. Кровью!

Из кинофильма «Кавказская пленница, или Новые приключения Шурика»

Тестирование кода связано с поиском ошибок в программе, снижением вероятности их присутствия. С ними же связаны отладка и инспекция кода. Поэтому начнем с резонного вопроса.

Что такое программная ошибка?

Казалось бы простое понятие «программная ошибка» при ближайшем рассмотрении порождает много вопросов – от риторических до философских. Рассмотрим несколько определений.

Определение по принципу «само собой разумеется» подразумевает, что ошибка – это когда программа работает неправильно. Если программа не компилируется, зацикливается или падает, то она содержит ошибку. В остальных случаях можно задать резонный вопрос: «А судьи кто?». Одно дело, если результат программы можно явно оценить как ошибочный. Но существуют и обратные примеры:

- результатом поискового алгоритма является решение, для оценки оптимальности которого требуется такой же, но заведомо правильно реализованный алгоритм, т. е. оценка достоверности результата соизмерима с затратами на его достижение;
- в одних случаях периодическое падение программы считается ошибкой, ведущей к катастрофическим последствиям, а в других случаях с этим можно смириться, т. е. программа не выходит за пределы допустимых потребительских качеств.

Определение технико-бюрократическое подразумевает, что программа содержит ошибку, если ее работа не соответствует спецификации, техническому заданию. Не все требования, касающиеся корректности работы программы, могут быть отражены в задании, представления о результатах работы программы могут быть неадекватно сформулированы, возможность проверки некоторых требований может оказаться под вопросом, да и сама спецификация может быть неполной.

Определение технологическое подразумевает, что программа не содержит ошибок компиляции, сборки или времени выполнения. Слишком слабое требование, сводящееся к тому, что любая программа, которая транслируется и выдает результат, правильная. Тем не менее языковая среда является *первым уровнем тестирования* – она накладывает ограничения на возможные формы записи программы и контролирует процесс ее исполнения. В связи с этим надо заметить, что чем больше свободы имеет программист в синтаксисе языка и чем меньше ограничений и контроля со стороны системы исполнения программы, тем выше вероятность наличия *необнаруженных ошибок* на этом уровне, которые перекладываются на следующие этапы. Здесь классическим примером является язык Си с его архитектурно-ориентированной моделью исполнения программы – адресная арифметика, указатели и т. п.

Программная ошибка содержит два технологических элемента:

- причину ошибки – **дефект** (фрагмент кода, по вине которого возникает ошибка);
- следствие – **сбой** (некорректное поведение программы, выражающееся в фатальной ошибке, зацикливании, нарушении целостности данных, дефектах структур данных, ошибках в выходных данных и т. п.).

Философия ошибок

Если понимать программную ошибку как *отклонение в поведении программы от ожидаемого*, то следует прежде всего определить, где находится идеал, с которым производится сравнение. Вообще-то он должен быть сформулирован в *требованиях* к программе на различных этапах ее проектирования, несоответствие которым может рассматриваться как ошибка. Здесь не будем рассматривать функциональные требования и ошибки функционала, а сосредоточимся на тех, которые связаны непосредственно с исполнением кода.

Формальная корректность результата. Если рассматривать программу как простое преобразование входного набора данных в выходной, то формальная правильность работы программы определяется соответствием ее результата ожидаемому набору значений. При этом возможны различные способы оценки результатов и автоматизации:

- результат очевиден или может быть вычислен вручную. Такой метод применяется, например, при отладке – путем выполнения программы со статическими тестовыми данными ограниченной размерности. Другой пример – поисковая задача, если решения существуют всегда и необходимо лишь нахождение первого подходящего;
- результат удовлетворяет формальным критериям и проверяется, затраты на проверку соизмеримы с затратами на тестирование. Например, при тестировании сортировки результат тестирования может быть проверен формально – по расположению элементов в порядке возрастания (попарное сравнение соседних) и соответствию элементов входа и выхода;
- результат не очевиден. Для его проверки необходимо решить эту же задачу другим способом. Например, поисковая задача должна возвращать оптимальное решение при возможном его отсутствии и наличии субоптимальных решений;
- в области тестирования существует термин «ожидаемое поведение». Не всегда ожидаемое поведение легко распределить по принципу «ошибка – терпимый недостаток».

Требования по производительности. Программа должна обеспечивать не только достоверный результат, но и получать его за приемлемое время, что соответствует требованиям производительности. При формулировании требований производительности нужно учитывать следующее:

- производительность зависит от архитектуры аппаратных средств и системного программного обеспечения;

- производительность программ является производной от трудоемкости используемых алгоритмов. Она, в свою очередь, может иметь различную зависимость от размерности входных данных. Поэтому в требованиях к производительности следует указывать диапазоны размерностей данных, на которых она обеспечивается;
- может иметь место чувствительность алгоритмов к данным – разброс трудоемкости и производительности на разных наборах входных данных.

Требования реального времени. Некоторые компоненты программы должны реагировать на заданные внешние воздействия в течение ограниченного интервала времени. Несоблюдение этих требований является ошибкой, последствия которой могут быть катастрофическими. К требованиям реального времени также относится *реактивность программы,* т. е. способность давать отклик на действия пользователя в ограниченное время. Требования этого вида сильно зависят от архитектурных аппаратных и программных решений.

Требования надежности, устойчивости и безопасности также могут составлять существенную часть требований для программ, работающих в условиях повышенной ответственности за возможный сбой.

Требования эргономики и человеческий фактор. Большинство общих требований графического интерфейса, а также часть требований производительности и реактивности базируются на субъективных, в лучшем случае экспертных оценках в категориях «удобный», «дружественный», объединяемых общим термином *Useability.* Формализация и количественная оценка требований здесь затруднена, ошибки в этой области проявляются не только в виде снижения потребительских свойств программы, но и более серьезно, например:

- нанесение ущерба случайным исполнением последовательности команд (отсутствие «защиты от дурака»);
- снижение общей производительности работы пользователя из-за излишних действий, неудобного интерфейса и т. п.

Все перечисленные требования могут быть положены в основу идеальной модели, в сравнении с которой и определяется факт наличия ошибок. Отсюда следует определение тестирования: **тестирование – целенаправленный процесс испытания программы в различных условиях с целью выявления ошибок путем сравнения ее поведения с эталонной моделью.**

Кроме того, программа функционирует не в вакууме, а в рамках среды исполнения, операционной системы, аппаратных средств, что может порождать значительное количество нюансов из-за ограничения ресурсов, использования форматов данных и API, производительности, параллелизма и т. п. Все это также может приводить к *программным ошибкам, вызванным окружением.*

Существуют также *ошибки синхронизации,* связанные с внутренним программным параллелизмом, они могут появляться только при определенных сочетаниях временных интервалов следования внешних событий.

Таким образом, область тестирования может расширяться бесконечно как за счет требований, так и за счет учета факторов, которые способны повлиять на ее работоспособность. Аналогично ведет себя и граница между ошибкой и «милым недостатком», с которым приходится мириться.

В связи с этим следует обсудить известный тезис: «Любая программа содержит ошибку». Он является следствием *невозможности полного тестирования программы* путем простого перебора всех возможных вариантов ее исполнения в различных условиях, в число которых входят:

- возможные сочетания наборов данных;
- возможные последовательности входных данных и команд;
- возможные архитектурные комбинации аппаратно-программного окружения;
- возможные временные сочетания событий, влияющих на поведение программы.

Поэтому тестирование не может гарантировать отсутствие ошибок в программе, а стремится выявить их и понизить вероятность их появления за счет следующих процессов:

- выявление факторов, влияющих на появление ошибок в программе;
- определение возможных видов ошибок, которые могут появиться в программе;
- разработка методики тестирования, сценариев и тестов, проведение тестирования;
- включение в программу кода, повышающего устойчивость программы к ошибкам.

Невозможность тотального тестирования. Для всестороннего тестирования требуется полный перебор сочетаний наборов данных, последовательностей входных данных и команд, вариантов аппаратно-программного окружения и временных сочетаний внешних событий.

Защита от ошибок и устойчивость программы

То, что программу нельзя тотально протестировать, означает, что она сама должна иметь средства для анализа и нейтрализации как собственных ошибок, так и вызванных окружением. В целом это можно назвать «защитой от дурака»,

если под дураком понимать всю окружающую физическую и логическую реальность. Поскольку большинство ошибок окружения может быть обнаружено вызовом соответствующих функций проверки состояния либо сопровождается исключениями, то задача программиста состоит в том, чтобы заниматься этой рутиной в программном коде. Что же касается собственных багов, то те из них, которые приводят к исключениям, программа также должна обнаруживать, фиксировать и нейтрализовывать. Перечислим основные виды ошибок, от которых следует защищаться программе:

- «защита от дурака» – проверка действий пользователя, возможно приводящих к некорректным результатам или деградации производительности (копирование больших массивов, недопустимые форматы ввода, перезапись файлов и т. п.);
- ошибки форматов – недопустимые форматы файлов или сообщений программ вследствие ошибок открытия / соединения или намеренного искажения;
- ошибки предоставления ресурсов – отсутствие ресурса или его ограниченность;
- сбои и аварийные завершения, например закрытие сетевого потока до окончания передачи файла.

Кроме средств самой защиты необходимы *средства фиксации ошибок*: сохранение максимально возможной информации об ошибке – дата, время, версия программы, окружение, стек вызова, текущие данные, запись данных в лог-файл, по возможности отправка на сервер сопровождения.

Тестирование и отладка

С тестированием связан еще ряд деятельностей (рис. 8.7), прежде всего отладка – обязательный начальный этап тестирования, связанный с получением минимально работоспособной версии программы (альфа-версии), годной для последующего тестирования. Принципиальная разница тестирования и отладки: задачей тестирования является обнаружение *факта ошибки,* а задачей отладки – *локализация и исправление уже известной.*

Специфика процесса отладки

Специфика процесса отладки заключается в обнаружении условий и закономерностей появления дефекта и в его локализации. Далее – набор общеизвестных банальностей.

Минимизация тестовых данных. Для упрощения поиска дефекта необходимо уменьшить набор входных данных до того минимума, на котором он проявляется. Лучше всего для этого использовать статические данные тестовых модулей.

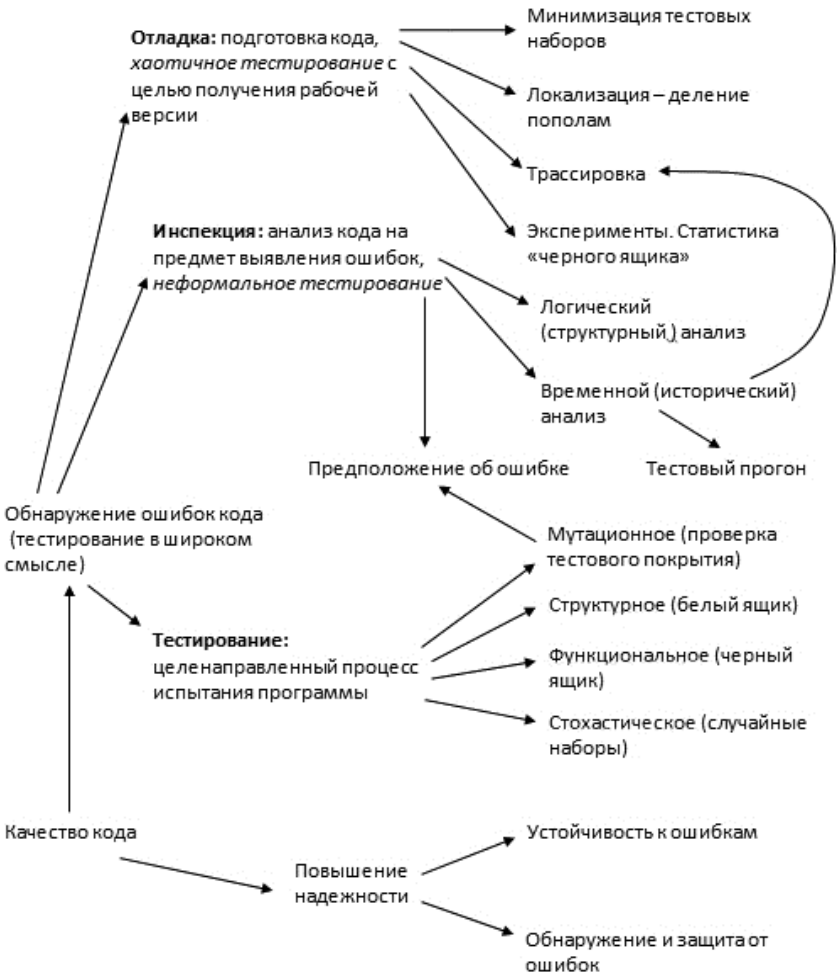


Рис. 8.7. Деятельности, связанные с тестированием кода

Эксперименты над программой. Статистика черного ящика. Если минимизировать данные не удается, необходимо набрать статистику ошибочного и правильного исполнения программы над разными наборами данных, т. е. выполнить некоторый вариант функционального тестирования по методу черного ящика. Тестовые наборы следует готовить, исходя либо из предположений

о характере ошибки, либо из анализа функционала программы, т. е. в виде наборов граничных значений.

Локализация дефекта. Метод половинного деления. Локализация дефекта проводится с помощью отладчика. Участок программы или последовательность шагов ее работы делится на две части точкой останова, в которой проверяется корректность промежуточных результатов. В зависимости от этого выбирается часть кода или последовательность шагов, содержащая дефект.

Трассировка против отладчика. Для многих ошибок необходим анализ поведения программы во времени, цели которого могут быть разными:

- обнаружение аномальных закономерностей неправильно работающей программы;
- локализация дефекта – определение шага программы или диапазона данных, на котором он проявляется.

Для таких целей отладчик не только не нужен, но и вреден. Полезным средством является лог-файл либо обычная функция вывода в консоль нужных данных в нужных точках: одновременное созерцание позволяет интуитивно или логически определить полезные закономерности. Кроме того, трассировка необходима, если дефект проявляется при сложных сочетаниях данных, которые трудно уловить обычным отладчиком.

Инспекция кода. Предположение об ошибке. Большой опыт отладки позволяет делать предположения о возможной причине ошибки и месте локализации дефекта на основании интуиции в сочетании с инспекцией программного кода.

Особенности анализа программ, содержащих ошибки

Анализ поведения программы, содержащей ошибку, может быть значительно более сложным по сравнению с работающей правильно, для которой известно, что установленные *инварианты и соглашения по данным* соблюдаются. Для программы с ошибкой это недостоверно, поэтому ее поведение может быть довольно парадоксальным.

Характеристики ошибок

Дилетант не в состоянии оценить масштабы проблем, поскольку не представляет их многообразия. Это относится и к программным ошибкам. Для начала перечислим их характеристики, потом – основные типы и примеры дефектов.

Условия проявления ошибки

Ошибки могут проявляться при различных условиях, связанных как с самой программой, так и с ее окружением.

Ошибка при определенном значении или наборе значений. Ошибка детерминированно проявляется при некотором значении или наборе значений, имеющих отношение к данным одного вида.

Ошибка при определенной последовательности значений. Ошибка детерминированно проявляется при определенном сочетании значений, имеющих отношение к данным различных видов либо вводимых в разное время в определенной последовательности.

Ошибка при определенных временных сочетаниях (гонки). Если в программе имеется внутренний параллелизм, то некоторые ошибки синхронизации могут проявляться только в определенных сочетаниях последовательности исполнения кодов из разных потоков. Так как потоки могут исполняться с произвольными скоростями, то ошибки такого рода являются *мерцающими*, т. е. они могут возникать в случайные моменты времени без каких-либо закономерностей.

Ошибки со случайным временем появления. Случайным временем появления могут характеризоваться многие другие ошибки:

- код, содержащий дефекты, может вызываться внешними событиями, происходящими в случайные моменты времени;
- для выполнения некоторых операций может не оказаться необходимых ресурсов;
- дефект вызывается при накоплении определенного количества данных;
- дефект вызывается при определенной последовательности команд пользователя или последовательности вызова методов, сама последовательность возникает случайным образом.

Момент проявления

Фактическое проявление программной ошибки не всегда может совпадать *с выполнением ошибочного кода.* Часто ошибка приводит к нарушению логической структуры данных (соглашений по данным), что приводит к ошибкам при их последующем использовании. Например, некорректное освобождение динамической памяти в языке Си приводит к краху программы при последующих операциях резервирования памяти. В этом случае вторичную ошибку можно назвать *наведенной*, хотя фактически она ошибкой не является. Сказанное не относится к ситуации, когда ошибочно вычисленные данные хранятся до момента их использования или визуализации.

Характер последствий, уровень ущерба

Снижение потребительских свойств программы. Программа может иметь неудобный интерфейс, снижающий производительность пользователя при работе с ней, подвисать при исполнении длительных операций, иметь невысокую «защиту от дурака».

Локальная устранимая ошибка. Ошибочное действие (результат) легко идентифицируется, и для устранения ошибки достаточно изменения или добавления кода.

Частичный восполнимый ущерб. Потеря или искажение результатов и данных, которые могут быть восстановлены повторением некоторой последовательности действий. Например, не сохраняется файл изменений, производимых последовательностью команд.

Частичный невосполнимый ущерб. Потеря или искажение результатов и данных, неисполнение действий, для восстановления или повторения которых требуются неординарные процедуры либо такие процедуры отсутствуют.

Катастрофа. Последствия программной ошибки связаны со значительным материальным ущербом, опасностью для здоровья и жизни, переходом в аварийный режим работы организации или предприятия.

Воздействие на программу

Ошибки оказывают влияние как на результат работы программы, так и на ее работоспособность.

Наблюдаемый или тестируемый ошибочный результат – несоответствие результата программы ожидаемому.

Нарушение целостности внутренних данных – дефект структуры данных. Программные ошибки могут приводить к тому, что конфигурация внутренних данных не будет соответствовать тем соглашениям, которые приняты при проектировании для этой структуры данных. В момент возникновения дефекта это может никак не проявиться, но зато привести к появлению наведенных ошибок при исполнении формально правильного кода.

Зацикливание внешне проявляется как зависание программы. Зацикливание рекурсивного алгоритма приводит к фатальной ошибке, связанной с переполнением стека.

Фатальная ошибка – неуправляемое завершение программы.

Фатальная ошибка – перехваченное исключение, приводящее к аварийному завершению отдельного этапа исполнения программы с последующим восстановлением либо к завершению программы с сообщением об ошибке, либо к ее перезагрузке.

Деградация по производительности или ресурсам. Программа снижает свою производительность либо использует значительно большее, чем обычно, количество ресурсов, например памяти. Деградация может быть обусловлена утечками памяти, «зависшими» неосвобожденными ресурсами, чувствительностью алгоритма к определенным наборам данных. Деградация может быть моментальной и накапливаемой. В последнем случае производительность постепенно снижается с течением времени.

С точки зрения тестирования зацикливание и фатальная ошибка являются даже более предпочтительными, чем продолжение работы программы, так как позволяют сразу идентифицировать ошибку, а неправильный результат или деградация не всегда могут быть замечены сразу.

Виды ошибок по отношению к структуре алгоритма

Опечатка – ошибка, внесенная бессознательно или автоматически в одно выражение или оператор, исправляется редактированием. Как правило, такая ошибка обнаруживается при отладке программы либо при структурном тестировании при исполнении фрагмента кода с ошибкой.

Крайняя ситуация (граничное условие). Ошибка, связанная с пропуском одного из *граничных условий* – сочетаний данных, с которыми работает программа, например пустая строка, строка, состоящая из одних пробелов и не содержащая слов. Также возможна ошибка, связанная с пропуском одного шага цикла или рекурсии, обычно первого или последнего. Основной причиной появления таких ошибок является то, что программа разрабатывается на основе анализа поведения образной модели, отражающей частный случай работы программы. Ошибка обнаруживается инспекцией кода или функциональным тестированием при достаточном тестовом покрытии. Исправляется путем вставки фрагмента, работающего с этим условием.

Методологическая ошибка алгоритма. Алгоритм, положенный в основу программы, не во всех случаях дает решение либо в определенных случаях дает неверное решение. Но в отличие от ошибок граничных условий эта частичная неработоспособность не может быть локализована и устранена. Иногда выбранный алгоритм не подходит по производительности и трудоемкости – не масштабируется по данным. Ошибка устраняется разработкой нового алгоритма решения задачи.

Методологическая ошибка архитектуры. Выбранное архитектурное решение не обеспечивает выполнение требований, предъявляемых к системе, т. е. является ошибкой проектирования.

Затраты на исправление

В зависимости от вида ошибок и структуры кода могут потребоваться различные действия:

- исправление выражения / оператора;
- исправление фрагмента кода в отдельном методе;
- рефакторинг – изменение заголовка метода, интерфейса, заголовка класса, прав доступа, сопровождающееся формальным изменением кода в разных частях проекта;
- реинжиниринг – изменение структуры значительной части программного кода (группы классов, характера их взаимодействия);
- изменения программной архитектуры проекта.

Причины появления ошибок

> Я понял. Оказывается, это неправильные программисты. И они, наверное, делают неправильный код.
>
> *Винни-Пух*

Причина большинства ошибок – человеческий фактор. Это не значит, что во всем нужно винить автора кода, но соблюдение некоторых принципов позволяет сократить количество ошибок в программном коде хотя бы на уровне модулей или их взаимодействия.

Незнание или отсутствие / недоступность информации. Причиа ошибки – неадекватное понимание механизма работы сервиса или службы, незнание семантики языка или особенностей среды исполнения.

Стиль редактирования или структурирования кода, способствующий появлению ошибок. При редактировании кода легко допустить ошибку, когда при формальной замене одной конструкции на другую меняется синтаксическая структура программы. Такие ошибки формально относятся к *опечаткам*, но от этого процедура их устранения не становится проще.

Типичным примером здесь является *вынесение за тело цикла* отдельных операторов или, наоборот, *попадание в тело цикла* операторов, не являющихся его частью. Чтобы этого избежать, необходимо соблюдать правила структурного редактирования кода. Для Си-подобного синтаксиса это:

- форматирование текста кода с равным отступом для всех строк одного уровня вложенности;
- заключение единственного оператора в парные скобки;

- использование правила парных скобок. При вводе синтаксической конструкции, содержащей компоненты в скобках, необходимо сначала ввести заголовок конструкции с пустыми парными скобками, например *if(){}* или *a.addActionListener(new ActionListener(){}),* а затем вписывать в них содержимое.

Технологический стиль разработки. Все правила хорошего тона технологии ООП (сокрытие данных, независимость интерфейса от реализации, абстрагирование, использование шаблонов проектирования) являются основой для потенциального снижения ошибок межмодульного взаимодействия.

Несоблюдение соглашений по данным или инвариантов. Выше были даны определения инвариантов программного кода и соглашений по данным. Отсутствие их четкого определения или размытость на этапе проектирования может привести к ошибкам граничных ситуаций:

- нарушение соглашений по данным – код не проверяет одно из возможных значений, например, допустимое по принятым соглашениям значение null-ссылки, что приводит к исключению;
- нарушение инвариантов кода – одна из ветвей фрагмента кода не формирует правильные начальные значения для следующей исполняемой по времени ветви. Например, одна из ветвей тела цикла не формирует правильные начальные значения следующего шага.

Классификация дефектов кода

В данном случае нас интересует не столько проявление ошибки – сбой, сколько его первопричина – дефект. Поэтому будем классифицировать не сами ошибки и их проявления, а вызывающие их дефекты.

Дефекты, связанные с особенностями языка, компилятора или системы исполнения кода

Сразу же оставим в стороне ошибки компиляции, поскольку они связаны со *статической* структурой программного кода. Они по большому счету не являются ошибками, поскольку относятся к формальной структуре программы. Тем не менее компилятор может просматривать *потенциально возможные варианты* исполнения ветвей программы и контролировать для всех них обязательное исполнение действий, устраняющих потенциальные ошибки. Например, в Java к этому относятся:

- синтаксический контроль источников синхронных исключений и передача полномочий на их обработку вызывающему модулю (спецификатор *throws*);

• синтаксический контроль проверки инициализации локальных переменных при исполнении всех возможных ветвей и контроль обработки всех исключений в методе.

Ошибки, возникающие во время исполнения кода (*run-time*), могут поразному обнаруживаться и обрабатываться системой исполнения кода. Это зависит от свободы и ответственности, которые предоставляются программе в среде исполнения, а также от принятой в *языке модели исполняемой программы* и ее особенностей. Например, в большинстве языков программирования переменные и массивы интерпретируются как области памяти, выход за пределы которых недопустим. Вследствие этого выход индекса за пределы массива считается ошибкой времени исполнения. В Си / Си++ благодаря адресной арифметике любая переменная рассматривается как часть неограниченной области памяти, доступной программе. Поэтому сознательный, случайный или злонамеренный выход за ее переделы не является ошибкой.

Еще один пример необнаруживаемой *run-time* ошибки в Си – возвращение указателя на локальную переменную, которая уничтожается в стеке при выходе из функции. При этом впоследствии через указатель можно повредить произвольное содержимое стека.

```
int  *F() {
        int a=5;
        return &a;        // Возвращается указатель на локальную
                             переменную в стеке.
}
```

Дефекты адресной арифметики в Си++. Адресная арифметика с Си / Си++ – инструмент работы с памятью на низком уровне. Любой указатель позволяет интерпретировать его содержимое как адрес переменной или как адрес неограниченного массива с произвольным значением индекса. Кроме того, возможно преобразование типа данных указуемых элементов. Все это позволяет работать с памятью на низком уровне – уровне физических адресов и вычисляемых размерностей со всеми вытекающими отсюда опасностями.

Дефекты вычислений и преобразований

Операции с вещественными числами. Наиболее распространенная ошибка при работе с вещественными числами – игнорирование того факта, что вещественное число является *принципиально неточным*, т. е. имеет фиксированное количество точных разрядов и соответствующую погрешность представления. При выполнении операций в цикле эта погрешность будет накапливаться,

в результате чего сравнения на равенство с известной суммой не произойдет (рис. 8.8). В следующем примере сумма из 33 значений 1/33 оказывается равной 0,9999999999999993 и цикл проскакивает через значение, равное единице. В то же время произведение (1/33.)*33 оказывается в точности равным единице.

```
//-----------------------------------------83__BugExamples
double dd=1/33.,w=0;
System.out.println("dd="+dd);      // dd=0.030303030303030304
//--------- Сумма пролетает через 1
for (w=0;w!=1;w+=dd){
    if (w>1) break;
    }
System.out.println(w);             // 1.0303030303030296
int i=0;
//---------- 33 раза по 1/33 меньше 1
for (w=0,i=0;i<33;w+=dd,i++); // 0.9999999999999993
System.out.println(w);
System.out.println(dd*33.);        // 1.0
```

Рис. 8.8. Дефект вычислений с вещественными числами

Дефекты диапазонов представления данных. Иногда диапазон представления данных, соответствующий используемому типу, может оказаться недостаточным. Например, метод *DataOutputStream.writeUTF* двоичного потока данных передает строку в формате «счетчик длины – байты кода UTF8». Счетчик длины передается переменной типа *short*, что ограничивает длину последовательности значением 65535 (а с учетом кодировки со знаком –32867). Хотя на сам параметр метода – строку таких ограничений нет. Вызов метода для длинной строки сопровождается исключением. Это можно рассматривать как ограничение метода, однако в самой программе, использующей этот метод, это уже будет дефектом, если необходимая проверка не будет выполнена программой.

Дефекты явного и неявного преобразования типов данных. Явные и неявные преобразования примитивных типов в выражениях могут приводить к следующим ошибкам:

- потеря значащих разрядов при уменьшении размерности целых;
- заполнение знаковым разрядом при увеличении размерности отрицательных чисел – *арифметическое расширение*. В арифметических задачах это вполне естественно, так как сохраняет значение переменной со знаком,

а в задачах поразрядной обработки данных приводит в неправильному заполнению разрядов слева.

```
byte a=(byte)0xFF;  // Значение со знаком =-1
int b=a;                        // Будет FFFFFFFF
```

Про неявные преобразования программист иногда просто забывает, поскольку они выполняются довольно часто. Например, все короткие типы перед вычислением выражений в Java и Си автоматически удлиняются до стандартной длины.

Дефекты структурирования и разработки кода

Дефекты этого вида связаны с процессом проектирования и кодирования программы и в большинстве своем выглядят как опечатки, т. е. исправляются редактированием одного выражения, условия или оператора. Тем не менее обнаружение и локализация таких ошибок представляет собой сложную задачу.

Дефекты инициализации. Инициализация переменных, используемых в программе, отсутствует либо производится не тем значением, либо не в том месте, поэтому инициализация осуществляется не всегда, когда требуется (рис. 8.9).

```
int i=0,j=0;                    // Неправильное место инициализации j
for(;i<n;i++){
    for(;j<n;j++){...}          // j=0 должно быть здесь for(j=0;...)
}
int max;                        // Неправильное значение инициализации
for(max=0,i=0;i<n;i++)          // Надо max=A[0], ошибка при всех отрицательных в массиве
    if ()
```

Рис. 8.9. Дефекты инициализации

В настоящее время многие компиляторы позволяют отслеживать наличие инициализации во всех ветвях программы, а отладчики или среда исполнения – использовать определенные значения переменных для установки их как неинициализированных.

Лишний или недостающий шаг цикла. Дефект можно назвать «*плюс-минус метр от столба*». Он обнаруживается на граничных условиях программы и связан с тем, что программист завершает цикл на один шаг раньше или позже.

Дефект смещения на один шаг цикла. Аналогичная ситуация, когда требуемое действие выполняется с ошибкой на один шаг цикла или на один элемент последовательности. Иногда это зависит от различий в соглашениях по данным. Например, в стеке указатель стека ссылается на последний записанный элемент – вершину стека, а количество элементов в массиве в Си является индексом первого свободного. В результате операции добавления в стек и в последовательность выглядят по-разному: A[++sp]=v и A[n++]=v.

Дефект начального или конечного шага цикла. Наиболее распространенным дефектом граничных условий является дефект первого и последнего шагов цикла. Например, программы, использующие пары соседних элементов массива, имеют такой потенциальный дефект, поскольку первый и последний элементы не имеют соответствующей пары слева и справа.

В тексте программы должны быть условия или условные операторы, срабатывающие на первом и последнем шагах и корректирующие поведение программы.

```
//-------------------------- Подсчет количества слов в строке
int ns=0;
]for(i=0; s[i]!=0' i++){
     if (s[i]!=' ' && (i==0 || s[i-1]==' ')) ns++;          // Обнаружение начала слова ИЛИ
     if (s[i]!=' ' && (s[i+1]==0 || s[i-1]==' ')) ns++;     // Обнаружение конца слова
}
```

Рис. 8.10. Дефекты начального / конечного шага цикла

Дефекты размерностей данных

Дефекты, локализованные в коде, который связан с перераспределением памяти и других ресурсов, могут проявлять себя только при превышении определенной размерности входных данных.

Дефекты форматов входных данных

Программа, использующая входные данные, получает их в определенном формате. Однако по различным причинам этот формат может быть нарушен. Причины могут быть разными, например:

- ошибка в программе, сгенерировавшей входной файл;
- содержимое файла изменено случайно или преднамеренно;
- предъявлен файл от устаревшей версии программы;
- используется база данных с устаревшей структурой таблиц (из предыдущих версий);

- программа открывает файл с другим содержимым или файл другого типа;
- сетевое соединение закрывается до того как будет передан весь файл.

В результате в программе возникают *наведенные ошибки,* которые могут приводить к зацикливанию, фатальному завершению и созданию некорректных структур данных. От ошибок этого типа программа может и должна защищаться. Способы защиты:

- внесение избыточности в структуры данных – контрольное суммирование данных, счетчики размерности, уникальные сигнатуры;
- перехват исключений от наведенных ошибок и их анализ с целью определения причин.

Многие дефекты этого вида имеют отношение к технологическому процессу *управления конфигурациями.* При проектировании программного продукта необходимо во все виды входных данных закладывать информацию о текущей версии формата: в заголовке файла, в таблице конфигурационных параметров БД. Такие дефекты необходимо рассматривать во взаимосвязи с вопросами устойчивости и защищенности программы.

Наведенные ошибки могут появляться и при передаче данных по сетевым соединениям, например при наличии дефекта в программе – передатчике сообщения нарушается его формат и программа-приемник может среагировать на это, начиная с обнаружения ошибки формата и заканчивая зависанием на приеме или исключением.

Дефекты форматов внутренних данных и соглашений по данным

Соглашения по данным – допустимые конфигурации элементов и их значений, которые обязаны соблюдаться всеми компонентами программного кода. Несоблюдение их в одной части кода может привести к появлению недопустимых сочетаний данных, которые при исполнении другой, правильной, части кода приведут к сбою. Эта ошибка может быть мерцающей.

Дефекты общего доступа и разделения данных

Возможность изменения состояния объекта из нескольких участков кода может приводить к ошибкам, связанным с корректностью их разделения. К этому имеют отношение определенные выше понятия *контекста* (см. раздел 3.2) – множества имен прямодоступных программных объектов в коде и *связности* / *сцепления* (см. раздел 3.4) – метрических характеристик кода.

Компактность и удобство работы с кодом напрямую зависят от объема контекста и его разнообразия: чем больше контекст, тем большее количество компонент находится «под рукой». Однако это увеличивает потенциальную

связность / сцепление по данным и повышает верятность возникновения ошибок, связанных с их разделением.

Дилемма «ссылка–значение». Базовыми понятиями модульной организации программы являются способы передачи данных между модулями *по ссылке* и *по значению.* Языки по-разному поддерживают этот механизм. В Си++ передачи объекта по ссылке и по значению синтаксически равноправны, тогда как в Java для объектов основной является передача по ссылке, а для передачи по значению используется явное клонирование объектов. Каждый способ имеет свои особенности возникновения возможных ошибок, связанных с управлением данными. Для ссылок – это *ошибки разделения*, для значения – *проблемы синхронизации и устаревания данных*.

Побочные эффекты и дефекты доступа по ссылке. Доступность ссылки на объект дает потенциальную возможность изменения его содержимого из компоненты кода, где она присутствует. Можно выделить несколько уровней программного контроля разделения данных.

Уровень 1. Закрытость данных. Безусловное закрытие прямого доступа к данным вне класса. Для этого данные объявляются как *private,* и доступ к каждому из них реализуется через пару методов *get* / *set.* Это позволяет:

- при изменении внутреннего представления данных в классе избежать наведенных ошибок компиляции в других частях кода, работающих с ними по прямой ссылке;
- быстрее локализовать ошибку, связанную с доступом к этим данным;
- легко добавить средства ограничения доступа и сбора статистики использования.

Уровень 2. Средства раздачи и контроля доступа к объектам. Объекты создаются и сохраняются в отдельном классе, сам класс раздает ссылки на эти объекты и подсчитывает их количество (существуют методы получения и возвращения ссылки). В идеальном случае класс сохраняет ссылки на запросившие объекты и делает обратные вызовы при изменении данных. Подобное решение реализуется в шаблоне проектирования «наблюдатель».

Уровень 3. Сервисы обработки данных и обмен сообщениями между компонентами. Компоненты кода не имеют прямых ссылок на данные, а изменяют их при помощи обращения к сервисам, управляющим данными, либо передают друг другу сообщения, в соответствии с которыми производят манипуляции с данными.

Ошибки сборки, конфигурирования, размещения, настройки

При наличии версий программной системы или ее сборке в различных конфигурациях возможны ошибки, связанные с включением устаревшей

версии кода либо кода, не соответствующего конфигурации. Такая же ситуация возможна при размещении программных компонент в распределенной системе при настройке их параметров.

Дефекты использования ресурсов

Внешние утечки памяти. В Си потеря ссылки на динамические данные приводит к утечке памяти, поскольку процесс ее освобождения контролируется только программой. В Java/C# это исключено, так как такая ситуация контролируется сборщиком мусора в системе исполнения кода.

Внутренние утечки памяти. Неиспользуемые объекты, на которые сохранены ссылки в структурах данных, являются внутренним мусором, который не утилизируется и может сохраняться неограниченное время.

Зависшие ресурсы. Ресурс, который зарезервирован программой, но не освобожден, например открытый файл или сетевое соединение, остается за программой в течение некоторого времени: до ее завершения либо до операции сбора мусора над соответствующим объектом.

Ошибки, связанные с ограничениями по ресурсам. Любой ресурс, запрашиваемый программой, имеет физические ограничения, поэтому возможны сбои, связанные с отказом в его предоставлении. Большой объем используемого ресурса может также приводить к деградации производительности, например, за счет интенсивоного страничного обмена при работе с виртуальной памятью, превышающей физически доступный объем.

Дефекты инициализации и восстановления программы

Дефекты инициализации уже обсуждались, но по отношению к фрагменту кода, например цикла. Аналогичные дефекты имеют место в процессах инициализации всей программы или ее восстановления. Как правило, они связаны с некорректным использованием статических или внешних данных конфигурации.

Эффект повторного запуска. При первом вызове модуль не срабатывает, но инициализирует структуры данных для корректного срабатывания последующих вызовов. Например, при отсутствии конфигурационного файла срабатывает исключение, при обработке которого в конфигурационном файле сохраняются параметры текущего состояния. Тогда при повторном запуске сбоя не происходит.

Эффект единичного вызова. Вызов метода или функции искажает содержимое объекта или структуру данных. При повторном вызове этого или других методов возникает сбой.

Сюда же относятся ошибки, связанные с сохранением необходимых данных и восстановлением программы после сбоев, при восстановлении сетевых соединений, внешних сбросов со стороны ОС и т. п.

Так бывает. В ОС Android при закрытии приложения оно не выгружается без необходимости из памяти и при повторном открытии просто перезапускается. Если при этом не инициализируются статические данные, то их значения остаются от предыдущего запуска.

Ошибки и дефекты реактивности и производительности

Ошибки приводят к неприемлемым задержкам при работе программы, например торможению или подвисанию программы при обработке больших массивов данных либо медленной реакции на интерактивные действия пользователей.

Зависание графического интерфейса (GUI). При синхронном выполнении продолжительных операций происходит блокирование интерфейса пользователя. Дефект устраняется исполнением продолжительного действия в отдельном фоновом потоке. При этом возникают следующие проблемы:

- разделение общих данных: проще всего организационно разграничить доступные данные для основного и фонового потоков, например временно ограничив функционал GUI;
- при любом варианте завершения фонового потока, когда требуется взаимодействие с элементами графического интерфейса, необходима синхронизация с потоком GUI (см. раздел 3.2).

Влияние фактора трудоемкости. При увеличении размерности обрабатываемых данных степень замедления программы зависит от трудоемкости алгоритма, которая, в свою очередь, оценивается как функция определенного вида – линейная, квадратичная и т. п. От этого зависит масштабируемость программы по входным данным.

Чувствительность алгоритма к данным. При определенных сочетаниях данных производительность программы может деградировать. Это определяется как чувствительность алгоритма к данным. Иногда это дает положительный эффект, но чаще – отрицательный.

Несоответствие размерностей данных. Программа должна адаптироваться под размерности данных, которые она обрабатывает. Например, перераспределение динамического массива следует производить в геометрической прогрессии, а размеры буферов увеличивать пропорционально размеру файла. Иначе возникает эффект «вычерпывания бочки чайной ложкой».

Технологические ошибки – лишние циклы и повторные вычисления. Каждый лишний цикл дает еще одну дополнительную степень трудоемкости. Если этот цикл касается основной размерности входных данных, то это качественно снижает характеристики производительности. Типичный пример на Си:

```
char s[…]; for(int i=0; i<strlen(s);i++)…s[]…
```

Функция определения длины строки *strlen(char*)* использует аналогичный цикл просмотра строки до символа-ограничителя, что при повторном ее вызове в заголоке цикла дает квадратичную трудоемкость вместо исходной линейной.

Повторные вычисления – повторение некоторой части программы при тех же исходных данных также может приводить к снижению производительности в размерах, определяемых частотой этих повторений. Основным методическим приемом для исключения этого эффекта является *динамическое программирование* – создание кэша результатов решенных подзадач. В этом случае при обнаружении в кэше решения с требуемыми входными параметрами его результат повторно используется на нужном шаге (см. [9]).

Методологические ошибки алгоритмов. Многие задачи могут быть решены как при помощи очевидного примитивного алгоритма, так и при помощи «умного». Если размерность задачи позволяет решить ее в приемлемое время, то примитивный алгоритм предпочтительней из-за простоты отладки и меньшего числа потенциальных ошибок. Однако решение задачи «в лоб» с определенного момента не может обеспечить необходимой производительности. Более сложный алгоритм зачастую нуждается в обосновании корректности своей работы (см. раздел 3.1 – жадные алгоритмы). Методологические ошибки как раз и возникают из-за того, что в основу неочевидного алгоритма закладываются неверные логические принципы.

Незнание или отсутствие информации о внутренних процессах в среде исполнения. Программист, используя готовые программные решения, не всегда имеет информацию о механизмах их реализации либо не считает нужным ее изучить. Например, особенности реализации классов *String* и *StringBuffer* в Java могут проявиться в виде различий в производительности при работе с длинными строками.

Особенности среды исполнения, программного и аппаратного окружения. Вопросы производительности могут завести далеко вглубь программно-аппаратной среды исполнения. Например, частота обновления кэш-памяти зависит от вида распределения обращений по пространству адресов – линейного, случайного, локализованного. Любая программа, работающая в ОС, имеет два уровня кэширования – это подкачка страниц виртуальной памяти

в физическую и процессорный кэш оперативной памяти. Поэтому разные способы работы с линейными массивами большой размерности, например работа по строкам или столбцам, могут давать различную производительность, обусловленную эффектами кэш-памяти.

Дефекты параллелизма и синхронизации

Сложность обнаружения дефектов параллелизма состоит в том, что они приводят к *мерцающим* и *наведенным* ошибкам. Некорректная синхронизация данных может привести к нарушению их целостности в связи с последовательностью их изменений, невозможной при обычном, непараллельном исполнении методов. Некорректная синхронизация потоков может вызывать зависание или блокирование некоторых из них.

Дефекты распределенных систем и протоколов

Дефекты в распределенных системах и поддерживающих их протоколах связаны с тем, что в них присутствуют физический параллелизм и разделение данных.

ГЛАВА 9
МЕТОДИЧЕСКИЕ МАТЕРИАЛЫ ПО ДИСЦИПЛИНАМ

9.1. Инжиниринг ПО. Индивидуальные задания

Дисциплина «Инжиниринг ПО» ориентирована на дальнейшее развитие навыков программирования и проектирования ПО с использованием шаблонов проектирования и стандартных технологических средств языка программирования и среды разработки, на получение навыков написания качественного управляемого кода в проектах небольшого и среднего размера.

Индивидуальное задание выполняется в виде разработки приложения, использующего один из шаблонов проектирования. Пояснительная записка должна содержать:

- выдержки из учебно-методического материала по шаблону (учебники, учебные пособия, лекционный материал, тематические сайты);
- материал по практике применения шаблона (форумы, тематические сайты);
- описание разработки;
- результаты тестирования.

Все сторонние материалы должны быть снабжены корректно оформленными библиографическими ссылками.

Варианты заданий

1. *Шаблон MVC*. **Сетевая (локальная) игра типа «Морской бой», «Домино».** Модель хранит структуру данных игры (расстановку кораблей), и над ней выполняются методы по управлению игрой. Контроллер определяет порядок ходов, проверяет возможность выполнения хода. Внешнее представление (*view*) связано с контроллером двунаправленным интерфейсом: контроллер управляет отображением элементов игрового поля, выводит текстовые

сообщения, получает от представления события – клик по элементу поля, начало новой игры, завершение и т. п. Варианты игры:

○ локальная – одна модель, один контроллер, два представления для игроков;

○ сетевая – одна модель, один контроллер, два представления для игроков. Представления связываются с приложением, содержащим контроллер и модель. Контроллер обеспечивает множественные соединения и синхронизацию;

○ сетевая – две модели, два контроллера, два представления. Контроллеры поддерживают соединение и передают команды: сделан ход, синхронизация моделей, сброс и начальная установка игры.

2. *Шаблон MVC (классический)*. **Связанные экранные формы**. Группа экранных форм (представлений) имеет ряд текстовых полей, отображающих параметры модели. На каждый параметр представление подписывается к модели на событие, связанное с его изменением. Интерфейс подписки идентичен для всех параметров. Для каждого параметра создается оригинальный контроллер, обрабатывающий команду изменения значения параметра со стороны представления. Модель описывается набором зависимостей, параметры модели могут быть *входными*, *выходными* (*результатами*) и *выходными с возможностью задания начальных значений* (*инициализацией*).

3. *Шаблон «прототип».* **Класс таблицы с произвольной структурой столбцов**. Хранимые данные разных типов – целые, вещественные, дата, время, GPS-координаты – создаются на основе абстракции данных с функционалом: имя класса, парсинг из строки и вывод в строку, клонирование, сравнение и сложение с объектом того же типа. Строка таблицы определяется набором объектов-прототипов для столбцов. Сама строка также клонируется. Таблица состоит из вектора строк-имен, строки-прототипа и строк самой таблицы. Функционал: создание таблицы, добавление столбцов, добавление строк, сортировка по столбцу с заданным номером, сложение строк, сохранение и загрузка из файла. Программный интерфейс и оконное приложение. Тестирование на больших данных – импорт из Excel, генерация тестовых таблиц.

4. *Шаблон «приспособленец»*. **Дерево версий текстового файла**. Файл редактируется по словам. Класс словаря содержит хэш-таблицу адаптеров со ссылками на оригинальные слова в виде «ключ – само слово». Адаптер содержит количество ссылок на слово из всех версий текста. Каждая версия текста – вектор ссылок на адаптеры. При добавлении или вставке слова в версию текста оно ищется в фабрике, при нахождении счетчик в адаптере увеличивается на единицу, при отсутствии добавляется в словарь с новым адаптером. При удалении счетчик ссылок уменьшается. Изменение слова рассматривается как

последовательность операций удаления и вставки. Вся структура данных сериализуется в файл. Протестировать шаблон на сказке «Репка».

5. *Шаблон «композиция для древовидной системы».* **Простой графический редактор с ограниченным набором действий**: группировкой / разгруппировкой элементов, изменением размеров, переносом на передний / задний план, перемещением объектов, сохранением картинки в файл. Абстрактный класс элемента, группы элементов и конкретных графических объектов – окружность, полигон, строка текста. Ограничивающий прямоугольник, селекция объектов по точке и прямоугольнику.

6. *Шаблон Command.* **Простой графический редактор с набором команд редактирования графических объектов**: создать объект, переместить, изменить размер, удалить, переместить на передний и задний план. Группа команд обработки объекта с общим интерфейсом Do/ReDo/UnDo. Производный класс запоминает параметры, необходимые для выполнения прямой и обратной команд. Класс-менеджер команд поддерживает очередь команд ограниченной длины, текущий обрабатываемый объект, методы *Do/ReDo/UnDo*, выбирая их из очереди и вызывая соответствующие методы в объектах-командах.

7. *Шаблон «пул потоков».* **Моделирование последовательного и параллельного исполнения запросов**. Запрос представляет собой класс с присоединенным Runnable, в котором содержится исполняемый код. Поток, содержащийся в пуле, запускается при создании. Код потока содержит цикл, в котором он засыпает, после пробуждения берет назначенный ему запрос, исполняет содержащийся в нем код, уведомляет менеджер о своем завершении и засыпает. Менеджер потоков содержит вектор потоков-исполнителей. При обращении к менеджеру с запросом последний выбирает поток из пула, передает ему запрос и пробуждает его. Если свободного потока нет, то запрос ставится в очередь. Провести сравнительное тестирование производительности при последовательном исполнении запросов, параллельном исполнении каждого в отдельном потоке и с использованием пула потоков.

8. *Шаблон «прокси».* **Фильтр слов в текстовом потоке**. Класс с интерфейсом текстового потока при конструировании делегируется к однотипному объекту-источнику и отфильтровывает набор слов, передаваемый при конструировании. Используется внутренняя очередь символов для отложенного распознавания. Протестировать на цепочке фильтров для разных наборов слов.

9. *Шаблон «сессия».* **Модель банкомата и платежной системы**. Клиент и сервер используют синхронный обмен «запрос–ответ». При первоначальном установлении соединения клиент получает уникальный идентификатор, сервер создает дескриптор соединения. Клиент нумерует передаваемые сообщения,

сервер сохраняет номер и ответ на последнее переданное сообщение. Сервер после каждого изменения сохраняет дескрипторы в файл. Клиент также запоминает в файле идентификатор сессии, номер и последнее переданное сообщение. Обеспечить восстановления соединения и сохранность последовательности сообщений (отсутствие пропадания и дублирования) при перезагрузке клиента и сервера.

10. *Шаблон параллелизма*. **Модель параллельных запросов к серверу от группы источников**. Множество потоков может посылать независимые запросы к серверу через единственное соединение. Используется последовательная нумерация передаваемых сообщений для идентификации сообщений и ответов, передаваемые сообщения запоминаются у клиента, потоки, передавшие их, засыпают. Имеется поток передачи у клиента, поток приема у сервера, отдельный поток исполнения на сервере для каждого сообщения, поток передачи ответов на сервере, поток приема ответов у клиента. Используется буферизация передаваемых сообщений у клиента и на сервере. Смоделировать группу потоков, посылающих случайные слова, которые сервер переворачивает со случайной задержкой, определить процент загрузки соединения.

11. *Шаблон «кэш объектов»*. **Кэширование промежуточных решений**. Разработать класс – кэш объектов с возможностью изменения размера кэша, сбора статистики и применения различных стратегий вытеснения (FIFO, LRU, RAND). В качестве кэша использовать хеш-таблицу с ключом соответствующего типа. Использовать при решении одной из задач динамического программирования – поиска оптимального решения на основе комбинаторного перебора с кэшированием промежуточных решений (см. [9], п. 7.7. Эффективность алгоритмов).

12. *Шаблон «итератор»*. **«Умные» итераторы**. Структура данных порождает итераторы и запоминает ссылки на них. При изменении или удалении одним из итераторов элемента структуры данных остальным итераторам, которые ссылаются на этот же элемент, передается событие (шаблон Observer), либо при следующем обращении к нему генерируется исключение. Операции над итератором: установка на первый, последний, следующий, предыдущий, по логическому номеру, извлечение через итератор, вставка на позицию итератора, обновление, удаление.

В вариантах 13–15 требуется разработать классы, использующие внутренние потоки для промежуточной буферизации данных. Для моделирования прикладных процессов используются потоки, которые засыпают на случайный момент времени, после чего читают / записывают очередную порцию данных постоянного или случайного размера. Необходимо предусмотреть сбор статистики в классах буферизации – средний объем данных в буфере.

13. *Класс буферизованного ввода в реальном времени*. При конструировании получает параметр – физический поток данных с интерфейсом *InputStream*. Использует внутренний циклический буфер или односвязный список блоков, содержащих массив байтов фиксированной размерности (буферный пул). Создает поток, который читает байты из входного потока и записывает в циклический буфер. При заполнении циклического буфера засыпает. Метод чтения извлекает из циклического буфера очередной байт, возвращает –1 при окончании данных в потоке-источнике, блокируется при отсутствии данных в циклическом буфере.

14. *Класс PipedStream с циклическим буфером данных*. Класс *PipedOutputStream* имеет циклический буфер, в который пишет поток байтов, блокируя текущий поток при заполнении буфера. Класс *PipeInputStream* получает ссылку на *PipedOutputStream* и при чтении данных либо блокируется при их отсутствии, либо извлекает данные, деблокируя поток записи.

15. *Класс отложенной записи*. При открытии файла классом с присоединенным интерфейсом *OutputStream* создается ByteOutputStream, в который пишутся данные потока. При закрытии объект, содержащий имя файла и байтный массив, ставится в очередь, из которой фоновый поток извлекает объекты и пишет их содержимое в файлы. Сравнить среднее время записи в обычный и отложенный файл.

9.2. Управление программными проектами. Коллективный проект

Цель практикума – коллективная разработка простого проекта, создание его реально действующего прототипа. Основное требование к тематике: проект должен разбиваться на достаточно автономные модули примерно по числу бригад, иначе при нулевом опыте коллективной разработки он завязнет в согласованиях. Желательно ограничить число вариантов взаимодействия таких частей, например по линейной схеме. Части проекта выполняются бригадами по два человека (парное программирование), эти части являются функциональными единицами разработки. Ведется метрика проекта: содержание работ, трудозатраты, результат. Проект размещается в системе контроля версий.

Этапы выполнения практикума:

- неформальное изложение преподавателем сути проекта (видение): предметная область, глоссарий, функционал, границы. Выборы лидера команды, деление на бригады, распределение функциональных модулей (рис. 9.1);

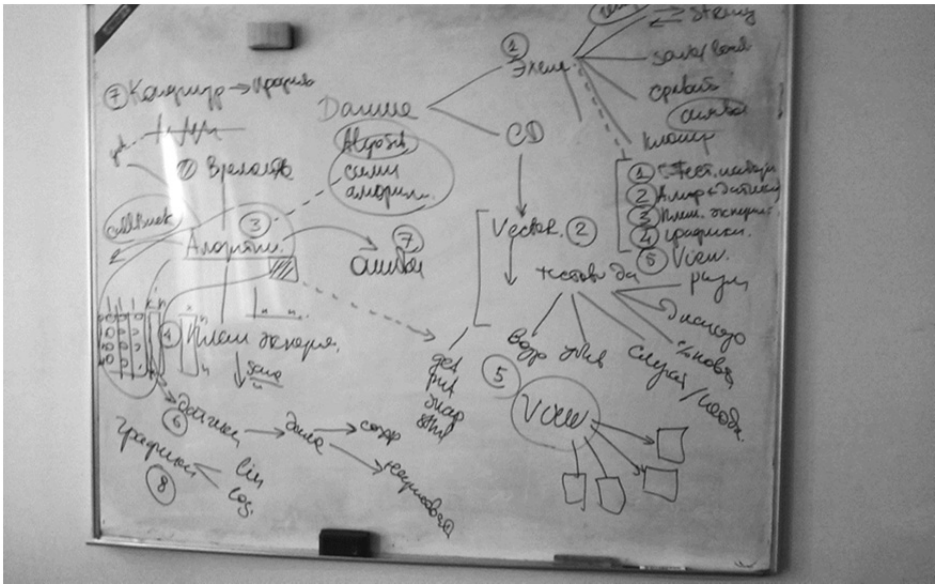


Рис. 9.1. Неформальное обсуждение проекта

- подготовка преподавателем и лидером каркаса проекта: абстракции, интерфейсы и заготовки классов (ключевых и используемых совместно несколькими бригадами). Обзор каркаса вместе со всеми участниками проекта. Регистрация в системе контроля версий;
- общая постановка задачи преподавателем и лидером каждой бригаде, обсуждение нюансов, оценка понимания поставленных задач и способов их реализации в коде;
- автономная разработка и тестирование модулей бригадами, создание тестовых объектных заглушек, замена их на объекты классов, разработанных другими бригадами, при необходимости коррекция и согласование интерфейсов в каркасе, ведение общей ветки в системе контроля версий;
- последовательное сведение частей проекта, интеграционное тестирование, сведение проекта в прототип. Разработка приемочных тестов. Демонстрация и обсуждение проекта. Оформление документации. Обзор метрики проекта.

Анализ проблем, тормозящих разработку и развитие соответствующих навыков:

- написание кода под абстракции;
- принятие решений по развитию абстракций – добавление методов в интерфейсы;

- движение по линии наименьшего сопротивления – разработка понятных частей;
- создание тестовых конфигураций;
- отсутствие инициативы – формулировка требований к коду других участников разработки, создание тестовых объектов для других разработчиков, совместная отладка и тестирование частей проекта.

Пример проекта

Видение

Система тестирования и анализа производительности алгоритмов. Desktop-приложение позволяет создавать тестовые наборы и тестировать по ним алгоритмы на предмет производительности и трудоемкости, отсутствия ошибок и соответствия тестовому результату. Содержание работы: получение функции трудоемкости для различных операций и «грязного» времени работы модуля, реализующего алгоритм, в зависимости от размерности входных данных при прочих варьируемых параметрах; вывод и сравнительный анализ результатов в графической форме, подбор О-нотации трудоемкости и оценка точности приближения; генерация и сохранение тестовых наборов и результатов выполнения в файлах. Сохранение последовательности действий в эксперименте в XML-файле. Исполняемый алгоритм оформляется в виде метода в унаследованном базовом классе, в котором он может использовать тестовые наборы данных и датчики событий. Набор датчиков генерируется самим классом, т. е. «зашит» в классе.

Распределение команд по частям проекта:

1) классы тестируемых данных, генераторы данных для части 2;
2) тестовый набор, тестовый план, генерация планов;
3) алгоритм, датчики, эксперимент, результат;
4) визуализация, графики;
5) графический интерфейс, профиль пользователя;
6) оценка вида зависимости, генератор функций;
7) командный язык для описания экспериментов.

Каркас проекта

Каркас проекта содержит интерфейсы и абстрактные классы для значимых компонент (сущностей), а также заготовленные технологические решения: шесть пакетов, десять интерфейсов, 16 классов (рис. 9.2):

- *I_File* – сериализуемый двоичный поток собственного формата;
- *I_Name, I_TypeName, I_ObjectName* – интерфейсы именованных классов (типов) и объектов. Классы с именованными типами и объектами необходимы для визуализации внешнего имени сущности или объекта, сервис используется классами-фабриками;
- *UNIException* – класс унифицированных исключений;
- *TypeFactory* – шаблон-фабрика объектов для имеющегося многообразия с интерфейсом *I_NameFactory.* Для указанного класса производит сканирование пакета средствами рефлексии, ищет классы-наследники и создает для них объекты-прототипы. Позволяет получить объект-клон по имени его класса;
- *BoxFactory* – шаблон, создает фабрику для класса – параметра шаблона и связывает ее с выпадающим списком в экранной форме;
- *I_DataElement* – интерфейс элемента тестируемых данных. Функционал: преобразование во внешнюю форму и из внешней формы, именуемый тип, клонирование, сравнение, сложение, вычитание, методы для генерации последовательностей;

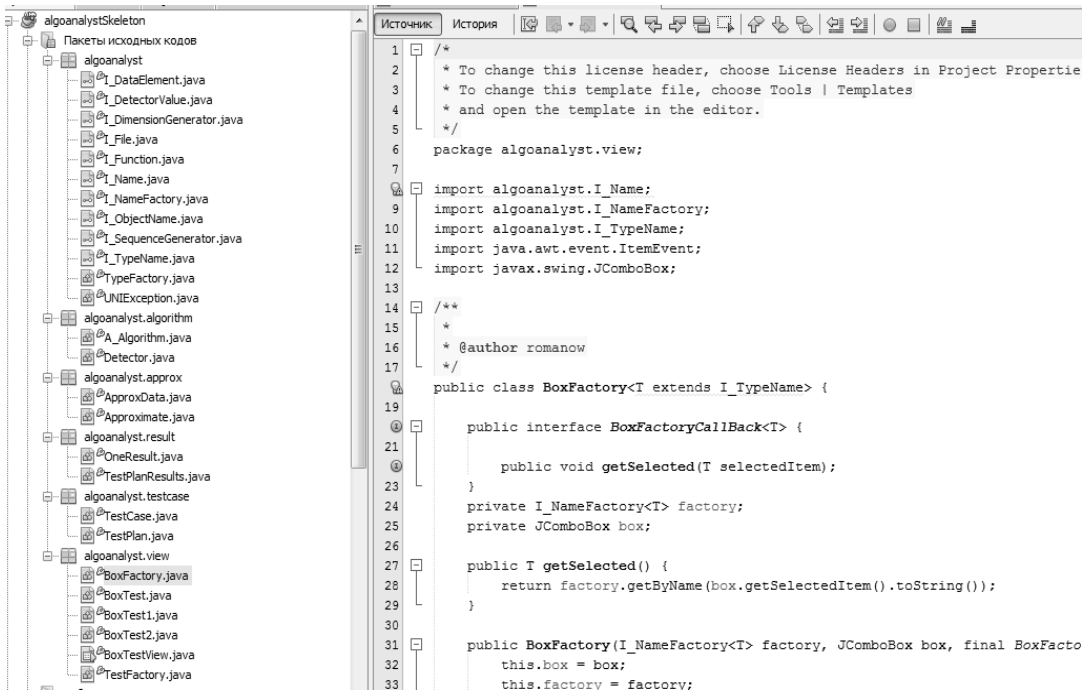


Рис. 9.2. Каркас проекта

- *I_DimensionGenerator* – интерфейс генератора размерностей для тестового плана. Классы генераторов: линейная, геометрическая, последовательность чисел Фибоначчи;
- *I_SequenceGenerator* – интерфейс генератора последовательности с заданием набора произвольных параметров Классы генераторов под разные типы тестируемых данных: случайные, монотонно возрастающие, убывающие данные и т. д.;
- *TestCase* – тестовый набор, вектор единиц тестируемых данных одного типа, при генерации получает размерность, объект-генератор и объект-фабрику;
- *TestPlan* – тестовый план, вектор тестовых наборов;
- *I_DetectorValue* – интерфейс измеряемого параметра, виды параметров: время в мс, счетчик операций;
- *Detector* – датчик, вектор значений измеряемого параметра *I_DetectorValue;*
- *OneResult* – результат измерения, вектор размерностей и датчик;
- *TestPlanResults* – результат эксперимента, вектор объектов-прототипов – измеряемых параметров, вектор размерностей тестовых наборов, вектор датчиков;
- *A_Algorithm* – алгоритм, базовый класс тестируемой программы. Для измеряемых параметров класс генерирует вектор прототипов датчиков с оригинальными именами. Создает датчик для «грязного» времени выполнения. При выполнении алгоритма получает набор инициализированных датчиков, тестовый набор и интерфейс обратного вызова для событий исполнения: оценочное значение прогресса, события (начало, завершение, исключение, трассировочное сообщение);
- графики – отображение и редактирование списка результатов (добавление, удаление, перенос на передний и задний план, линейные и логарифмические масштабирования по осям, автоматическое масштабирование, ручные движки масштаба);
- *ApproxData, Approximate* – подбор функциональной зависимости для измеряемого параметра из набора функций с интерфейсом *I_Function*: линейная, логарифмическая, линейно-логарифмическая, квадратичная, кубическая, степенная, экспоненциальная, факториал. Определение масштабных коэффициентов путем минимизации общего отклонения по методу наименьших квадратов. Метод оптимизации – градиентный спуск.

Метрика проекта

Собираются данные по объему реализованного кода, трудозатратам (в человеко-часах), приросту трудоемкости и объема, производительности (рис. 9.3).

ревизия		77	91	99	% кода
Пакеты	Неделя 4	Неделя 5	Неделя 6	Неделя 7	
results	172	206	211	338	7,7%
algo	84	262	262	363	8,3%
analyse	44	335	464	769	17,5%
root	35	35	36	81	1,8%
wiev	3	364	565	930	21,2%
exprerience	3	3	3	15	0,3%
data	267	294	304	521	11,9%
testcase	254	338	344	503	11,5%
graph	204	192	170	867	19,8%
Всего	1066	2029	2359	4387	100,0%
Недели	4	5	6	7	
Реальная ч*мес	**1,13**	**1,41**	**1,69**	**1,97**	
Прирост чел*мес реальный	1,13	0,28	0,28	0,28	
Строк / чел.день	47	72	70	111	
чел*мес вычисленный	**2,6**	**5,0**	**5,9**	**11,3**	
Срок	**2,6**	**3,3**	**3,5**	**4,4**	
	a	b	c	d	
1. Обсуждение проекта	2,4	1,05	2,5	0,38	
2. Разработка каркаса					
3. Разработка буксующая					

Рис. 9.3. Метрика проекта

Производится анализ динамики проекта, оценивается степень расхождения с теорией по методике COCOMO.

9.3. Программная инженерия

Лабораторный практикум

Цикл лабораторных работ по обзорному курсу «Программная инженерия» включает моделирование различных дисциплин и деятельностей в процессе проектирования программной системы по заданному варианту. Для результатов очередной лабораторной работы должна быть проведена верификация (тестирование) на предмет соответствия результатам предыдущих.

Варианты заданий

1. **Система продажи билетов в кинотеатре.** Клиент, кассир, смена, билетер, администратор, планирование сеансов, продажа билетов кассиром, бронирование и продажа через Интернет, финансовые отчеты, план зала.

2. **Система продажи театральных билетов.** Приложение кассира – множество точек продажи, приложение распространителя, бронирование через Интернет, связь с платежными системами, план зала (метауровень описания), спектакли, репертуар.

3. **Система автоматизации диспетчерской службы такси.** Диспетчер, водитель, клиент, директор, прием заказов, ведение очередей, ручное распределение заказов, приложение водителя, мониторинг прохождения заказа.

4. **Система автоматизированного заказа такси через Интернет.** Серверное приложение для автоматического распределения заказов с учетом нагрузки, web-приложение и мобильное приложение для клиентов, приложение водителя, приложение администратора для форс-мажорных и конфликтных ситуаций. Автоматическое распределение заказов на основе расстояния для клиента и других критериев, предложение свободных заказов водителю, голосование за заказ, мониторинг прохождения заказа. Виды адресов с привязкой к GPS-координатам: почтовый, место, корпоративный (фирма, организация).

5. **Система ведения корпоративной адресной базы для мобильных клиентов.** Типы адресов: служебный – подразделение, корпус, кабинет; домашний – почтовый. Типы контактов: электронная почта, телефон, социальная сеть, адрес. Административная структура организации. Хранение списка контактов, обмен контактами, иерархическая многомерная адресная книга с каталогами (тегами), общая и личная адресные книги.

6. **Мессенжер мобильных клиентов** (аналогичный WhatsUp). Регистрация по номеру мобильного телефона. Передача сообщений, файлов, синхронизация адресных книг, иерархическая многомерная адресная книга с каталогами (тегами), поиск по общей адресной книге, личная адресная книга с собственной системой каталогов (тегов)

7. **Мессенжер с прямой связью клиентов.** Сервер контактов, регистрация, сохранение IP-адресов клиента и статистики пребывания, авторизация с сообщением текущего IP-адреса, дозвон по списку IP-адресов, прямая связь с передачей сообщений и файлов, локальные адресные книги, обмен адресами.

8. **Система заказных грузоперевозок по городу.** Клиент, диспетчер, магазин, водитель. Прием и оформление заказов, отчеты и сопроводительные документы, распределение заказов диспетчером. Два вида заказов: точка–точка и развоз товаров со склада по клиентам. Приложение водителя: просмотр

заказов, мониторинг проведения заказа, планирование последовательности исполнения для развоза, времени доставки. Приложение диспетчера: прием и оформление заказа, распределение, планирование доставки. Параметры заказа – вес и габариты грузов. Транспортные средства и водители. Оплата доставки авансом и при выполнении заказа.

9. **Система продажи билетов на междугородные автобусы.** Планирование рейсов, расписание, чартерные рейсы, типы автобусов, планы рассадки, водители, кассиры, смены, визуализация рассадки, приобретение билетов в кассе и в кассовых терминалах, бронирование через Интернет, сводные отчеты по маршруту и дате.

10. **Система мониторинга междугородных автобусных перевозок.** Карта автодорог, маршрут, график движения по маршруту, планирование рейсов, GPS-навигация транспортных средств, отслеживание графика движения по маршруту, обработка аварийных ситуаций.

11. **Система планирования междугородных транспортных перевозок.** Транспортные средства – тип, тоннаж, вместимость. Населенные пункты, дорожная сеть, прием заявок, поиск подходящего транспорта с учетом его местонахождения и порожнего прогона, составление маршрута, оформление и проводка заявок, отчеты по периоду времени, транспортному средству, водителю.

12. **Система мониторинга междугородных транспортных перевозок.** Населенные пункты, дорожная сеть, маршрут, планирование движения по маршруту, GPS-навигация транспортных средств, отслеживание графика движения по маршруту, обработка аварийных ситуаций, отслеживание заправки, отчеты по расходу топлива и рабочему времени водителей.

13. **Справочная система наличия товаров.** Многоуровневая система категорий и марок товара. Метасистема классификационных признаков и их значений, например вес, цвет, производитель, объем памяти, наличие GPS и т. п. Торговые сети, торговые точки с привязкой к GPS-координатам. Ассортимент в торговой точке, количество товара. Приложение пользователя: поиск по местоположению, по условиям, сформированным для признаков. Приложение торговой точки – редактирование ассортимента. Приложение администратора – редактирование категорий и марок, классификационных признаков.

14. **Логистическая система интернет-магазина с пунктами выдачи и доставкой по городу.** Прием заказов, оформление заявок поставщикам, уведомление клиентов, отслеживание работы курьеров, отчеты по пунктам выдачи. Многоуровневая система категорий и марок товара. Метасистема классификационных признаков и их значений, например вес, цвет, производитель,

объем памяти, наличие GPS и т. п. Наличие товара на складе. Отслеживание балансов по каждому виду товара: заказано, наличие на складе, в заявках к поставщикам. Принятие товара на складе, формирование комплектов заказов для пунктов выдачи и курьеров.

15. **Система электронного документооборота учебного процесса в вузе.** Структура учебного заведения, факультеты, кафедры, группы, студенты, преподаватели. Сторонние организации. Приказы о зачислении / отчислении, назначении тем выпускных и курсовых работ, распределение на практику, распоряжения по подразделениям. Прохождение приказа: создание, визирование, утверждение, нумерация, рассылка.

16. **Система книговыдачи школьной библиотеки с использованием QR-кодов.** Систематический и алфавитный каталог книг с учетом экземпляров, рекомендованные учебники по предметам, структура учебного процесса: класс, ученик, предметы, преподаватели, сторонние лица. Формирование комплектов учебников, выдача литературы на абонемент. QR-коды читательских билетов, экземпляров книг, выдача и прием, ведение формуляра с историей, уведомление о просрочках.

17. **Система книговыдачи библиотеки с использованием QR-кодов.** Алфавитный каталог книг с учетом экземпляров, многомерный иерархический тематический каталог или система тегов. Выдача литературы на абонемент. QR-коды читательских билетов, экземпляров книг, ведение формуляра с историей, уведомление о просрочках. Ведение очередей на дефицитные книги – артефакты, уведомления об очередности.

18. **Система управления кафе / баром.** План зала, закрепление официантов за столиками, планирование смен. Меню – категории, позиции, описание, привязка к кухне или бару. Проведение заказа: закрепление столика за официантом, выбор по меню, частичный заказ, заявки в бар и на кухню, повторение заказа, итоговый расчет. Приложение посетителя, официанта, бармена – планшет, администратора – desktop.

19. **Система бронирования мест для клубных мероприятий.** План концертов. Анонсы. Стоимость столиков. План зала – метауровень описания, настройка под конкретный клуб. Билеты – столики, танцпол. Электронная предоплата. Бронирование. Приложение кассира. Мобильное или web-приложение клиента.

20. **Система бронирования мест в гостинице.** Метауровень описания конкретной гостиницы – расположение и типы номеров, поэтажные планы, список услуг, фото общие и отдельных номеров, расценки. Бронирование через Интернет, визуализация свободных / занятых номеров, расчет стоимости,

квитанции, отчеты по периодам. Бронирование индивидуальное и групповое. Балансы по занятым, свободным и забронированным номерам по датам и категориям. Заселение, продление проживания, дополнительные услуги, частичный и итоговый расчет.

21. **Система планирования взаимосвязанных работ.** Учет сотрудников и их занятости запланированными работами. Определение работ в виде цепочек заданий с параллельными ветвями. Распределение заданий по работникам с учетом их занятости. Сдвиг сроков зависимых работ при задержке выполнения одной из них. Приложения сотрудника, руководителя подразделения, генерация отчетов.

22. **Система мониторинга обслуживания по заявкам.** Система с предварительным сбором заявок и обслуживанием. Категории и виды работ, исполнители, возможность выполнения ими работ по категориям и видам (квалификация). Прием заявок, планирование исполнения, распределение по исполнителям. Оперативное планирование времени исполнения заявок, отслеживание времени исполнения, коррекция времени при задержках с уведомлением клиентов, отказы. Мобильный клиент сотрудника, приложение диспетчера.

23. **Система поддержки технологии Scrum для удаленной работы.** Поддержка основных элементов технологии Scrum – исполнители (команда), лидер, собственник проекта, истории пользователей (задачи), спринты, бэклоги, мониторинг исполнения задач, диаграммы сгорания. Митинги, покер-планирование в режиме чата.

24. **Система многомерной организации документов.** В качестве документов могут выступать короткие заметки (записи в БД), документы, изображения, звуковые файлы. Поддерживается дерево редактирования версий файла. Создается произвольное количество систем классификаций (каталогов ссылок на файлы) по набору ключевых параметров, например, изображения могут классифицироваться по размеру, цветовой палитре, содержанию, наличию определенных предметов, звуковые файлы – по исполнителю, стилю, наличию музыкальных инструментов.

Лабораторная работа № 1.
Анализ предметной области, разработка видения

Для выбранного варианта разработать документы фазы исследования проекта. Содержание документов:

- глоссарий для значимых элементов предметной области;
- бизнес-требования;
- границы проекта;

- перечень заинтересованных лиц, пользователей проекта и приложений;
- словесное описание бизнес-процессов предметной области;
- формальное описание отдельных бизнес-процессов в виде диаграмм потоков данных или диаграмм деятельности;
- диаграмма классов предметной области.

Лабораторная работа № 2. Разработка модели прецедентов

Разработать полную модель прецедентов, кратко описать роли и содержание прецедентов, расписать сценарии двух-трех наиболее значимых прецедентов, основываясь на модели предметной области.

Лабораторная работа № 3. Разработка требований

Определить полный перечень функциональных и нефункциональных требований к системе. На его основе разработать документ «Спецификация требований к ПО».

Лабораторная работа № 4. Разработка прототипа графического интерфейса

Для всех приложений с учетом их функционала и имеющихся прецедентов разработать систему окон графического интерфейса пользователя (GUI), диаграмму оконных классов или граф связей. Обосновать принятые решения требованиями из «Спецификации требований к ПО». Дополнить спецификацию с учетом выполненного проектирования.

Лабораторная работа № 5. Разработка архитектуры системы

Разработать архитектуру системы, оформить документы технологического процесса проектирования. Содержание документов:

- общее архитектурное решение: архитектурные модели, компоненты (подсистемы), слои, граница разделения клиент / сервер с учетом видов, специфики и способов реализации клиентов, стили реализации отдельных компонент и подсистем;
- структура программного кода – языки реализации, пакеты, основные классы (диаграмма пакетов и классов реализации);
- структура программных компонент и артефактов – структура БД, форматы файлов данных, конфигурационных файлов, интерфейсы служб и сторонних библиотек (диаграмма компонентов);

- средства коммуникаций, протоколы, форматы, стандарты обмена и хранения данных;
- описание внутреннего параллелизма и синхронизации программных компонент.

Лабораторная работа № 6. Реализация прецедента

Для одного из сценариев, разработанного в лабораторной работе № 2, проиллюстрировать его архитектурную реализацию (диаграмма устойчивости, диаграмма последовательности или коммуникационная диаграмма).

Лабораторная работа № 7. Оценка трудоемкости и стоимости программного проекта

На основе имеющихся данных оценить объем программного кода. Провести оценку трудоемкости и сроков реализации методами PERT и COCOMO II.

Лабораторная работа № 8. Разработка бизнес-плана программного проекта

Определить этапы реализации, деятельности, структуру команды разработчиков и план исполнения отдельных работ (деятельностей).

ГЛОССАРИЙ

A priory (лат. до опыта) – априорный, известный заранее, по определению.

Ad hoc (лат. к этому) – для данного случая, для этой цели. Частное техническое решение, принятое для конкретного случая. Программный код, решающий проблему частным образом по месту возникновения.

AJAX, Ajax (Asynchronous Javascript and XML) – технология передачи серверу асинхронных запросов программами на JavaScript, не привязанных к процессу обновления web-страниц браузером. Сервер возвращает сериализованный текстовый ответ в XML-формате той компоненте, которая выполнила вызов.

API (Application Programming Interface) – интерфейс программирования приложений, интерфейс прикладного программирования. Программный интерфейс в виде набора согласованных процедур для работы прикладных программ с соответствующим сервисом.

BE (Big endian) – последовательность размещения в памяти или передачи по каналу связи байтов машинного слова, начиная со старшего байта. Используется процессорами IBM 360/370/390, Motorola 68000, SPARC, а также в двоичных потоках ввода / вывода в Java.

DAO (Data Access Objects) – объекты доступа к данным. Средство для работы с БД. Класс, создающий однозначное отображение своих наследников на таблицы БД по принципу «свойство (данные) класса – поле записи таблицы». Один из способов реализации ORM.

Desktop – компьютер с общепринятым интерфейсом пользователя в виде многооконного интерфейса, в отличие от планшетного или мобильного.

GUI (Graphical User Interface), графический интерфейс пользователя, общепринятая аббревиатура для всех интерфейсных компонент программной системы, взаимодействующих с пользователем.

IDE (Integrated Development Environment) – интегрированная среда разработки, система программных средств, включающая средства редактирования,

компиляции, локальной и удаленной сборки проекта, отладки, профилирования, контроля версий, фреймвовки для конкретных видов приложений (мобильных, web, серверных и т. п.).

IT (Information Technologies) – информационные технологии.

JVM (Java Virtual Machine) – виртуальная машина Java. Архитектурно зависимая компонента исполнения байт-кода Java на конкретной платформе.

Keep-alive (англ. оставаться живым) – периодическая передача сообщений в соединение, свидетельствующая о нормальной работе программы на одном из концов соединения. Неполучение keep-alive-сообщений на другом конце соединения в течение заданного интервала рассматривается как сбой, соединение принудительно разрывается.

LE (Little endian) – последовательность размещения в памяти или передачи по каналу связи байтов машинного слова, начиная с младшего байта. Является стандартным для архитектуры Intel x86.

Log-файл – см. **логирование.**

MVC (Model-View-Controller) – архитектурный шаблон и шаблон проектирования, принцип разделения внешнего вида (View), внутреннего представления (Model) и поведения (Controller).

ORM (Object-Relational Mapping) – объектно-реляционное отображение. Технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования (см. DAO).

PMBOK (Project Management Body of Knowledge) – обобщающий документ «Свод знаний об управлении проектами».

RUP (Rational Unified Process) – методология и соответствующий ей фреймворк, производимый фирмой IBM, основанные на методологии UP.

Shutdown – процесс завершения работы системы или процесса. В многопоточном процессе завершение может быть продолжительным по времени и асинхронным. Это связано с уведомлением потоков, сохранением ими своего состояния и уведомлением об этом основного потока.

SWEBOK (Software Engineering Body of Knowledge) – обобщающий документ «Свод знаний о программной инженерии».

TDD (Test-driven Development) – техника разработки «тестами вперед» или разработки, движимой тестированием. Инкрементная разработка в виде последовательности шагов: создание тестов под интерфейс функционала, разработка заглушки, наполнение кодом и рефакторинг.

UML (Unified Modeling Language) – унифицированный язык моделирования для описания программной системы на всех этапах жизненного цикла с использованием парадигмы объектного моделирования.

UP (Unified Process) – см. **Унифицированный процесс.**

WebAPI – протокол программного доступа на основе протокола HTTP. WebAPI может быть оригинальным либо представлять собой альтернативу для удаленного программного доступа к данным web-сервера. Возвращает сериализованные текстовые данные в XML- или JSON-формате.

Артефакт – любой искусственно созданный элемент программной системы в процессе ее проектирования: исполняемый файл, исходные текст, веб-страница, справочный файл, сопроводительный документ, файл с данными, модель, база данных (термин UP).

Архитектура – минимально избыточное, согласованное описание системы, включающее структуру, поведение, компоновку, стили разработки и значимые решения, создающее адекватное представление о ключевых моментах ее организации для всех заинтересованных сторон.

Асинхронная процедура (функция) – процедура, вызываемая в контексте основного потока команд, но в произвольный момент времени, обычно по внешнему событию. Асинхронная процедура является аналогом прерывания на уровне процесса.

Асинхронное взаимодействие (асинхронный обмен данными) – возможность исполнения отдельных частей взаимодействия (обмена данными) в произвольный момент времени, без привязки к определенному событию.

Синхронное взаимодействие с опросом – периодический синхронный опрос клиентом накопленных событий на сервере. Для повышения реактивности частота опроса может пропорционально уменьшаться при отсутствии событий и увеличиваться при их появлении.

Асинхронный обратный вызов (англ. Callback) – при передаче запроса серверу или вызове метода в объекте клиент не блокируется и продолжает исполнение потока команд. При получении ответа вызывается асинхронная процедура в контексте вызвавшего запрос потока.

Базовая версия (BaseLine) – спецификация или продукт, части которого были официально рассмотрены и согласованы, чтобы впоследствии служить основой для дальнейшего развития, и которые могут быть изменены только посредством официальных и контролируемых процедур.

Бизнес-процесс – формализованное описание процессов предметной области, лишенное материальной составляющей, описание взаимодействия участников процесса (бизнес-сущностей).

Бизнес-аналитика – описание бизнес-процессов до и после включения в них программной системы.

Бизнес-архитектура – бизнес-цели проекта, обоснование его успешности, экономические, социальные и прочие аспекты разработки программных систем. Обоснование того, что проект «выстрелит».

Бизнес-объект – класс представления бизнес-сущности в программе, удобный для использования как единое целое. Обычно класс бизнес-объекта наследует или делегирует класс бизнес-сущности и содержит ссылки на связанные с ним классы бизнес-сущностей или бизнес-объектов.

Бизнес-правила – особый вид требований, связанных с предметной областью: факты, ограничения, условия, зависимости, которые могут повлиять на функционал, Бейсик предметной области.

Бизнес-слой – в многослойной архитектуре приложения слой, предназначенный для манипулирования бизнес-объектами: создания, уничтожения, связывания, моделирования поведения в целом. Является основной компонентой для реализации функционала.

Бизнес-сущность – сущность предметной области; в модели предметной области, моделях анализа и проектирования представленна соответствующим классом;

Валидация – доказательство того, что система удовлетворяет бизнес-целям проекта, проверка, насколько сами требования адекватны действительности.

Верификация – проверка соответствия артефакта процесса разработки функциональным требованиям и стандартам качества.

Верификация кода формальная – способ формального доказательства правильности программного кода путем анализа утверждений о состоянии данных (предикаты) и их преобразовании операциями и операторами программы.

Дейтаграмма – сообщение ограниченного размера без гарантий доставки, передаваемое без установления логического соединения.

Демон – компьютерная программа, работающая в фоновом режиме без прямого взаимодействия с пользователем, фоновый процесс.

Деятельность – отдельная единица работы процесса разработки программной системы (термин UP).

Динамический порт – порт с номером, выделяемый узлом сети для данного конкретного соединения при установлении его приложением-клиентом. Занятость номеров динамических портов контролируется службой установления соединений.

Динамическое программирование – качественное изменение производительности (трудоемкости) за счет сохранения промежуточных решений для

повторного использования результата при появлении задачи с такими же параметрами.

Дисциплина – рабочий поток высокого уровня, единица технологического процесса проектирования (термин UP), синоним **технологический процесс ЖЦ.**

Жадный алгоритм – алгоритм, который на каждом шаге из нескольких возможных вариантов продолжения для достижения результата выбирает единственный.

Жизненный цикл (ЖЦ) – период существования программной системы (программного обеспечения).

Интерфе́йс (англ. *Interface* – сопряжение, поверхность раздела, перегородка) – граница раздела двух систем, устройств или программ с заданными характеристиками соединения, сигналов и алгоритмов обмена и т. п.

ИТ – информационные технологии.

Клиент-сервер (приложения) – распределение функций «клиент–сервер» между приложениями.

Клиент-сервер (роли в протоколе) – распределение функций в конкретном взаимодействии между протокольными процессами. Событие, инициирующее взаимодействие, поступает в клиентский процесс, который обращается к процессу-серверу.

Контекст – текущее неявное окружение, в котором функционирует процесс.

Контекст кода – неявное окружение, множество программных объектов – типов (классов), переменных и функций (методов), доступных по непосредственным именам. Контекст является понятием, обратным области видимости.

Копипаст – программный код, полученный простым копированием образца и редактированием его по месту использования под новый контекст, а также сам процесс такого копирования.

Логирование – сохранение сообщений программы в текстовый файл (log-файл) или в записях БД. Используется для регистрации сбоев, трассировки исполнения программного кода.

Метауровень Java – представление загруженной программы в виде системы объектов – описателей классов типа Class. Позволяет реализовать значительную часть функций JVM на самой Java, а также обеспечить рефлексию.

Методология – учение о структуре, логической организации, методах и средствах деятельности.

Модель предметной области (модель производства) – формальное описание средствами бизнес-аналитики структуры и процессов предметной

области, подлежащей реорганизации или автоматизации. Исходная модель в цепочке моделей описания функционала программной системы (терминология UP**)**.

Модель анализа – описание программной системы, включенной в бизнес процесс на функциональном уровне.

Область видимости объекта – область программного кода, в котором объект доступен по имени – виден.

Обратный инжиниринг кода – восстановление или создание исходного текста программы (кода) по спецификации, функциональному описанию, исследованию поведения или исполняемому коду существующего программного модуля. Восстановление структурного описания проекта (детальной архитектуры) по его исходному коду.

Онтология – раздел философии, учение о сущем, о фундаментальных принципах бытия. Онтология системной инженерии – описание самых общих принципов и сущностей (альф) в данной области.

ООП – объектно-ориентированное программирование, парадигма программирования, идеи которой положены в основу одноименной методологии проектирования.

Парсинг – получение объекта с уникальными данными из текстового описания его содержимого в определенном формате (XML, JSON) с использованием элементов синтаксического анализа.

Парсер – программная компонента, выполняющая такое преобразование.

Поток (поток управления) – последовательность действий программного кода, исполняемая параллельно (независимо, асинхронно) с другими потоками в едином адресном пространстве процесса.

Прецедент – атомарное, ограниченное по времени и функционалу взаимодействие пользователя с системой.

Программная система (ПС) – программное обеспечение в сочетании с аппаратным, программным и пользовательским окружением.

Программное обеспечение (ПО) – набор компьютерных программ, процедур, связанной с ними документации и данных.

Программный продукт (ПП) – программное обеспечение как товар (услуга), включающий само ПО, средства и сервисы установки, обновления, сопровождения и поддержки среды функционирования.

Протокол – описание правил взаимодействия параллельных независимых (асинхронных) процессов путем обмена сообщениями через ненадежную инертную пространственную среду.

Протокольные процессы – процессы в операционной системе, которые реализуют протокол.

Процесс – единица управления и планирования в ОС. Характеризуется адресным пространством, загруженным в него кодом и связанными с ним ресурсами.

Рабочий поток – множество деятельностей, связанных результатами (артефактами) и составляющих единую компоненту процесса разработки (термин UP).

Распределенная система (в широком смысле) – система, элементы которой распределены по пространственно разнесенным узлам.

Реинжиниринг кода – коренная перестройка функциональности, архитектуры или структуры кода проекта в целом с использованием существующего программного кода.

Релиз – выпуск готового программного продукта, сам готовый программный продукт в очередной версии.

Рефакторинг кода – изменение (улучшение) структуры и качества кода без изменения его функциональности, интерфейсов и поведения.

Рефлексия – возможность программы получить описание любого класса и через это описание выполнять операции над классом, синтаксически не используя его имени: порождать объекты, вызывать методы и т. п. Возможность программы анализировать и управлять собственной структурой.

Роль – исполнение одним разработчиком нескольких связанных видов деятельности (термин UP).

Сериализация – передача в потоке данных значений полей данных объекта и, возможно, его описания.

Сетевой порт (точка доступа) – числовой идентификатор, обозначающий приложение или сервис, с которым устанавливается соединение.

Синхронизация потоков – установление ограничений на порядок исполнения фрагментов кода потоков при наличии между ними причинно-следственной связи или логической связи по используемым данным.

Синхронизация причинно-следственная – один или несколько потоков ожидают некоторого действия со стороны другого потока, события в другом потоке.

Синхронизация разделения ресурса – использование одного и того же ресурса несколькими потоками; монопольное использование ресурса в критической секции, которое не может быть прервано другой такой же операцией.

Синхронная процедура (функция) – процедура, вызов которой производится явно в определенной точке основного потока команд, т. е. синхронно с его исполнением.

Синхронное взаимодействие (обмен данными) – наличие у протокольного процесса логической привязки всего взаимодействия (обмена данными) к какому-либо одному событию.

Синхронный / асинхронный (в вычислительных процессах) – наличие логической привязки исполнения отдельных частей программного кода, приводящее к корреляции их исполнения во времени.

Синхронный / асинхронный (в широком толковании) – наличие или отсутствие одновременности выполнения или привязки во времени.

Система – в данном изложении сокращенное наименование **программной системы**.

Система с распределенным управлением – распределенная система, в которой административные функции управления и контроля не сосредоточены в одном узле, а функционально распределены по узлам.

Системная инженерия – наука об общих принципах решения задачи с учетом технического, технологического и социального окружения, оценки оптимальности, сбалансированности, эффективности ее решения.

Статический порт – порт с фиксированным номером, зарегистрированный или общеизвестный для соединения с конкретной службой.

Сценарий – текстовое описание потока событий при выполнении конкретного варианта использования, выражающее некий аспект поведения системы (термин UP).

Тайм-аут – интервал времени, в течение которого процесс ожидает наступления события. Истечение тайм-аута рассматривается как ошибка взаимодействия либо инициирует действия по восстановлению. При наступлении события тайм-аут сбрасывается.

Тестирование (как деятельность) – исполнение (воспроизведение поведения) экземпляра артефакта при заданном входе.

Тестирование (как элемент жизненного цикла) – деятельность, направленная на выявление ошибок в различных артефактах проекта (цели, требования, архитектура, код, сборка, релиз, документация и в проекте в целом).

Транзакция – атомарное короткое взаимодействие между клиентом и сервером без последействия и использования контекста. Все необходимые данные должны быть переданы клиентом в параметрах транзакции, создаваемое сервером окружение (контекст) для исполнения запроса не сохраняется и повторно не используется.

Трудоемкость программы (алгоритма) – зависимость количества массовых операций (сравнения, обмены, сдвиги, повторения цикла) от размерностей обрабатываемых данных.

Удаленный вызов процедур (Remote Procedure Call, RPC) – технологический вариант синхронного взаимодействия «клиент–сервер», в котором клиент передает серверу имя и параметры вызываемой процедуры и получает в ответ результаты ее выполнения на сервере

Унифицированный процесс (Unified Process, UP) – тяжеловесная методология на основе итеративной модели жизненного цикла, идеях и артефактах ООП и документировании средствами UML.

Фаза (этап жизненного цикла) – самый крупный временной отрезок жизненного цикла, в результате которого достигается уникальное качественное состояние разработки (термины UP).

Формат – описание порядка следования данных в последовательном потоке. В формате очередной элемент является либо элементом данных, либо управляющим элементом, содержащим параметры следующих за ним данных. Формат является саморазворачивающимся, может быть иерархическим или рекурсивным.

Форматная строка – строка, содержащая специальные символы, на место которых подставляются значения в порядке их следования за самой строкой.

Фреймворк (англ. framework – каркас, структура) – программная система разработки приложений, создающая начальный каркас структуры и кода, в который заливается конкретное содержимое.

Функционально распределенная система – распределенная система, в которой некоторая ее функция (функциональность) реализуется в нескольких узлах.

БИЛИОГРАФИЧЕСКИЙ СПИСОК

1. Профессиональные стандарты [Электронный ресурс] / Министерство труда и социальной защиты Российской Федерации. – Режим доступа: http://profstandart.rosmintrud.ru (дата обращения: 07.09.2017).

2. *Орлик С.* Перевод SWEBOK. Основы программной инженерии (по SWEBOK) [Электронный ресурс] / С. Орлик. – Режим доступа: http://www.studmed.ru/orlik-s-perevod-swebok_8ff66e3c6e3.html (дата обращения: 07.09.2017).

3. О преподавании программной инженерии [Электронный ресурс]. – Режим доступа: http://www.interface.ru/home.asp?artId=1064 (дата обращения: 07.09.2017).

4. *Бюрер К.* От ремесла к науке: поиск основных принципов разработки ПО [Электронный ресурс] / К. Бюрер. – Режим доступа: http://www.interface.ru/home.asp?artId=4820 (дата обращения: 07.09.2017).

5. *Boehm B. W.* A spiral model of software development and enhancement / B. W. Boehm // Computer. – 1988. – Vol. 21, iss. 5. – P. 61–72.

6. Guide to Software Engineering Base of Knowledge (SWEBOK) [Electronic resource] / IEEE Computer Society. – 2004. – Available at: http://www.swebok.org/ (accessed: 07.09.2017).

7. *Фаулер М.* UML. Основы : краткое руководство по стандартному языку объектного моделирования / М. Фаулер. – 3-е изд. – Санкт-Петербург : Символ-Плюс, 2004. – ISBN 5-93286-060-X.

8. Рекомендации к стилю кода [Электронный ресурс]. – Режим доступа: http://habrahabr.ru/post/112042/ (дата обращения: 07.09.2017).

9. *Романов Е. Л.* Язык программирования Си/Си++. От дилетанта до профессионала [Электронный ресурс] / Е. Л. Романов. – Режим доступа: http://ermak.cs.nstu.ru/cprog (дата обращения: 07.09.2017).

10. *Вязовик Н. А.* Программирование на Java : курс лекций / Н. А. Вязовик. – Москва : Интернет-университет информ. технологий, 2003. – 586 с.

11. *Васильев А. Н.* Java. Объектно-ориентированное программирование : для магистров и бакалавров : базовый курс по объектно-ориентированному программированию : [учебное пособие] / А. Н. Васильев. – Санкт-Петербург [и др.] : Питер, 2011. – 395 с.

12. *Макконнелл С.* Совершенный код : пер. с англ. / С. Макконнелл. – Москва : Русская Редакция ; Санкт-Петербург : Питер, 2005. – 896 с. – (Мастер-класс).

13. *Гранд М.* Шаблоны проектирования в Java / М. Гранд ; пер. с англ. С. Беликовой. – Москва : Новое знание, 2004. – 599 с.

14. *Стелтинг С.* Применение шаблонов Java : библиотека профессионала : пер. с англ. / С. Стелтинг, О. Маасен. – Москва : Вильямс, 2002. – 576 с.

15. Программный код и его метрики [Электронный ресурс]. – Режим доступа: https://habrahabr.ru/company/intel/blog/106082/ (дата обращения: 07.09.2017).

16. *Ледовских И.* Метрики сложности кода [Электронный ресурс] : технический отчет 2012-2 / И. Ледовских ; Институт системного программирования РАН. – Режим доступа: http://www.ispras.ru/preprints/docs/prep_25_2013.pdf (дата обращения: 07.09.2017).

17. Официальный сайт проекта SonarQube [Электронный ресурс]. – Режим доступа: http://www.sonarqube.org (дата обращения: 07.09.2017).

18. SourceCodeMetrics – plugin detail [Electronic resource] // NetBeans : website. – Available at: http://plugins.netbeans.org/plugin/42970/sourcecodemetrics (accessed: 07.09.2017).

19. Облачный сервис CodeNForcer [Электронный ресурс]. – Режим доступа: http://cloud.codenforcer.com (дата обращения: 07.09.2017).

20. *Матюшкин Е.* Синхронизация потоков [Электронный ресурс] // Е. Матюшкин. Записки трезвого практика : web-сайт. – Режим доступа: http://www.skipy.ru/technics/synchronization.html (дата обращения: 07.09.2017).

21. *Арлоу Д.* UML 2 и Унифицированный процесс: практический объектно-ориентированный анализ и проектирование : пер. с англ. / Д. Арлоу, И. Нейштадт. – 2-е изд. – Санкт-Петербург : Символ-Плюс, 2007. – 624 с.

22. *Кратчен Ф.* Введение в Rational Unified Process [Электронный ресурс] / Ф. Кратчен. – Режим доступа: http://www.proklondike.com/books/upravlenie/kratchen_rup_intro.html (дата обращения: 07.09.2017).

23. *Амблер С.* Гибкие технологии : экстремальное программирование и унифицированный процесс разработки / С. Амблер. – Санкт-Петербург : Питер, 2005. – ISBN 5-94723-545-5.

24. *Орлов С.* Технологии разработки программного обеспечения : учебник / С. Орлов. – Санкт-Петербург : Питер, 2002. – 464 с.

25. RUP. Method Composer [Электронный ресурс]. – Режим доступа: http://dit.isuct.ru/Publish_RUP (дата обращения: 08.09.2017).

26. *Бек К.* Экстремальное программирование / К. Бек. – Санкт-Петербург : Питер, 2002. – 224 с. – ISBN 5-94723-032-1.

27. Объектно-ориентированный анализ и проектирование с примерами приложений : пер. с англ. / Г. Буч, Р. А. Максимчук, М. У. Энгл, Б. Дж. Янг, Д. Коналлен, К. А. Хьюстон. – 3-е изд. – Москва : Вильямс, 2008. – 720 с.

28. *Соммервилл И.* Инженерия программного обеспечения / И. Соммервилл. – 6-е изд. – Москва [и др.] : Вильямс, 2002. – 623 с. – ISBN 5-8459-0330-0.

29. *Головач В. В.* Дизайн пользовательского интерфейса [Электронный ресурс] / В. В. Головач. – Режим доступа: http://www.proklondike.com/books/usability/golovach_ui_design.html (дата обращения: 08.09.2017).

30. *Головач В. В.* Дизайн пользовательского интерфейса: искусство мыть слона [Электронный ресурс] / В. В. Головач. – Режим доступа: http://uibook2.usethics.ru (дата обращения: 08.09.2017).

31. Памятка UX / UI дизайнеру. 19 принципов построения интерфейсов [Электронный ресурс]. – Режим доступа: http://habrahabr.ru/company/SECL_GROUP/blog/182208 (дата обращения: 08.09.2017).

32. *Вигерс К. И.* Разработка требований к программному обеспечению / К. И. Вигерс. – Москва : Русская редакция, 2004. – XVIII, 554 с.

33. *Удовиченко Ю.* Управление изменениями и кессонная болезнь проектов [Электронный ресурс] / Ю. Удовиченко. – Режим доступа: http://experience.openquality.ru/software-configuration-management/ (дата обращения: 08.09.2017).

34. *Макконнелл С.* Остаться в живых! Руководство для менеджера программных проектов / С. Макконнелл. – Санкт-Петербург : Питер, 2006.

35. *Архипенков С.* Лекции по управлению программными проектами [Электронный ресурс] / С. Архипенков. – Режим доступа: http://www.arkhipenkov.ru/ (дата обращения: 14.09.2017).

36. *Макконнелл С.* Сколько стоит программный проект / С. Макконнелл. – Москва : Русская Редакция ; Санкт-Петербург : Питер, 2007. – 297 с.

37. Руководство к Своду знаний по управлению проектами. (Руководство PMBOK®). – 4-е изд. – Newtown Square, PA : Project Management Institute, 2008. – 241 с.

38. *Брукс Ф.* Мифический человеко-месяц или как создаются программные системы / Ф. Брукс. – 2-е изд. – Санкт-Петербург : Символ, 2005. – ISBN 5-93286-005-7.

39. *Иванова В.* Путь аналитика : практическое руководство IT-специалиста / В. Иванова, А. Перерва. – 2-е изд. – Санкт-Петербург : Питер, 2015. – 304 с. – ISBN 978-5-496-01679-7.

40. Полезные метрики для оценки проектов [Электронный ресурс]. – Режим доступа: http://habrahabr.ru/post/141671/ (дата обращения: 14.09.2017).

41. *Коуберн А.* Каждому проекту своя методология [Электронный ресурс] / А. Коуберн. – Режим доступа: http://www.maxkir.com/sd/methyperproject_RUS.htm (дата обращения: 14.09.2017).

42. BPMN 2.0. Из чего состоит модель бизнес-процесса [Электронный ресурс]. – Режим доступа: http://rzbpm.ru/knowledge/bpmn-2-0-iz-chego-sostoit-model-biznes-processa.html (дата обращения: 14.09.2017).

43. Главное не результат, главное процесс. BPM-блог Анатолия Белайчука [Электронный ресурс]. – Режим доступа: http://mainthing.ru/ru/ (дата обращения: 14.09.2017).

44. *Левенчук А. И.* Материалы к курсу «Системно-инженерное мышление» [Электронный ресурс] / А. И. Левенчук. – Режим доступа: http://techinvestlab.ru/systems_engineering_thinking/ (дата обращения: 14.09.2017).

45. *ДеМарко Т.* Вальсируя с Медведями : управление рисками в проектах по разработке программного обеспечения / Т. ДеМарко, Т. Листер. – Москва : Компания p.m.Office, 2005.

46. Software cost estimation with COCOMO II / B. Boehm [et al.]. – Upper Saddle River, NJ : Prentice Hall, 2000.

47. *Вольфсон Б.* Гибкие_методологии разработки [Электронный ресурс]. – Режим доступа: https://tados.ru/wp-content/uploads/2017/04/Борис_Вольфсон_Гибие_методологии.pdf (дата обращения: 03.05.2017).

48. О чем молчит диаграмма Ганта или почему проекты всегда опаздывают [Электронный ресурс]. – Режим доступа: http://habrahabr.ru/post/193592/ (дата обращения: 14.09.2017).

49. Калькулятор COCOMO II [Электронный ресурс]. – Режим доступа: http://csse.usc.edu/csse/research/COCOMOII/cocomo_downloads.htm (дата обращения: 14.09.2017).

50. Руководство к своду знаний по системной инженерии (SEBoK) [Электронный ресурс]. – Режим доступа: http://sebokwiki.org/wiki/Guide_to_the_Systems_Engineering_Body_of_Knowledge_(SEBoK) (дата обращения: 14.09.2017).

51. *Хэррис Д. М.* Цифровая схемотехника и архитектура компьютера / Д. М. Хэррис, С. Л. Хэррис. – 2-е изд. – [Б. м.] : Morgan Kaufman, 2013. – 1621 с. – ISBN 978-0-12-394424-5.

52. Нейрон. Обработка сигналов. Пластичность. Моделирование : фундаментальное руководство / Ю. И. Александров [и др.] ; под ред. Е. Н. Соколова, В. А. Филиппова, А. М. Черноризова. – Тюмень : Изд-во Тюмен. гос. ун-та, 2008. – 548 с.

53. Documents Associated With Essence™ – Kernel And Language For Software Engineering Methods, Version 1.1 [Electronic resource]. – Available at: http://www.omg.org/spec/Essence/1.1/ (accessed: 14.09.2017).

54. System and Software Engeneering – Architecture Description. ISO/IEC/IEEE 42010 [Electronic resource]. – Available at: http://www.iso-architecture.org/42010/index.html (accessed: 14.09.2017).

55. *Kruchten P.* The "4+1" view model of software architecture [Electronic resource] / P. Kruchten. – Available at: https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf (accessed: 14.09.2017).

56. Путь ИТ архитектора [Электронный ресурс]. – Режим доступа: http://itarchitect.ru (дата обращения: 14.09.2017).

57. *Котляров В. П.* Основы тестирования программного обеспечения / В. П. Котляров, Т. В. Коликова. – Москва : Интернет-университет информационных технологий, 2006. – 285 с.

58. *Канер С.* Тестирование программного обеспечения : фундаментальные концепции менеджмента бизнес-приложений : пер. с англ. / С. Канер, Д. Фолк, Енг Кек Нгуен. – Киев : ДиаСофт, 2001. – 544 с.

59. *Майерс Г.* Искусство тестирования программ : пер. с англ. / Г. Майерс. – Москва : Финансы и статистика, 1982. – 176 с.

60. *Макгрегор Д.* Тестирование объектно-ориентированного программного обеспечения : практическое пособие : пер. с англ. / Д. Макгрегор, Д. Сайкс. – Киев : ДиаСофт, 2002. – 432 с.

61. *Степанченко И. В.* Методы тестирования программного обеспечения : учебное пособие / И. В. Степанченко. – Волгоград : ВолгГТУ, 2006. – 74 с.

62. *Бек К.* Экстремальное программирование: разработка через тестирование / К. Бек. – Санкт-Петербург : Питер, 2003. – 224 с. – (Библиотека программиста).

63. *Мищевский Г.* Разработка. Тестирование. Фундаментальная теория [Электронный ресурс] / Г. Мищевский. – Режим доступа: https://habrahabr.ru/post/279535/ (дата обращения: 15.09.2017).

64. *Кулямин В. В.* Методы верификации программного обеспечения [Электронный ресурс] / В. В. Кулямин // Информационно-коммуникационные технологии в образовании. Система федеральных образовательных порталов. – Режим доступа: http://www.ict.edu.ru/ft/005645/62322e1-st09.pdf (дата обращения: 15.09.2017).

65. *Мазурок И.* Знаете ли Вы массивы? [Электронный ресурс] / И. Мазурок. – Режим доступа: http://habrahabr.ru/post/211747/ (дата обращения: 15.09.2017).

66. Отладка самолета? Это очень просто [Электронный ресурс]. – Режим доступа: https://habrahabr.ru/post/202184/ (дата обращения: 15.09.2017).

67. *Аджиев В.* Мифы о безопасном ПО: уроки знаменитых катастроф [Электронный ресурс] / В. Аджиев // Открытые системы. СУБД. – 1998. – № 6. – Режим доступа: http://www.osp.ru/os/1998/06/179592/ (дата обращения: 15.09.2017).

68. *Гурин Р. Е.* Методы верификации программного обеспечения [Электронный ресурс] / Р. Е. Гурин, И. В. Рудаков, А. В. Ребриков // Наука и образование. – 2015. – № 10. – Режим доступа: http://technomag.neicon.ru/doc/823129.html (дата обращения: 15.09.2017).

69. Руководство Microsoft по проектированию архитектуры приложений [Электронный ресурс]. – 2-е изд. – 2009. – Режим доступа: http://download.microsoft.com/documents/rus/msdn/ры_приложений_полная_книга.pdf (дата обращения: 15.09.2017).

70. *Рогачев С.* Обобщенный Model-View-Controller [Электронный ресурс] / С. Рогачев. – Режим доступа: http://rsdn.ru/article/patterns/generic-mvc.xml (дата обращения: 15.09.2017).

71. *Базь М.* Разделение визуализации и бизнес-логики [Электронный ресурс] / М. Базь. – Режим доступа: https://ru.wikiversity.org/wiki/Разделение_визуализации_и_бизнес-логики (дата обращения: 15.09.2017).

72. *Олифер В. Г.* Компьютерные сети. Принципы, технологии, протоколы : [учебное пособие для вузов по направлению «Информатика и вычислительная техника» и по специальностям «Вычислительные машины, комплексы, системы и сети», «Программное обеспечение вычислительной техники и автоматизированных систем»] / В. Олифер, Н. Олифер. – Санкт-Петербург [и др.] : Питер, 2012. – 943 с.

73. *Паркер Т.* TCP/IP / Т. Паркер, К. Сиян. – 3-е изд. – Санкт-Петербург : Питер, 2004. – 859 с. – (Для профессионалов).

74. Протокол передачи данных [Электронный ресурс] // Википедия. – Режим доступа: http://ru.wikipedia.org/wiki/Протокол_передачи_данных (дата обращения: 15.09.2017).

75. Толковый словарь: протокол (protocol) [Электронный ресурс]. – Режим доступа: http://your-hosting.ru/terms/p/protocol/ (дата обращения: 15.09.2017).

76. Интерфейс [Электронный ресурс] // Википедия. – Режим доступа: http://ru.wikipedia.org/wiki/Интерфейс (дата обращения: 15.09.2017).

77. TCP-IP крупным планом [Электронный ресурс]. – Режим доступа: http://www.hardline.ru/4/49/1236/1630-48.html (дата обращения: 15.09.2017).

78. *Перри Б.* Java сервлеты и JSP : сборник рецептов : пер. с англ. / Б. Перри. – Изд. 2-е. – Москва : Кудиц Пресс, 2006. – 768 с.

79. *Мамаев М.* Телекоммуникационные технологии (Сети TCP/IP) [Электронный ресурс] : учебное пособие / М. Мамаев. – Режим доступа: http://www.hardline.ru/4/49/1131/ (дата обращения: 15.09.2017).

80. Сериализация в Java [Электронный ресурс]. – Режим доступа: http://habrahabr.ru/post/60317/ (дата обращения: 15.09.2017).

81. *Андрюнин К.* Java. HTTP протокол и работа с WEB [Электронный ресурс] / К. Андрюнин. – Режим доступа: http://www.javaportal.ru/java/articles/java_http_web/article05.html (дата обращения: 15.09.2017).

82. RFC 2.0 – Русские переводы RFC [Электронный ресурс]. – Режим доступа: http://rfc2.ru/ (дата обращения: 15.09.2017).

83. *Романов Е. Л.* Преподавание программной инженерии. Взгляд от кода [Электронный ресурс] / Е. Л. Романов, Г. В. Трошина, А. А. Якименко // Науковедение. – 2016. – Т. 8, № 5. – Режим доступа: http://naukovedenie.ru/PDF/66TVN516.pdf (дата обращения: 15.09.2017).

84. Reflection в Java [Электронный ресурс]. – Режим доступа: http://www.javenue.info/post/84 (дата обращения: 15.09.2017).

85. Jsoup: Парсинг Html в Android-приложении [Электронный ресурс]. – Режим доступа: http://findevelop.blogspot.ru/2011/04/jsoup-html-android.html (дата обращения: 15.09.2017).

86. Java-XStream [Электронный ресурс]. – Режим доступа: https://github.com/itcuties/Java-XStream (дата обращения: 15.09.2017).

87. *Веденин В.* Шпаргалка Java программиста 8. Библиотеки для работы с Json (Gson, Fastjson, anSquare, Jackson, JsonPath и другие) [Электронный ресурс] / В. Веденин. – Режим доступа: https://habrahabr.ru/company/luxoft/blog/280782/ (дата обращения: 15.09.2017).

88. Использование транзакций в программах на Java [Электронный ресурс]. – Режим доступа: http://oooportal.ru/?cat=article&id=1367 (дата обращения: 15.09.2017).

89. Библиотека Java поддержки WebSocket [Электронный ресурс]. – Режим доступа: https://github.com/TooTallNate/Java-WebSocket (дата обращения: 15.09.2017).

90. *Маккарти Ф.* Ajax для разработчиков. Ч. 1. Строим динамические приложения на языке Java [Электронный ресурс] / Ф. Маккарти. – Режим доступа: http://www.webmas.ru/webprog/ajax/ajax1/6.html (дата обращения: 15.09.2017).

91. Обучение Java. Отслеживание сессий [Электронный ресурс]. – Режим доступа: http://java.markune.ru/servlets/session-tracking.html (дата обращения: 15.09.2017).

92. Direct Web Remoting [Электронный ресурс]. – Режим доступа: http://directwebremoting.org/dwr/ (дата обращения: 15.09.2017).

93. *Бегтин И.* Полезные ресурсы по открытым данным в России [Электронный ресурс] / И. Бегтин. – Режим доступа: http://habrahabr.ru/company/infoculture/blog/201892/ (дата обращения: 15.09.2017).

94. Описание API VKontakte [Электронный ресурс]. – Режим доступа: http://vk.com/dev/main (дата обращения: 15.09.2017).

95. API Яндекс карт. Геокодер [Электронный ресурс]. – Режим доступа: https://tech.yandex.ru/maps/geocoder/ (дата обращения: 15.09.2017).

96. *Романов Е. Л.* Автоматизация учета рейтинга успеваемости студентов / Е. Л. Романов // Открытое и дистанционное образование. – 2014. – № 2 (54). – С. 55–62.

97. Система учета рейтинга успеваемости, ядро [Электронный ресурс]. – Режим доступа: https://bitbucket.org/solus_rex/brs_core (дата обращения: 15.09.2017).

98. Система учета рейтинга успеваемости, desktop-приложения администратора, преподавателя и студента [Электронный ресурс]. – Режим доступа: https://bitbucket.org/solus_rex/brs_desktop (дата обращения: 15.09.2017).

99. Система учета рейтинга успеваемости, серверное приложение тонкого клиента (web-сервис) [Электронный ресурс]. – Режим доступа: https://bitbucket.org/solus_rex/brs_web (дата обращения: 15.09.2017).

100. Система учета рейтинга успеваемости, мобильное android-приложение преподавателя [Электронный ресурс]. – Режим доступа: https://bitbucket.org/solus_rex/brs_android (дата обращения: 15.09.2017).

101. Система учета рейтинга успеваемости, web-приложение преподавателя [Электронный ресурс]. – Режим доступа: https://bitbucket.org/solus_rex/brs_web_client (дата обращения: 15.09.2017).

102. *Романов Е. Л.* Программная инженерия [Электронный ресурс] : электронный учебно-методический комплекс / Е. Л. Романов. – Режим доступа: http://dispace.edu.nstu.ru/didesk/course/show/5164 (дата обращения: 15.09.2017).

УЧЕБНОЕ ИЗДАНИЕ

Романов Евгений Леонидович

ПРОГРАММНАЯ ИНЖЕНЕРИЯ

Учебное пособие

Редактор *Е.Н. Николаева*
Выпускающий редактор *И.П. Брованова*
Художественный редактор *А.В. Ладыжская*
Корректор *И.Е. Семенова*
Компьютерная верстка *С.И. Ткачева*

Подписано в печать 25.12.2017
Формат 70 × 100 1/16. Бумага офсетная
Уч.-изд. л. 31,92. Печ. л. 24,75
Тираж 3000 экз. (1-й з-д – 1–100 экз.)
Изд. № 135. Заказ № 156

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

Издательство Новосибирского государственного
технического университета
630073, г. Новосибирск, пр. К. Маркса, 20
Тел. (383) 346-31-87
E-mail: office@publish.nstu.ru

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20