

ЧИСТАЯ АРХИТЕКТУРА

ИСКУССТВО РАЗРАБОТКИ
ПРОГРАММНОГО
ОБЕСПЕЧЕНИЯ



РОБЕРТ МАРТИН 

ЧИСТАЯ АРХИТЕКТУРА

ИСКУССТВО РАЗРАБОТКИ
ПРОГРАММНОГО
ОБЕСПЕЧЕНИЯ



РОБЕРТ МАРТИН



Роберт Мартин

Чистая архитектура. Искусство разработки программного обеспечения

|



2018

Переводчик *A. Макарова*

Технический редактор *H. Суслова*

Литературный редактор *E. Герасимова*

Художники *L. Егорова, C. Заматевская , P. Яцко*

Корректоры *C. Беляева, H. Викторова*

Верстка *L. Егорова*

Роберт Мартин

Чистая архитектура. Искусство разработки программного
обеспечения. — СПб.: Питер, 2018.

ISBN 978-5-4461-0772-8

© [ООО Издательство "Питер"](#), 2018

*Все права защищены. Никакая часть данной книги не может
быть воспроизведена в какой бы то ни было форме без
письменного разрешения владельцев авторских прав.*

*Посвящается моей любимой супруге, моим четырем замечательным детям и их семьям,
включая пятерых внуков — радость моей жизни*

Предисловие

О чём мы говорим, когда обсуждаем архитектуру?

Так же как любая метафора, описание программного обеспечения с точки зрения архитектуры может что-то скрыть, а что-то, наоборот, проявить; может обещать больше, чем давать, и давать больше, чем обещать.

Очевидная привлекательность архитектуры — это структура. А структура — это то, что доминирует над парадигмами и суждениями в мире разработки программного обеспечения — компонентами, классами, функциями, модулями, слоями и службами, микро или макро. Но макроструктура многих программных систем часто пренебрегает убеждениями или пониманием — организация советских предприятий, невероятные небоскребы-башни Джэнга, достигающие облаков, археологические слои, залегающие в горной породе. Структура программного обеспечения не всегда интуитивно очевидна, как структура зданий.

Здания имеют очевидную физическую структуру, независимо от материала, из которого они построены, от их высоты или ширины, от их назначения и от наличия или отсутствия архитектурных украшений. Их структура мало чем отличается — в значительной мере она обусловлена законом тяготения и физическими свойствами материалов. Программное обеспечение, напротив, никак не связано с тяжестью, кроме чувства серьезности. И из чего же сделано программное обеспечение? В отличие от зданий, которые могут быть построены из кирпича, бетона, дерева, стали и стекла, программное обеспечение строится из программных компонентов. Большие программные конструкции создаются

из меньших программных компонентов, которые, в свою очередь, построены из еще более мелких программных компонентов, и т.д., вплоть до основания.

Говоря об архитектуре, можно сказать, что программное обеспечение по своей природе является фрактальным и рекурсивным, выгравированным и очерченным в коде. Здесь важны все детали. Переплетение уровней детализации также вносит свой вклад в архитектуру, но бессмысленно говорить о программном обеспечении в физических масштабах. Программное обеспечение имеет структуру — множество структур и множество их видов, — но их разнообразие затмевает диапазон физических структур, которые можно увидеть на примере зданий. Можно даже довольно убедительно утверждать, что при проектировании программного обеспечения архитектуре уделяется куда больше внимания, чем при проектировании зданий, — в этом смысле архитектура программного обеспечения является более многообразной, чем архитектура зданий!

Но физический масштаб привычнее людям, и они часто ищут его в окружающем мире. Несмотря на привлекательность и визуальную очевидность, прямоугольники на диаграммах PowerPoint не являются архитектурой программного обеспечения. Да, они представляют определенный взгляд на архитектуру, но принимать прямоугольники за общую картину, отражающую архитектуру, значит не получить ни общей картины, ни понятия об архитектуре: архитектура программного обеспечения ни на что не похожа. Конкретный способ визуализации — не более чем частный выбор. Этот выбор основан на следующем наборе вариантов: что включить; что исключить; что подчеркнуть формой или цветом; что, наоборот, затенить. Никакой взгляд не имеет никаких преимуществ перед другим.

Возможно, нет смысла говорить о законах физики и физических масштабах применительно к архитектуре программного обеспечения, но мы действительно учитываем некоторые физические ограничения. Скорость работы процессора и пропускная способность сети могут вынести суровый приговор производительности. Объем памяти и дискового пространства может ограничить амбиции любого программного кода. Программное обеспечение можно сравнить с такой материей, как мечты, но ему приходится работать в реальном, физическом мире.

В любви, дорогая, чудовищна только безграничность воли, безграничность желаний, несмотря на то, что силы наши ограничены, а осуществление мечты – в тисках возможности.

Вильям Шекспир¹

Физический мир – это мир, в котором мы живем, в котором находятся и действуют наши компании и экономика. Это дает нам другой подход к пониманию архитектуры программного обеспечения, позволяющий говорить и рассуждать не в терминах физических законов и понятий.

Архитектура отражает важные проектные решения по формированию системы, где важность определяется стоимостью изменений.

Гради Буч

Время, деньги, трудозатраты – вот еще одна система координат, помогающая нам различать большое и малое и отделять относящееся к архитектуре от всего остального. Она также помогает дать качественную оценку архитектуре – хорошая она или нет: хорошая архитектура отвечает

потребностям пользователей, разработчиков и владельцев не только сейчас, но и продолжит отвечать им в будущем.

Если вы думаете, что хорошая архитектура стоит дорого, попробуйте плохую архитектуру.

Брайан Фут и Джозеф Йодер

Типичные изменения, происходящие в процессе разработки системы, не должны быть дорогостоящими, сложными в реализации; они должны укладываться в график развития проекта и в рамки дневных или недельных заданий.

Это ведет нас прямиком к большой физической проблеме: путешествиям во времени. Как узнать, какие типичные изменения будут происходить, чтобы на основе этого знания принять важные решения? Как уменьшить трудозатраты и стоимость разработки без машины времени и гадания на кофейной гуще?

Архитектура – это набор верных решений, которые хотелось бы принять на ранних этапах работы над проектом, но которые не более вероятны, чем другие.

Ральф Джонсон

Анализ прошлого сложен; понимание настоящего в лучшем случае переменчиво; предсказание будущего нетривиально.

К цели ведет много путей.

На самом темном пути подстерегает мысль, что прочность и стабильность архитектуры зависят от строгости и жесткости. Если изменение оказывается слишком дорогостоящим, оно отвергается — его причины отменяются волевым решением. Архитектор имеет полную и безоговорочную власть, а архитектура превращается в антиутопию для разработчиков и постоянный источник недовольств.

От другого пути исходит сильный запах спекулятивной общности. Он полон догадок, бесчисленных параметров, могильников с «мертвым» кодом и на нем подкарауливает множество случайных сложностей, способных покачнуть бюджет, выделенный на обслуживание.

Но самый интересный — третий, чистый путь. Он учитывает природную гибкость программного обеспечения и стремится сохранить ее как основное свойство системы. Он учитывает, что мы оперируем неполными знаниями и, будучи людьми, неплохо приспособились к этому. Он играет больше на наших сильных сторонах, чем на слабостях. Мы создаем что-то и совершаем открытия. Мы задаем вопросы и проводим эксперименты. Хорошая архитектура основывается скорее на понимании движения к цели как непрерывного процесса исследований, а не на понимании самой цели как зафиксированного артефакта.

Архитектура — это гипотеза, которую требуется доказать реализацией и оценкой.

Том Гилб

Чтобы пойти по этому пути, нужно быть усердным и внимательным, нужно уметь думать и наблюдать, приобретать практические навыки и осваивать принципы. Сначала кажется, что это долгий путь, но в действительности все зависит от темпа вашей ходьбы.

Поспешай не торопясь.

Роберт С. Мартин

Получайте удовольствие от путешествия.

Кевлин Хенни май, 2017

От издательства

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

Мы будем рады узнать ваше мнение!

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

[1](#) Перевод Т. Гнедич. — Примеч. ред.

Вступление

Эта книга называется «Чистая архитектура». Смелое название. Кто-то посчитает его самонадеянным. Почему я решил написать эту книгу и выбрал такое название?

Свою первую строку кода я написал в 1964 году, в 12 лет. Сейчас на календаре 2016-й, то есть я пишу код уже больше полувека. За это время я кое-что узнал о структурировании программных систем, и мои знания, как мне кажется, многие посчитали бы ценными.

Я приобрел эти знания, создав множество систем, больших и маленьких. Я создавал маленькие встраиваемые системы и большие системы пакетной обработки, системы реального времени и веб-системы, консольные приложения, приложения с графическим интерфейсом, приложения управления процессами, игры, системы учета, телекоммуникационные системы, инструменты для проектирования, графические редакторы и многое, многое другое.

Я писал однопоточные и многопоточные приложения, приложения с несколькими тяжеловесными процессами и приложения с большим количеством легковесных процессов, многопроцессорные приложения, приложения баз данных, приложения для математических вычислений и вычислительной геометрии и многие, многие другие.

Я написал очень много приложений, я создал очень много систем. И благодаря накопленному опыту я пришел к потрясающему выводу:

Все архитектуры подчиняются одним и тем же правилам!

Потрясающему, потому что все системы, в создании которых я участвовал, радикально отличались друг от друга. Почему архитектуры таких разных систем подчиняются одним

и тем же правилам? Я пришел к выводу, что *правила создания архитектур не зависят от любых других переменных*.

Этот вывод кажется еще более потрясающим, если вспомнить, как изменились компьютеры за те же полвека. Я начинал программировать на машинах размером с холодильник, имевших процессоры с тактовой частотой в полмегагерца, 4 Кбайт оперативной памяти, 32 Кбайт дисковой памяти и интерфейс телетайпа со скоростью передачи данных 10 символов в секунду. Это вступление я написал в автобусе, путешествуя по Южной Африке. Я использовал MacBook, оснащенный процессором i7 с четырьмя ядрами, каждое из которых действует на тактовой частоте 2,8 ГГц, имеющий 16 Гбайт оперативной памяти, 1 Тбайт дисковой памяти (на устройстве SSD) и дисплей с матрицей 2880×1800, способный отображать высококачественное видео. Разница в вычислительной мощности умопомрачительная. Любой анализ покажет, что этот MacBook по меньшей мере в 10^{22} раз мощнее ранних компьютеров, на которых я начинал полвека назад.

Двадцать два порядка — очень большое число. Это число ангстремов от Земли до альфы Центавра. Это количество электронов в мелких монетах в вашем кармане или кошельке. И еще это число описывает, во сколько раз (как минимум) увеличилась вычислительная мощность компьютеров на моем веку.

А как влиял рост вычислительной мощности на программы, которые мне приходилось писать? Определенно они стали больше. Раньше я думал, что 2000 строк — это большая программа. В конце концов, такая программа занимала полную коробку перфокарт и весила 4,5 килограмма. Однако теперь программа считается по-настоящему большой, только если объем кода превысит 100 000 строк.

Программное обеспечение также стало значительно более производительным. Сегодня мы можем быстро выполнять такие вычисления, о которых и не мечтали в 1960-х. В произведениях *The Forbin Project*², *The Moon Is a Harsh Mistress*³ и *2001: A Space Odyssey*⁴ была сделана попытка изобразить наше текущее будущее, но их авторы серьезно промахнулись. Все они изображали огромные машины, обладающие чувствами. В действительности мы имеем невероятно маленькие машины, которые все же остаются машинами.

И еще одно важное сходство современного программного обеспечения и прошлого программного обеспечения: *и то и другое сделано из того же материала*. Оно состоит из инструкций `if`, инструкций присваивания и циклов `while`.

Да, вы можете возразить, заявив, что современные языки программирования намного лучше и поддерживают превосходящие парадигмы. В конце концов, мы программируем на Java, C# или Ruby и используем объектно-ориентированную парадигму. И все же программный код до сих пор состоит из последовательностей операций, условных инструкций и итераций, как в 1950-х и 1960-х годах.

Внимательно рассмотрев практику программирования компьютеров, вы заметите, что очень немногое изменилось за 50 лет. Языки стали немного лучше. Инструменты стали фантастически лучше. Но основные строительные блоки компьютерных программ остались прежними.

Если взять программиста из 1966 года, переместить ее⁵ в 2016-й, посадить за мой MacBook, запустить IntelliJ и показать ей Java, ей потребовались бы лишь сутки, чтобы оправиться от шока. А затем она смогла бы писать современные программы. Язык Java не сильно отличается от С или даже от Fortran.

И если вас переместить обратно в 1966-й год и показать, как писать и править код на PDP-8, пробивая перфокарты на телетайпе, поддерживающем скорость 10 символов в секунду, вам также могут понадобиться сутки, чтобы оправиться от разочарования. Но потом и вы смогли бы писать код. Сам код мало бы изменился при этом.

В этом весь секрет: неизменность принципов программирования — вот причина общности правил построения программных архитектур для систем самых разных типов. Правила определяют последовательность и порядок компоновки программ из строительных блоков. И поскольку сами строительные блоки являются универсальными и не изменились с течением времени, правила их компоновки также являются универсальными и постоянными.

Начинающие программисты могут подумать, что все это чепуха, что в наши дни все совсем иначе и лучше, что правила прошлого остались в прошлом. Если они действительно так думают, они, к сожалению, сильно ошибаются. Правила не изменились. Несмотря на появление новых языков, фреймворков и парадигм, правила остались теми же, как во времена, когда в 1946-м Аллан Тьюринг написал первый программный код.

Изменилось только одно: тогда, в прошлом, мы не знали этих правил. Поэтому нарушили их снова и снова. Теперь, с полувековым опытом за плечами, мы знаем и понимаем правила.

Именно об этих правилах — не стареющих и не изменяющихся — рассказывает эта книга.

[2](#) Фильм, вышедший в США в 1970 году, в нашей стране известный под названием «Колосс: Проект Форбина». — Примеч. пер.

[3](#) «Луна жестко стелет», роман Роберта Хайнлайна. — Примеч. пер.

[4](#) Фильм, вышедший в 1968 году, в нашей стране известный под названием «2001 год: Космическая одиссея». — *Примеч. пер.*

[5](#) Именно «ее», потому что в те годы программистами были в основном женщины.

Благодарности

Ниже перечислены все (не в каком-то определенном порядке), кто сыграл важную роль в создании этой книги:

- Крис Гузиковски (Chris Guzikowski)
- Крис Зан (Chris Zahn)
- Мэтт Хойзер (Matt Heuser)
- Джефф Оверби (Jeff Overbey)
- Мика Мартин (Micah Martin)
- Джастин Мартин (Justin Martin)
- Карл Хикман (Carl Hickman)
- Джеймс Греннинг (James Grenning)
- Симон Браун (Simon Brown)
- Кевлин Хенни (Kevlin Henney)
- Джайсон Горман (Jason Gorman)
- Дуг Бредбери (Doug Bradbury)
- Колин Джонс (Colin Jones)
- Гради Буч (Grady Booch)
- Кент Бек (Kent Beck)
- Мартин Фаулер (Martin Fowler)
- Алистер Кокберн (Alistair Cockburn)
- Джеймс О. Коплиен (James O. Coplien)
- Тим Конрад (Tim Conrad)
- Ричард Ллойд (Richard Lloyd)
- Кен Финдер (Ken Finder)
- Крис Ивер (Kris Iyer (CK))
- Майк Карев (Mike Carew)
- Джерри Фицпатрик (Jerry Fitzpatrick)

Джим Ньюкирк (Jim Newkirk)

Эд Телен (Ed Thelen)

Джо Мейбл (Joe Mabel)

Билл Дегнан (Bill Degnan)

И многие другие.

Когда я просматривал книгу в окончательном варианте и перечитывал главу о кричащей архитектуре, передо мной стоял образ улыбающегося Джима Уэрича (Jim Weirich) и в ушах слышался его звонкий смех. Удачи тебе, Джим!

Об авторе



Роберт С. Мартин (Robert C. Martin), известный также как «дядюшка Боб» (Uncle Bob),

профессионально занимается программированием с 1970 года. Сооснователь компании *cleancoders.com*, предлагающей видеоуроки для разработчиков программного обеспечения, и основатель компании *Uncle Bob Consulting LLC*, оказывающей консультационные услуги и услуги по обучению персонала крупным корпорациям. Служит мастером в консалтинговой фирме *8th Light, Inc.* в городе Чикаго. Опубликовал десятки статей в специализированных журналах и регулярно выступает на международных конференциях и демонстрациях. Три года был главным редактором журнала *C++ Report* и первым председателем *Agile Alliance*.

Мартин написал и выступил редактором множества книг, включая *The Clean Coder*⁶, *Clean Code*⁷, *UML for Java Programmers*, *Agile Software Development*⁸, *Extreme Programming in Practice*, *More C++ Gems*, *Pattern Languages of Program Design 3* и *Designing Object Oriented C++ Applications Using the Booch Method*.

[6](#) Роберт Мартин. Идеальный программист. Как стать профессионалом разработки ПО. СПб.: Питер, 2016. — Примеч. пер.

[7](#) Роберт Мартин. Чистый код: создание, анализ и рефакторинг. СПб.: Питер, 2013. — Примеч. пер.

[8](#) Роберт Мартин. Быстрая разработка программ. Принципы, примеры, практика. М.: Вильямс, 2004. — Примеч. пер.

I. Введение

Чтобы написать действующую программу, не нужно много знать и уметь. Дети в школе делают это постоянно. Юноши и девушки в колледжах начинают свой бизнес, вырастающий до стоимости в миллиарды долларов, написав несколько строк кода на PHP или Ruby. Начинающие программисты в офисах по всему миру перелопачивают горы требований, хранящихся в гигантских баг-трекерах, и вносят исправления, чтобы заставить свои системы «работать». Возможно, они пишут не самый лучший код, но он работает. Он работает, потому что заставить что-то работать — один раз — не очень сложно.

Создать программу, которая будет работать правильно, — совсем другое дело. Написать правильную программу сложно. Для этого необходимы знания и умения, которые молодые программисты еще не успели приобрести. А чтобы приобрести их, требуется мыслить и анализировать, на что у многих программистов просто нет времени. Это требует такой самодисциплины и организованности, которые не снились большинству программистов. А для этого нужно испытывать страсть к профессии и желание стать профессионалом.

Но когда создается правильный программный код, происходит что-то необычное: вам не требуются толпы программистов для поддержки его работоспособности. Вам не нужна объемная документация с требованиями и гигантские баг-трекеры. Вам не нужны огромные опенспейсы, работающие круглые сутки без выходных.

Правильный программный код не требует больших трудозатрат на свое создание и сопровождение. Изменения вносятся легко и быстро. Ошибки немногочисленны.

Трудозатраты минимальны, а функциональность и гибкость — максимальны.

Да, такое представление кажется утопическим. Но я видел это воочию. Я участвовал в проектах, где дизайн и архитектура системы упрощали их создание и сопровождение. У меня есть опыт работы в проектах, потребовавших меньшего числа участников, чем предполагалось. Мне довелось работать над системами, в которых ошибки встречались удивительно редко. Я видел, какой невероятный эффект оказывает хорошая архитектура на систему, проект и коллектив разработчиков. Я был на земле обетованной.

Но я не прошу верить мне на слово. Оглянитесь на свой опыт. Случалось ли у вас что-то противоположное? Доводилось ли вам работать над системами с настолько запутанными и тесными связями, что любое изменение, независимо от сложности, требовало недель труда и было сопряжено с огромным риском? Испытывали ли вы сопротивление плохого кода и неудачного дизайна? Приходилось ли вам видеть, как дизайн системы оказывал отрицательное влияние на моральный дух команды, доверие клиентов и терпение руководителей? Доводилось ли вам оказываться в ситуации, когда отделы, подразделения и целые компании становились жертвами неудачной архитектуры их программного обеспечения? Были ли в аду программирования?

Я был, и многие из нас были там. В нашей среде чаще встречается опыт борьбы с плохим дизайном, чем получение удовольствия от воплощения хорошо продуманной архитектуры.

1. Что такое дизайн и архитектура?



За долгие годы вокруг понятий «дизайн» и «архитектура» накопилось много путаницы. Что такое дизайн? Что такое архитектура? Чем они различаются?

Одна из целей этой книги — устраниТЬ весь этот беспорядок и определить раз и навсегда, что такое дизайн и архитектура. Прежде всего, я утверждаю, что между этими понятиями нет никакой разницы. *Вообще никакой.*

Слово «архитектура» часто используется в контексте общих рассуждений, когда не затрагиваются низкоуровневые детали, а слово «дизайн» обычно подразумевает организацию и решения на более низком уровне. Но такое разделение

бессмысленно, когда речь идет о том, что делает настоящий архитектор.

Возьмем для примера архитектора, спроектировавшего мой новый дом. Этот дом имеет архитектуру? Конечно! А в чем она выражается? Ну... это форма дома, внешний вид, уступы, а также расположение комнат и организация пространства внутри. Но когда я рассматривал чертежи, созданные архитектором, я увидел на них массу деталей. Я увидел расположение всех розеток, выключателей и светильников. Я увидел, какие выключатели будут управлять теми или иными светильниками. Я увидел, где будет находиться узел отопления, а также местоположение и размеры водонагревательного котла и насоса. Я увидел подробное описание, как должны конструироваться стены, крыша и фундамент.

Проще говоря, я увидел все мелкие детали, поддерживающие все высокоуровневые решения. Я также увидел, что низкоуровневые детали и высокоуровневые решения вместе составляют дизайн дома.

То же относится к архитектуре программного обеспечения. Низкоуровневые детали и высокоуровневая структура являются частями одного целого. Они образуют сплошное полотно, определяющее форму системы. Одно без другого невозможно; нет никакой четкой линии, которая разделяла бы их. Есть просто совокупность решений разного уровня детализации.

Цель?

В чем состоит цель таких решений, цель хорошего дизайна программного обеспечения? Главная цель — не что иное, как мое утопическое описание:

Цель архитектуры программного обеспечения – уменьшить человеческие трудозатраты на создание и сопровождение системы.

Мерой качества дизайна может служить простая мера трудозатрат, необходимых для удовлетворения потребностей клиента. Если трудозатраты невелики и остаются небольшими в течение эксплуатации системы, система имеет хороший дизайн. Если трудозатраты увеличиваются с выходом каждой новой версии, система имеет плохой дизайн. Вот так все просто.

Пример из практики

В качестве примера рассмотрим результаты исследований из практики. Они основаны на реальных данных, предоставленных реальной компанией, пожелавшей не разглашать своего названия.

Сначала рассмотрим график роста численности инженерно-технического персонала. Вы наверняка согласитесь, что тенденция впечатляет. Рост численности, как показано на рис. 1.1, должен служить признаком успешного развития компании!

Жизненный цикл известного программного продукта

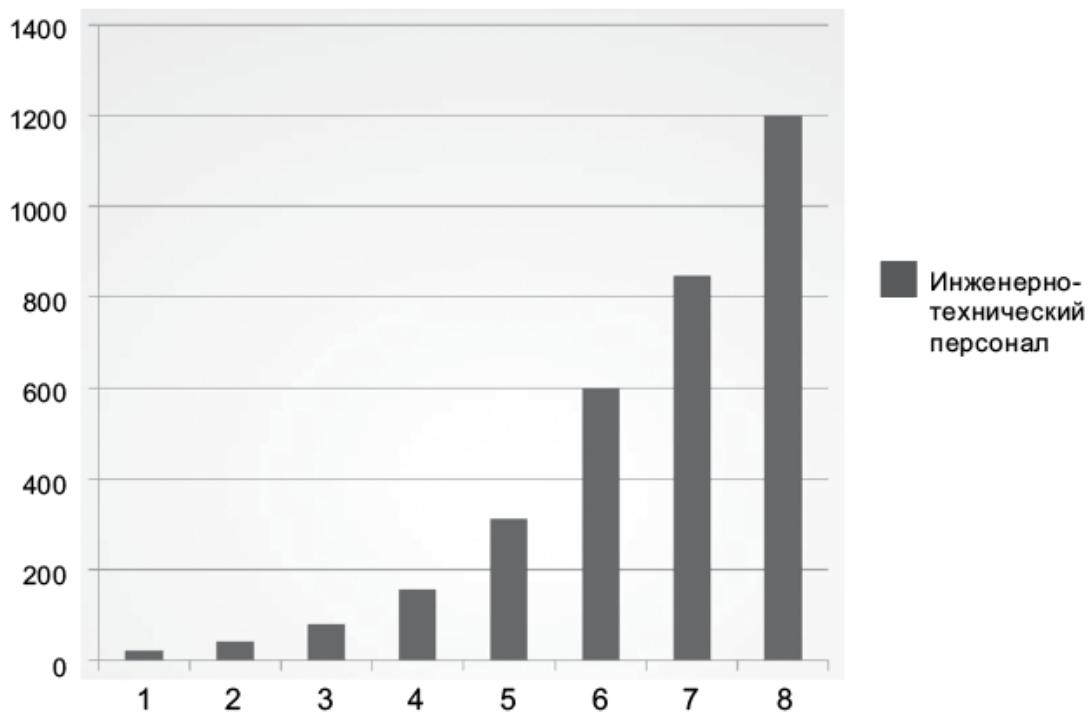


Рис. 1.1. Рост численности инженерно-технического персонала. Воспроизводится с разрешения автора презентации Джейсона Гормана (Jason Gorman)

Теперь взгляните на график продуктивности компании за тот же период, измеряемой в количестве строк кода (рис. 1.2).

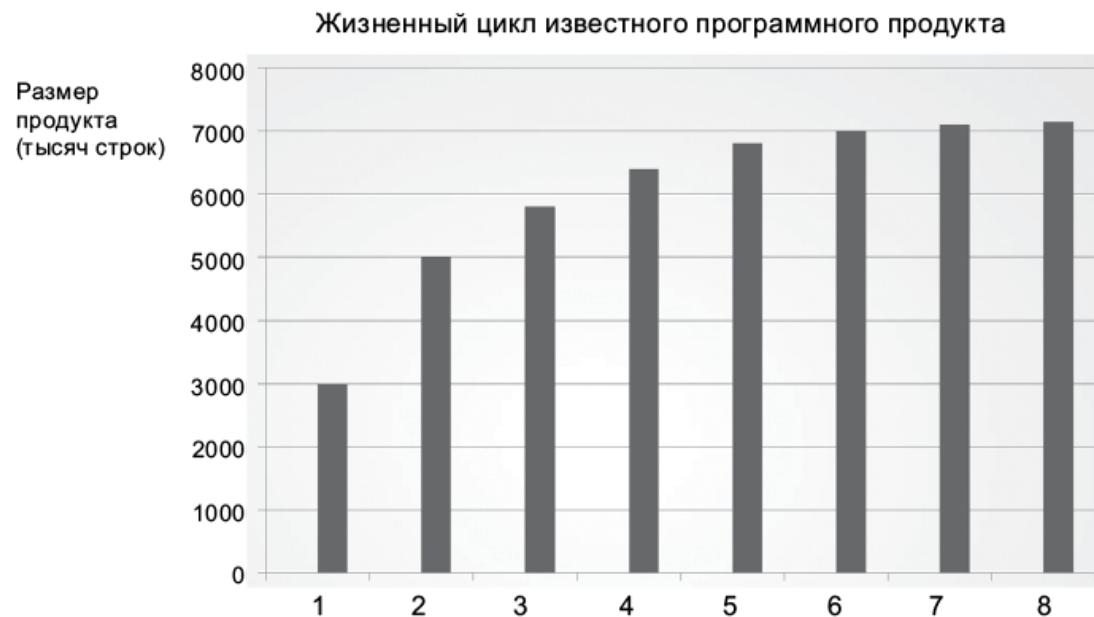


Рис. 1.2. Продуктивность за тот же период

Очевидно, что здесь что-то не так. Даже при том, что выпуск каждой версии поддерживается все большим количеством разработчиков, похоже, что количество строк кода приближается к своему пределу.

А теперь взгляните на по-настоящему удручающий график: на рис. 1.3 показан рост стоимости строки кода с течением времени.

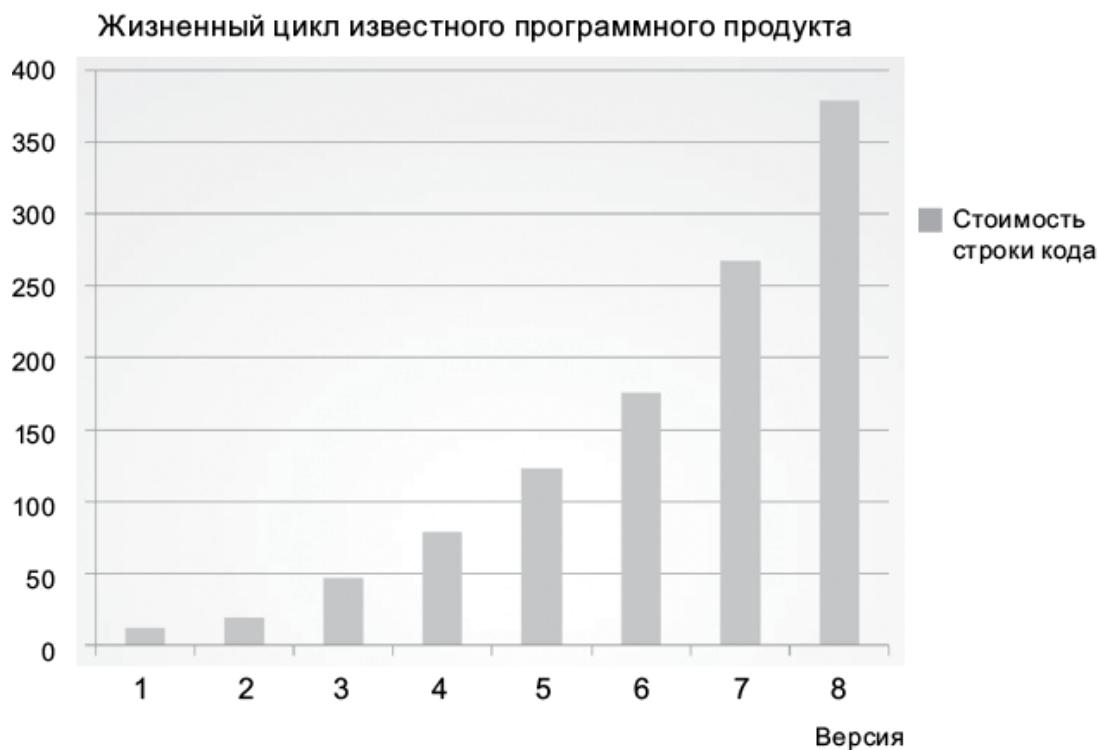


Рис. 1.3. Изменение стоимости строки кода с течением времени

Эта тенденция говорит о нежизнеспособности. Какой бы рентабельной ни была компания в настоящее время, растущие накладные расходы поглотят прибыль и приведут ее к застою, если не к краху.

Чем обусловлено такое значительное изменение продуктивности? Почему строка кода в версии 8 продукта стоит в 40 раз дороже, чем в версии 1?

Причины неприятностей

Причины неприятностей перед вашими глазами. Когда системы создаются второпях, когда увеличение штата программистов — единственный способ продолжать выпускать новые версии и когда чистоте кода или дизайну уделяется минимум внимания или не уделяется вообще, можно даже не сомневаться, что такая тенденция рано или поздно приведет к краху.

На рис. 1.4 показано, как выглядит эта тенденция применительно к продуктивности разработчиков. Сначала разработчики показывают продуктивность, близкую к 100%, но с выходом каждой новой версии она падает. Начиная с четвертой версии, как нетрудно заметить, их продуктивность приближается к нижнему пределу — к нулю.



Рис. 1.4. Изменение продуктивности с выпуском новых версий

С точки зрения разработчиков, такая ситуация выглядит очень удручающее, потому все они продолжают трудиться с

полной отдачей сил. Никто не отлынивает от работы.

И все же, несмотря на сверхурочный труд и самоотверженность, они просто не могут произвести больше. Все их усилия теперь направлены не на реализацию новых функций, а на борьбу с беспорядком. Большую часть времени они заняты тем, что переносят беспорядок из одного места в другое, раз за разом, чтобы получить возможность добавить еще одну мелочь.

Точка зрения руководства

Если вы думаете, что ситуация скверная, взгляните на нее с точки зрения руководства! На рис. 1.5 изображен график изменения месячного фонда оплаты труда разработчиков за тот же период.

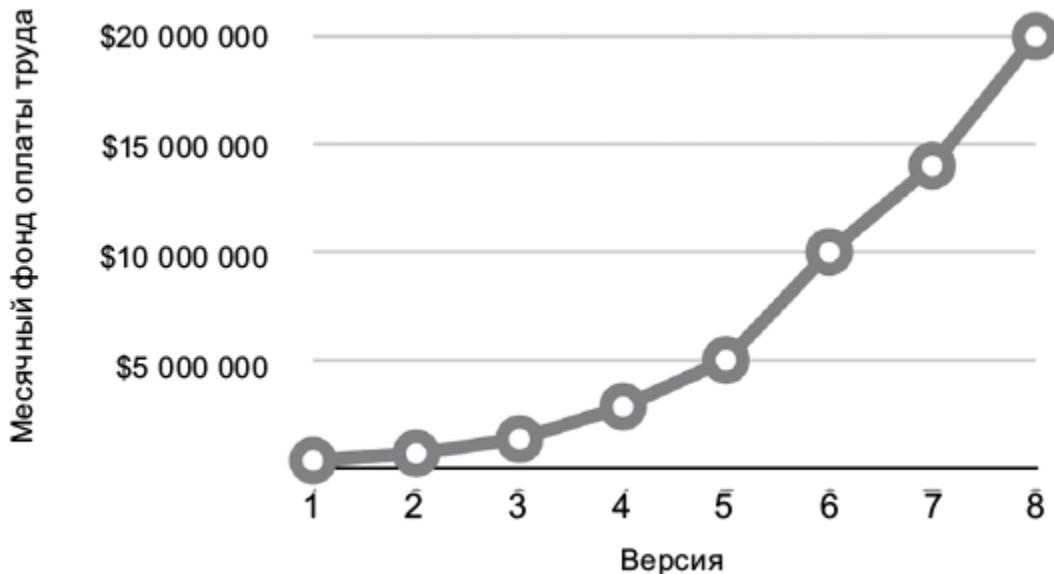


Рис. 1.5. Изменение фонда оплаты труда разработчиков с выпуском новых версий

Когда была выпущена версия 1, месячный фонд оплаты труда составлял несколько сотен тысяч долларов. К выпуску второй версии фонд увеличился еще на несколько сотен тысяч. К выпуску восьмой версии месячный фонд оплаты труда

составил 20 миллионов долларов и тенденция к увеличению сохраняется.

Этот график выглядит не просто скверно, он пугает. Очевидно, что происходит что-то ужасное. Одна надежда, что рост доходов опережает рост затрат, а значит, оправдывает расходы. Но с любой точки зрения эта кривая вызывает беспокойство.

А теперь сравните кривую на рис. 1.5 с графиком роста количества строк кода от версии к версии на рис. 1.2. Сначала, всего за несколько сотен тысяч долларов в месяц, удалось реализовать огромный объем функциональности, а за 20 миллионов в последнюю версию почти ничего не было добавлено! Любой менеджер, взглянув на эти два графика, придет к выводу, что необходимо что-то предпринять, чтобы предотвратить крах.

Но что можно предпринять? Что пошло не так? Что вызвало такое невероятное снижение продуктивности? Что могут сделать руководители, кроме как топнуть ногой и излить свой гнев на разработчиков?

Что не так?

Примерно 2600 лет тому назад Эзоп сочинил басню о Зайце и Черепахе. Мораль той басни можно выразить по-разному:

- «медленный и постоянный побеждает в гонке»;
- «в гонке не всегда побеждает быстрейший, а в битве — сильнейший»;
- «чем больше спешишь, тем меньше успеваешь».

Причина подчеркивает глупость самонадеянности. Заяц был настолько уверен в своей скорости, что не отнесся всерьез к состязанию, решил вздремнуть и проспал, когда Черепаха пересекла финишную черту.

Современные разработчики также участвуют в похожей гонке и проявляют похожую самонадеянность. О нет, они не спят, нет. Многие современные разработчики работают как проклятые. Но часть их мозга *действительно* спит — та часть, которая знает, что хороший, чистый, хорошо проработанный код играет немаловажную роль.

Эти разработчики верят в известную ложь: «Мы сможем навести порядок потом, нам бы только выйти на рынок!» В результате порядок так и не наводится, потому что давление конкуренции на рынке никогда не ослабевает. Выход на рынок означает, что теперь у вас на хвосте висят конкуренты и вы должны стремиться оставаться впереди них и бежать вперед изо всех сил.

Поэтому разработчики никогда не переключают режим работы. Они не могут вернуться и навести порядок, потому что должны реализовать следующую новую функцию, а потом еще одну, и еще, и еще. В результате беспорядок нарастает, а продуктивность стремится к своему пределу около нуля.

Так же как Заяц был излишне уверен в своей скорости, многие разработчики излишне уверены в своей способности оставаться продуктивными. Но ползучий беспорядок в коде, иссушающий их продуктивность, никогда не спит и не никогда не существует. Если впустить его, он уменьшит производительность до нуля за считанные месяцы.

Самая большая ложь, в которую верят многие разработчики, — что грязный код поможет им быстро выйти на рынок, но в действительности он затормозит их движение в долгосрочной перспективе. Разработчики, уверовавшие в эту ложь,

проявляют самонадеянность Зайца, полагая, что в будущем смогут перейти от создания беспорядка к наведению порядка, но они допускают простую ошибку. Дело в том, что *создание беспорядка всегда оказывается медленнее, чем неуклонное соблюдение чистоты*, независимо от выбранного масштаба времени.

Рассмотрим на рис. 1.6 результаты показательного эксперимента, проводившегося Джейсоном Горманом в течение шести дней. Каждый день он писал от начала до конца простую программу преобразования целых чисел из десятичной системы счисления в римскую. Работа считалась законченной, когда программа успешно проходила предопределенный комплект приемочных тестов. Каждый день на решение поставленной задачи затрачивалось чуть меньше 30 минут. В первый, второй и третий дни Джейсон использовал

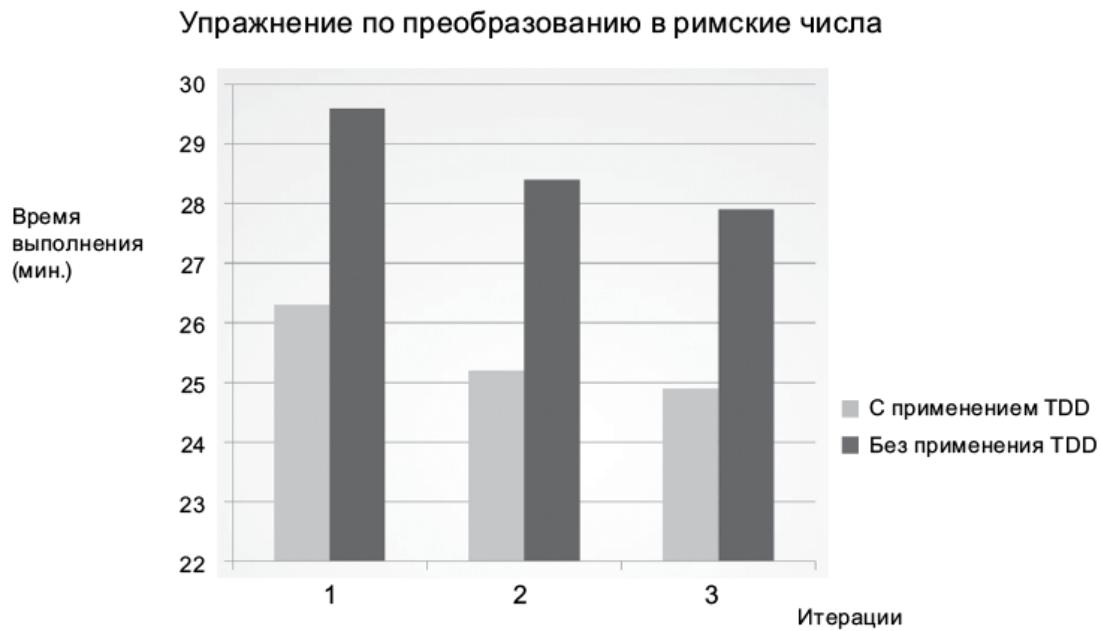


Рис. 1.6. Время на выполнение итерации с использованием и без использования методики TDD

хорошо известную методику разработки через тестирование (Test-Driven Development; TDD). В остальные дни он писал код, не ограничивая себя рамками этой методики.

Прежде всего обратите внимание на кривую обучения, заметную на рис. 1.6. Каждый раз на решение задачи затрачивалось меньше времени. Отметьте также, что в дни, когда применялась методика TDD, упражнение выполнялось примерно на 10% быстрее, чем в дни без применения TDD, и что даже худший результат, полученный с TDD, оказался лучше самого лучшего результата, полученного без TDD.

Кто-то, взглянув на этот результат, может посчитать его удивительным. Но для тех, кто не поддался обману самонадеянности Зайца, результат будет вполне ожидаемым, потому что они знают простую истину разработки программного обеспечения:

Поспешай не торопясь.

И она же является ответом на дилемму, стоящую перед руководством. Единственный способ обратить вспять снижение продуктивности и увеличение стоимости — заставить разработчиков перестать думать как самонадеянный Заяц и начать нести ответственность за беспорядок, который они учинили.

Разработчики могут подумать, что проблему можно исправить, только начав все с самого начала и перепроектировав всю систему целиком, — но это в них говорит все тот же Заяц. Та же самонадеянность, которая прежде уже привела к беспорядку, теперь снова говорит им, что они смогут построить лучшую систему, если только вновь вступят в гонку. Однако в действительности все не так радужно:

Самонадеянность, управляющая перепроектированием, приведет к тому же беспорядку, что и прежде.

Заключение

Любой организации, занимающейся разработкой, лучше всего избегать самонадеянных решений и с самого начала со всей серьезностью отнестись к качеству архитектуры ее продукта.

Серьезное отношение к архитектуре программного обеспечения подразумевает знание о том, что такая хорошая архитектура. Чтобы создать систему, дизайн и архитектура которой способствуют уменьшению трудозатрат и увеличению производительности, нужно знать, какие элементы архитектуры ведут к этому.

Именно об этом рассказывается в данной книге. В ней рассказывается, как выглядит добротная, чистая архитектура и дизайн, чтобы разработчики могли создавать системы, способные приносить прибыль долгое время.

2. История о двух ценностях



Всякая программная система имеет две разные ценности: поведение и структуру. Разработчики отвечают за высокий уровень обеих этих ценностей. Но, к сожалению, они часто сосредоточиваются на чем-то одном, забывая про другое. Хуже того, они нередко сосредоточиваются на меньшей из двух ценностей, что в конечном итоге обесценивает систему.

Поведение

Первая ценность программного обеспечения — его поведение. Программистов нанимают на работу, чтобы они заставили

компьютеры экономить деньги или приносить прибыль заинтересованной стороне. Для этого мы помогаем заинтересованным сторонам разработать функциональную спецификацию или документ с требованиями. Затем пишем код, заставляющий компьютеры заинтересованных сторон удовлетворять этим требованиям.

Когда компьютер нарушает требования, программисты вынимают свои отладчики и исправляют проблему.

Многие программисты полагают, что этим их работа ограничивается. Они уверены, что их задача — заставлять компьютеры соответствовать требованиям и исправлять ошибки. Они жестоко ошибаются.

Архитектура

Вторая ценность программного обеспечения заключена в самом названии «программное обеспечение». Слово «обеспечение» означает «продукт»; а слово «программное»... Как раз в нем и заключается вторая ценность.

Идея программного обеспечения состоит в том, чтобы дать простую возможность изменять поведение компьютеров. Поведение компьютеров в некоторых пределах можно также изменять посредством аппаратного обеспечения, но этот путь намного сложнее.

Для достижения этой цели программное обеспечение должно быть податливым — то есть должна быть возможность легко изменить его. Когда заинтересованные стороны меняют свое мнение о некоторой особенности, приведение ее в соответствие с пожеланиями заинтересованной стороны должно быть простой задачей. Сложность в таких случаях должна быть пропорциональна лишь масштабу изменения, но никак не его форме.

Именно эта разница между масштабом и формой часто является причиной роста стоимости разработки программного обеспечения. Именно по этой причине стоимость растет пропорционально объему требуемых изменений. Именно поэтому стоимость разработки в первый год существенно ниже, чем во второй, а во второй год ниже, чем в третий.

С точки зрения заинтересованных сторон они просто формируют поток изменений примерно одинакового масштаба. С точки зрения разработчиков, заинтересованные стороны формируют поток фрагментов, которые они должны встраивать в мозаику со все возрастающей сложностью. Каждый новый запрос сложнее предыдущего, потому что форма системы не соответствует форме запроса.

Я использовал здесь слово «форма» не в традиционном его понимании, но, как мне кажется, такая метафора вполне уместна. У разработчиков программного обеспечения часто складывается ощущение, что их заставляют затыкать круглые отверстия квадратными пробками.

Проблема, конечно же, кроется в архитектуре системы. Чем чаще архитектура отдает предпочтение какой-то одной форме, тем выше вероятность, что встраивание новых особенностей в эту структуру будет даваться все сложнее и сложнее. Поэтому архитектуры должны быть максимально независимыми от формы.

Наибольшая ценность

Функциональность или архитектура? Что более ценно? Что важнее — правильная работа системы или простота ее изменения?

Если задать этот вопрос руководителю предприятия, он наверняка ответит, что важнее правильная работа.

Разработчики часто соглашаются с этим мнением. Но оно ошибочно. Я могу доказать ошибочность этого взгляда простым логическим инструментом исследования экстремумов.

- *Если правильно работающая программа не допускает возможности ее изменения, она перестанет работать правильно, когда изменятся требования, и вы не сможете заставить ее работать правильно. То есть программа станет бесполезной.*
- *Если программа работает неправильно, но легко поддается изменению, вы сможете заставить работать ее правильно и поддерживать ее работоспособность по мере изменения требований. То есть программа постоянно будет оставаться полезной.*

Эти аргументы могут показаться вам неубедительными. В конце концов, нет таких программ, которые нельзя изменить. Однако есть системы, изменить которые практически невозможно, потому что стоимость изменений превысит получаемые выгоды. Многие системы достигают такого состояния в некоторых своих особенностях или конфигурациях.

Если спросить руководителя, хотел бы он иметь возможность вносить изменения, он ответит, что безусловно хотел бы, но тут же может уточнить, что работоспособность в данный момент для него важнее гибкости в будущем. Однако если руководитель потребует внести изменения, а ваши затраты на эти изменения окажутся несоизмеримо высокими, он почти наверняка будет негодовать оттого, что вы довели систему до такого состояния, когда стоимость изменений превышает разумные пределы.

Матрица Эйзенхауэра

Рассмотрим матрицу президента Дуайта Дэвида Эйзенхауэра для определения приоритета между важностью и срочностью (рис. 2.1). Об этой матрице Эйзенхаэр говорил так:

У меня есть два вида дел, срочные и важные. Срочные дела, как правило, не самые важные, а важные – не самые срочные⁹.



Рис. 2.1. Матрица Эйзенхауэра

Это старое изречение несет много истины. Срочное действительно редко бывает важным, а важное – срочным.

Первая ценность программного обеспечения – поведение – это нечто срочное, но не всегда важное.

Вторая ценность – архитектура – нечто важное, но не всегда срочное.

Конечно, имеются также задачи важные и срочные одновременно и задачи не важные и не срочные. Все эти четыре вида задач можно расставить по приоритетам.

1. Срочные и важные.
2. Не срочные и важные.
3. Срочные и не важные.

4. Не срочные и не важные.

Обратите внимание, что архитектура кода — важная задача — оказывается в двух верхних позициях в этом списке, тогда как поведение кода занимает первую и третью позиции.

Руководители и разработчики часто допускают ошибку, поднимая пункт 3 до уровня пункта 1. Иными словами, они неправильно отделяют срочные и не важные задачи от задач, которые по-настоящему являются срочными и важными. Эта ошибочность суждений приводит к игнорированию важности архитектуры системы и уделению чрезмерного внимания не важному поведению.

Разработчики программного обеспечения оказываются перед проблемой, обусловленной неспособностью руководителей оценить важность архитектуры. Но именно для ее решения они и были наняты. Поэтому разработчики должны всякий раз подчеркивать приоритет важности архитектуры перед срочностью поведения.

Битва за архитектуру

Эта обязанность означает постоянную готовность к битве — возможно, в данном случае лучше использовать слово «борьба». Честно говоря, подобная ситуация распространена практически повсеместно. Команда разработчиков должна бороться за то, что, по их мнению, лучше для компании, так же как команда управленцев, команда маркетинга, команда продаж и команда эксплуатации. *Это всегда борьба.*

Эффективные команды разработчиков часто выходят победителями в этой борьбе. Они открыто и на равных вступают в конфликт со всеми другими заинтересованными сторонами. Помните: как разработчик программного

обеспечения вы тоже являетесь заинтересованной стороной. У вас есть свой интерес в программном обеспечении, который вы должны защищать. Это часть вашей роли и ваших обязанностей. И одна из основных причин, почему вас наняли.

Важность этой задачи удваивается, если вы выступаете в роли архитектора программного обеспечения. Архитекторы, в силу своих профессиональных обязанностей, больше сосредоточены на структуре системы, чем на конкретных ее особенностях и функциях. Архитекторы создают архитектуру, помогающую быстрее и проще создавать эти особенности и функции, изменять их и дополнять.

Просто помните, что если поместить архитектуру на последнее место, разработка системы будет обходиться все дороже, и в конце концов внесение изменений в такую систему или в отдельные ее части станет практически невозможным. Если это случилось, значит, команда разработчиков сражалась недостаточно стойко за то, что они считали необходимым.

⁹ Из речи, произнесенной в Северо-Западном университете в 1954 году.

II. Начальные основы: парадигмы программирования

Архитектура программного обеспечения начинается с кода, поэтому начнем обсуждение архитектуры с рассказа о самом первом программном коде.

Основы программирования заложил Алан Тьюринг в 1938 году. Он не первый, кто придумал программируемую машину, но он первым понял, что программы — это всего лишь данные. К 1945 году Тьюринг уже писал настоящие программы для настоящих компьютеров, используя код, который мы смогли бы прочитать (приложив определенные усилия). В своих программах он использовал циклы, конструкции ветвления, операторы присваивания, подпрограммы, стеки и другие знакомые нам структуры. Тьюринг использовал двоичный язык.

С тех пор в программировании произошло несколько революций. Одна из самых известных — революция языков. Во-первых, в конце 1940-х появились ассемблеры. Эти «языки» освободили программистов от тяжкого бремени трансляции их программ в двоичный код. В 1951 году Грейс Хоппер изобрела первый компилятор A0. Именно она фактически ввела термин *компилятор*. В 1953 году был изобретен язык Fortran (через год после моего рождения). Затем последовал непрерывный поток новых языков программирования: COBOL, PL/1, SNOBOL, C, Pascal, C++, Java и так до бесконечности.

Другая, еще более важная, как мне кажется, революция произошла в *парадигмах* программирования. Парадигма — это способ программирования, не зависящий от конкретного языка. Парадигма определяет, какие структуры использовать и когда их использовать. До настоящего времени было

придумано три такие парадигмы. По причинам, которые мы обсудим далее, едва ли стоит ожидать каких-то других, новых парадигм.

3. Обзор парадигм



В этой главе дается общий обзор следующих трех парадигм: структурное программирование, объектно-ориентированное программирование и функциональное программирование.

Структурное программирование

Первой, получившей всеобщее признание (но не первой из придуманных), была парадигма структурного программирования, предложенная Эдсгером Вибе Дейкстрой в 1968 году. Дейкстра показал, что безудержное использование переходов (инструкций `goto`) вредно для структуры программы. Как будет описано в последующих главах, он

предложил заменить переходы более понятными конструкциями `if/then/else` и `do/while/until`.

Подводя итог, можно сказать, что:

Структурное программирование накладывает ограничение на прямую передачу управления.

Объектно-ориентированное программирование

Второй парадигмой, получившей широкое распространение, стала парадигма, в действительности появившаяся двумя годами ранее, в 1966-м, и предложенная Оле-Йоханом Далем и Кристеном Нюгором. Эти два программиста заметили, что в языке ALGOL есть возможность переместить кадр стека вызова функции в динамическую память (кучу), благодаря чему локальные переменные, объявленные внутри функции, могут сохраняться после выхода из нее. В результате функция превращалась в конструктор класса, локальные переменные — в переменные экземпляра, а вложенные функции — в методы. Это привело к открытию полиморфизма через строгое использование указателей на функции.

Подводя итог, можно сказать, что:

Объектно-ориентированное программирование накладывает ограничение на косвенную передачу управления.

Функциональное программирование

Третьей парадигмой, начавшей распространяться относительно недавно, является самая первая из придуманных. Фактически изобретение этой парадигмы предшествовало появлению самой идеи программирования. Парадигма функционального программирования является

прямым результатом работы Алонзо Чёрча, который в 1936 году изобрел лямбда-исчисление (или λ -исчисление), исследуя ту же математическую задачу, которая примерно в то же время занимала Алана Тьюринга. Его λ -исчисление легло в основу языка LISP, изобретенного в 1958 году Джоном Маккарти. Основополагающим понятием λ -исчисления является неизменяемость — то есть невозможность изменения значений символов. Фактически это означает, что функциональный язык не имеет инструкции присваивания. В действительности большинство функциональных языков обладает некоторыми средствами, позволяющими изменять значение переменной, но в очень ограниченных случаях.

Подводя итог, можно сказать, что:

Функциональное программирование накладывает ограничение на присваивание.

Пища для ума

Обратите внимание на шаблон, который я преднамеренно ввел в представление этих трех парадигм программирования: каждая отнимает у программиста какие-то возможности. Ни одна не добавляет новых возможностей. Каждая накладывает какие-то дополнительные ограничения, *отрицательные* по своей сути. Парадигмы говорят нам не столько *что делать*, сколько *чего нельзя делать*.

Если взглянуть под другим углом, можно заметить, что каждая парадигма что-то отнимает у нас. Три парадигмы вместе отнимают у нас инструкции *goto*, указатели на функции и оператор присваивания. Есть ли у нас еще что-то, что можно отнять?

Вероятно, нет. Скорее всего, эти три парадигмы станут единственными, которые мы увидим, — по крайней мере

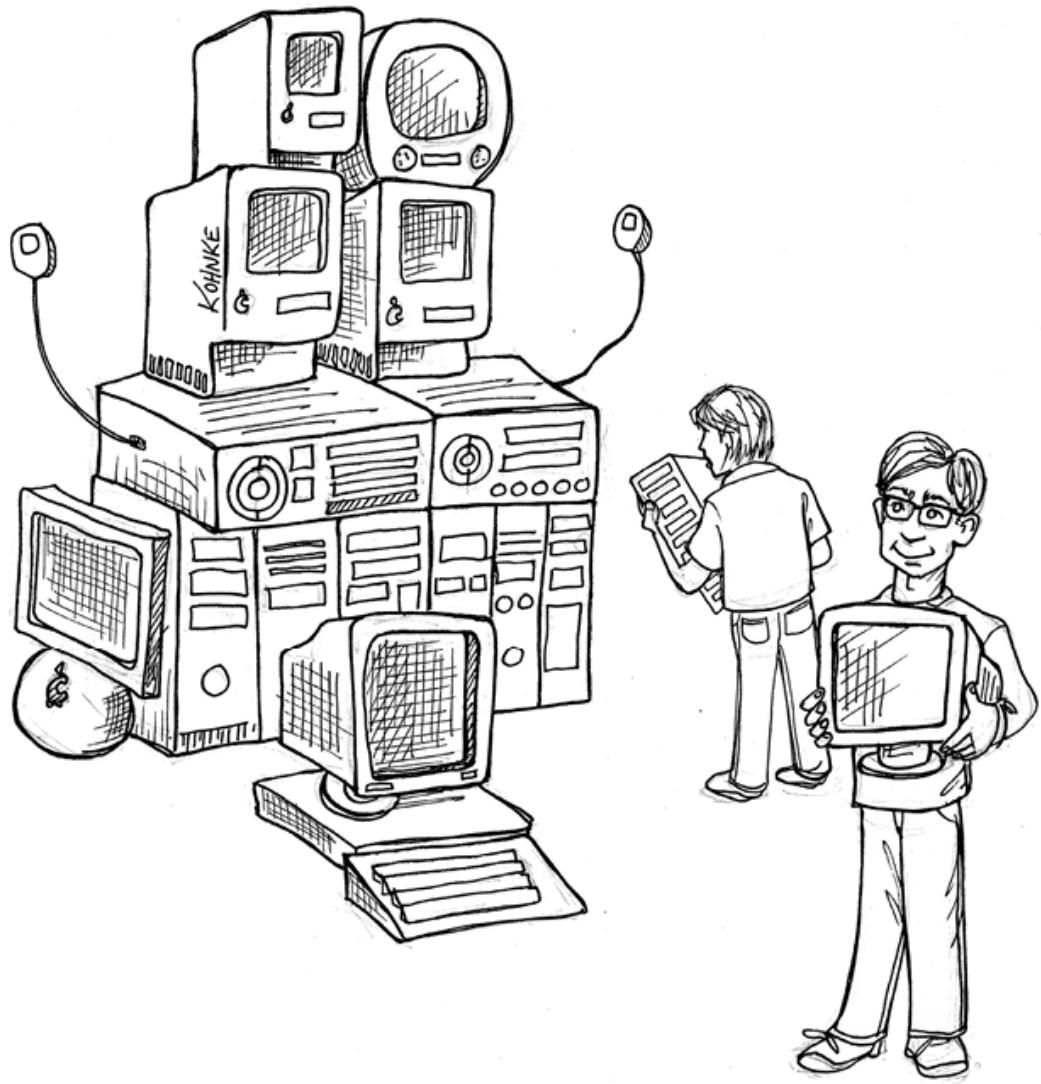
единственными, что-то отнимающими у нас. Доказательством отсутствия новых парадигм может служить тот факт, что все три известные парадигмы были открыты в течение десяти лет, между 1958 и 1968 годами. За многие последующие десятилетия не появилось ни одной новой парадигмы.

Заключение

Какое отношение к архитектуре имеет эта поучительная история о парадигмах? Самое непосредственное. Мы используем полиморфизм как механизм преодоления архитектурных границ, мы используем функциональное программирование для наложения ограничений на местоположение данных и порядок доступа к ним, и мы используем структурное программирование как алгоритмическую основу для наших модулей.

Отметьте, как точно высказанное соответствует трем главнейшим аспектам строительства архитектуры: функциональности, разделению компонентов и управлению данными.

4. Структурное программирование



Эдсгер Вибе Дейкстра родился в Роттердаме в 1930 году. Он пережил бомбардировки Роттердама во время Второй мировой войны, оккупацию Нидерландов Германией и в 1948 году окончил среднюю школу с наивысшими отметками по математике, физике, химии и биологии. В марте 1952 года, в возрасте 21 года (и всего за 9 месяцев до моего рождения), Дейкстра устроился на работу в Математический центр

Амстердама и стал самым первым программистом в Нидерландах.

В 1955 году, имея трехлетний опыт программирования и все еще будучи студентом, Дейкстра пришел к выводу, что интеллектуальные вызовы программирования намного обширнее интеллектуальных вызовов теоретической физики. В результате в качестве своей дальнейшей стези он выбрал программирование. В 1957 году Дейкстра женился на Марии Дебетс. В то время в Нидерландах требовали от вступающих в брак указывать профессию. Голландские власти не пожелали принять от Дейкстры документы с указанной профессией «программист»; они никогда не слышали о такой профессии. Поэтому ему пришлось переписать документы и указать профессию «физик-теоретик».

Решение выбрать карьеру программиста Дейкстра обсудил со своим руководителем, Адрианом ван Вейнгаарденом. Дейкстру волновало, что программирование в то время не признавалось ни профессией, ни наукой и что по этой причине его никто не будет воспринимать всерьез. Адриан ответил, что Дейкстра вполне может стать одним из основателей профессии и превратить программирование в науку.

Свою карьеру Дейкстра начинал в эпоху ламповой электроники, когда компьютеры были огромными, хрупкими, медленными, ненадежными и чрезвычайно ограниченными (по современным меркам). В те годы программы записывались двоичным кодом или на примитивном языке ассемблера. Ввод программ в компьютеры осуществлялся с использованием перфолент или перфокарт. Цикл правка—компиляция—тестирование занимал часы, а порой и дни.

В такой примитивной среде Дейкстра сделал свои величайшие открытия.

Доказательство

С самого начала Дейкстра заметил, что программирование — сложная работа и что программисты справляются с ней не очень успешно. Программа любой сложности содержит слишком много деталей, чтобы человеческий мозг смог справиться с ней без посторонней помощи. Стоит упустить из виду одну маленькую деталь, и программа, которая кажется работающей, может завершаться с ошибкой в самых неожиданных местах.

В качестве решения Дейкстра предложил применять математический аппарат *доказательств*. Оно заключалось в построении евклидовой иерархии постулатов, теорем, следствий и лемм. Дейкстра полагал, что программисты смогут использовать эту иерархию подобно математикам. Иными словами, программисты должны использовать проверенные структуры и связывать их с кодом, в правильности которого они хотели бы убедиться.

Дейкстра понимал, что для этого он должен продемонстрировать методику написания доказательств на простых алгоритмах. Но эта задача оказалась довольно сложной.

В ходе исследований Дейкстра обнаружил, что в некоторых случаях использование инструкции `goto` мешает рекурсивному разложению модулей на все меньшие и меньшие единицы и тем самым препятствует применению принципа «разделяй и властвуй», что является необходимым условием для обоснованных доказательств.

Однако в других случаях инструкция `goto` не вызывала этой проблемы. Дейкстра заметил, что эти случаи «доброта качественного» использования `goto` соответствуют простым структурам выбора и управления итерациями, таким

как `if/then/else` и `do/while`. Модули, использующие только такие управляющие структуры, можно было рекурсивно разложить на доказуемые единицы.

Дейкстра знал, что эти управляющие структуры в сочетании с последовательным выполнением занимают особое положение. Они были идентифицированы за два года до этого Бёмом и Якопини, доказавшими, что любую программу можно написать, используя всего три структуры: последовательность, выбор и итерации.

Это было важное открытие: управляющие структуры, делающие доказуемой правильность модуля, в точности совпадали с набором структур, минимально необходимым для написания любой программы. Так родилось структурное программирование.

Дейкстра показал, что доказать правильность последовательности инструкций можно простым перечислением. Методика заключалась в прослеживании последовательности математическим способом от входа до выхода. Она ничем не отличалась от обычного математического доказательства.

Правильность конструкций выбора Дейкстра доказывал через повторяющееся применение приема перечисления, когда прослеживанию подвергался каждый путь. Если оба пути в конечном итоге давали соответствующие математические результаты, их правильность считалась доказанной.

Итерации — несколько иной случай. Чтобы доказать правильность итерации, Дейкстру пришлось использовать *индукцию*. Он доказал методом перечисления правильность случая с единственной итерацией. Затем, опять же методом перечисления, доказал, что если случай для N итераций правильный, значит, правильным будет случай для $N + 1$.

итераций. Используя тот же метод перечисления, он доказал правильность критериев начала и окончания итераций.

Такие доказательства были сложными и трудоемкими, но они были доказательствами. С их развитием идея создания евклидовой иерархии теорем выглядела достижимой в реальности.

Объявление вредным

В 1968 году Дейкстра написал редактору журнала *CACM* письмо под заголовком *Go To Statement Considered Harmful* («О вреде оператора Go To»)¹⁰, которое было опубликовано в мартовском выпуске. В статье он обосновал свою позицию в отношении трех управляющих структур¹¹.

И мир программирования запылал. Тогда у нас не было Интернета, поэтому люди не могли публиковать злобные мемы на Дейкстру и затопить его недружественными сообщениями. Но они могли писать — и писали — письма редакторам многих популярных журналов.

Не все письма были корректными. Некоторые из них были резко отрицательными; другие выражали решительную поддержку. И эта битва продолжалась почти десять лет.

В конце концов спор утих по одной простой причине: Дейкстра победил. По мере развития языков программирования инструкция *goto* все больше сдавала свои позиции, пока, наконец, не исчезла. Большинство современных языков программирования не имеют инструкции *goto*, а некоторые, такие как LISP, никогда ее не имели.

В настоящее время все программисты используют парадигму структурного программирования, хотя и не всегда

осознанно. Просто современные языки не дают возможности неограниченной прямой передачи управления.

Некоторые отмечают сходство инструкции `break` с меткой и исключений в Java с инструкцией `goto`. В действительности эти структуры не являются средствами неограниченной прямой передачи управления, имевшимися в старых языках, таких как Fortran или COBOL. Кроме того, даже языки, сохранившие ключевое слово `goto`, часто ограничивают возможность переходов границами текущей функции.

Функциональная декомпозиция

Структурное программирование дает возможность рекурсивного разложения модулей на доказуемые единицы, что, в свою очередь, означает возможность функциональной декомпозиции. То есть решение большой задачи можно разложить на ряд функций верхнего уровня. Каждую из этих функций в свою очередь можно разложить на ряд функций более низкого уровня, и так до бесконечности. Кроме того, каждую из таких функций можно представить с применением ограниченного набора управляющих структур, предлагаемых парадигмой структурного программирования.

Опираясь на этот фундамент, в конце 1970-х годов и на протяжении 1980-х годов приобрели популярность такие дисциплины, как структурный анализ и структурное проектирование. В тот период многие, например Эд Йордан, Ларри Константин, Том Демарко и Меилир Пейдж-Джонс, продвигали и популяризовали эти дисциплины. Следуя им, програмисты могли разбивать большие системы на модули и компоненты, которые затем можно было разбить на маленькие и доказуемые функции.

Формальные доказательства отсутствуют

Но доказательства так и не появились. Евклидова иерархия теорем не была построена. И программисты не увидели преимуществ использования трудоемкого процесса формального доказательства правильности каждой, даже самой маленькой функции. В конечном итоге мечта Дейкстры рассеялась как дым. Не многие из современных программистов считают, что формальные доказательства являются подходящим способом производства высококачественного программного обеспечения.

Конечно, формальный евклидов стиль математических доказательств не единственная стратегия создания чего-то правильного. Другой, более успешной стратегией является *научный метод*.

Наука во спасение

Принципиальное отличие науки от математики заключается в том, что правильность научных теорий и законов нельзя доказать. Я не смогу доказать верность второго закона движения Ньютона, $F = ma$, или закона гравитации, $F = Gm_1m_2/r^2$. Я могу продемонстрировать действие этих законов и провести измерения, подтверждающие их правильность до многих знаков после запятой, но я не смогу доказать их в математическом смысле. Я могу провести массу экспериментов и собрать массу эмпирических подтверждений, но всегда остается вероятность, что какой-то эксперимент покажет, что эти законы движения и гравитации неверны.

Такова природа научных теорий и законов: их можно *сфальсифицировать*, но нельзя доказать.

Тем не менее мы верим в эти законы. Каждый раз, садясь в автомобиль, мы ставим свою жизнь на то, что формула $F = ma$ точно описывает окружающий мир. Каждый раз, делая шаг, мы ставим свои здоровье и безопасность на верность формулы $F = Gm_1m_2/r^2$.

Наука не требует доказательства истинности утверждений, чаще она требует *доказательства их ложности*. Утверждения, доказать ложность которых не удается после многих усилий, мы считаем истинными.

Конечно, не все утверждения требуют доказательств. Например, утверждение «это — ложь» не является ни истинным, ни ложным. Это один из простейших примеров утверждений, не требующих доказательств.

Подводя итог, можно сказать, что математика — это дисциплина доказательства истинности утверждений, требующих доказательства. Наука, напротив, — дисциплина доказательства ложности утверждений, требующих доказательства.

Тестирование

Однажды Дейкстра сказал: «Тестирование показывает присутствие ошибок, а не их отсутствие». Иными словами, тестированием можно доказать неправильность программы, но нельзя доказать ее правильность. Все, что дает тестирование после приложения достаточных усилий, — это уверенность, что программа действует достаточно правильно.

Следствия из этого факта могут показаться ошеломляющими. Разработка программного обеспечения не является математической задачей, даже при том, что она связана с применением математических конструкций. Эта

сфера деятельности больше похожа на науку. Мы убеждаемся в правильности, потерпев неудачу в попытке доказать неправильность.

Такие доказательства неправильности можно применить только к доказуемым программам. Недоказуемую программу — например, из-за неумеренного использования `goto` — нельзя считать правильной, сколько бы тестов к ней ни применялось.

Парадигма структурного программирования заставляет нас рекурсивно разбивать программы на множество мелких и доказуемых функций. В результате мы получаем возможность использовать тесты, чтобы попытаться доказать их неправильность. Если такие тесты терпят неудачу, тогда мы считаем функции достаточно правильными.

Заключение

Именно эта возможность создавать программные единицы, неправильность которых можно доказать, является главной ценностью структурного программирования. Именно поэтому современные языки обычно не поддерживают неограниченное применение инструкций `goto`. Кроме того, именно поэтому *функциональная декомпозиция* считается одним из лучших приемов на архитектурном уровне.

На всех уровнях, от маленьких функций до больших компонентов, разработка программного обеспечения напоминает науку, и поэтому в ней применяется правило опровергающих доказательств. Программные архитекторы стремятся определить модули, компоненты и службы, неправильность которых легко можно было бы доказать (протестировать). Для этого они используют ограничения,

напоминающие ограничения в структурном программировании, хотя и более высокого уровня.

Именно эти ограничения мы будем подробно изучать в последующих главах.

[10](#) На самом деле Дейкстра озаглавил свое письмо *A Case Against the Goto Statement* («Дело против оператора goto»), но редактор CACM Никлаус Вирт изменил заголовок.
— Примеч. пер.

[11](#) Перевод статьи на русский язык можно найти по адресу <http://hosting.vspu.ac.ru/~chul/dijkstra/goto/goto.htm>. — Примеч. пер.

5. Объектно-ориентированное программирование



Как мы увидим далее, для создания хорошей архитектуры необходимо понимать и уметь применять принципы объектно-ориентированного программирования (ОО). Но что такое ОО?

Один из возможных ответов на этот вопрос: «комбинация данных и функций». Однако, несмотря на частое цитирование, этот ответ нельзя признать точным, потому что он предполагает, что `o.f()` — это нечто отличное от `f(o)`. Это абсурд. Программисты передавали структуры в функции задолго до 1966 года, когда Даль и Нюгор перенесли кадр стека функции в динамическую память и изобрели ОО.

Другой распространенный ответ: «способ моделирования реального мира». Это слишком уклончивый ответ. Что в действительности означает «моделирование реального мира» и почему нам может понадобиться такое моделирование? Возможно, эта фраза подразумевает, что ОО делает программное обеспечение проще для понимания, потому что оно становится ближе к реальному миру, но и такое объяснение слишком размыто и уклончиво. Оно не отвечает на вопрос, что же такое ОО.

Некоторые, чтобы объяснить природу ОО, прибегают к трем волшебным словам: *инкапсуляция, наследование и полиморфизм*. Они подразумевают, что ОО является комплексом из этих трех понятий или, по крайней мере, что объектно-ориентированный язык должен их поддерживать.

Давайте исследуем эти понятия по очереди.

Инкапсуляция?

Инкапсуляция упоминается как часть определения ОО потому, что языки ОО поддерживают простой и эффективный способ инкапсуляции данных и функций. Как результат, есть возможность очертить круг связанных данных и функций. За пределами круга эти данные невидимы и доступны только некоторые функции. Воплощение этого понятия можно

наблюдать в виде приватных членов данных и общедоступных членов-функций класса.

Эта идея определенно не уникальная для ОО. Например, в языке С имеется превосходная поддержка инкапсуляции. Рассмотрим простую программу на С:

point.h

```
struct Point;
struct Point* makePoint(double x, double y);
double distance (struct Point *p1, struct Point *p2);
```

point.c

```
#include "point.h"
#include <stdlib.h>
#include <math.h>
```

```
struct Point {
    double x,y;
};
```

```
struct Point* makepoint(double x, double y) {
    struct Point* p = malloc(sizeof(struct Point));
    p->x = x;
    p->y = y;
    return p;
}
```

```
double distance(struct Point* p1, struct Point* p2) {
```

```
double dx = p1->x - p2->x;  
double dy = p1->y - p2->y;  
return sqrt(dx*dx+dy*dy);  
}
```

Пользователи `point.h` не имеют доступа к членам структуры `Point`. Они могут вызывать функции `makePoint()` и `distance()`, но не имеют никакого представления о реализации структуры `Point` и функций для работы с ней.

Это отличный пример поддержки инкапсуляции не в объектно-ориентированном языке. Программисты на С постоянно использовали подобные приемы. Мы можем объявить структуры данных и функции в заголовочных файлах и реализовать их в файлах реализации. И наши пользователи никогда не получат доступа к элементам в этих файлах реализации.

Но затем пришел объектно-ориентированный C++ и превосходная инкапсуляция в С оказалась разрушенной.

По техническим причинам¹² компилятор C++ требует определять переменные-члены класса в заголовочном файле. В результате объектно-ориентированная версия предыдущей программы `Point` приобретает такой вид:

point.h

```
class Point {  
public:  
    Point(double x, double y);  
    double distance(const Point& p) const;  
private:  
    double x;
```

```
    double y;  
};
```

point.cc

```
#include "point.h"  
#include <math.h>
```

```
Point::Point(double x, double y)  
: x(x), y(y)  
{}
```

```
double Point::distance(const Point& p) const {  
    double dx = x-p.x;  
    double dy = y-p.y;  
    return sqrt(dx*dx + dy*dy);  
}
```

Теперь пользователи заголовочного файла `point.h` знают о переменных-членах `x` и `y`! Компилятор не позволит обратиться к ним непосредственно, но клиент все равно знает об их существовании. Например, если имена этих членов изменятся, файл `point.cc` придется скомпилировать заново! Инкапсуляция оказалась разрушенной.

Введением в язык ключевых слов `public`, `private` и `protected` инкапсуляция была частично восстановлена. Однако это был лишь *грубый прием* (хак), обусловленный технической необходимостью компилятора видеть все переменные-члены в заголовочном файле.

Языки Java и C# полностью отменили деление на заголовок/реализацию, ослабив инкапсуляцию еще больше. В этих языках

невозможно разделить объявление и определение класса.

По описанным причинам трудно согласиться, что ОО зависит от строгой инкапсуляции. В действительности многие языки ОО практически не имеют принудительной инкапсуляции¹³.

ОО безусловно полагается на поведение программистов — что они не станут использовать обходные приемы для работы с инкапсулированными данными. То есть языки, заявляющие о поддержке ОО, фактически ослабили превосходную инкапсуляцию, некогда существовавшую в С.

Наследование?

Языки ОО не улучшили инкапсуляцию, зато они дали нам наследование.

Точнее — ее разновидность. По сути, наследование — это всего лишь повторное объявление группы переменных и функций в ограниченной области видимости. Нечто похожее программисты на С проделывали вручную задолго до появления языков ОО¹⁴.

Взгляните на дополнение к нашей исходной программе `point.h` на языке С:

namedPoint.h

```
struct NamedPoint;
```

```
struct NamedPoint* makeNamedPoint(double x, double y, char* name);
```

```
void setName(struct NamedPoint* np, char* name);
```

```
char* getName(struct NamedPoint* np);
```

namedPoint.c

```
#include "namedPoint.h"
#include <stdlib.h>

struct NamedPoint {
    double x,y;
    char* name;
};

struct NamedPoint* makeNamedPoint(double x, double y, char*
name) {
    struct NamedPoint* p = malloc(sizeof(struct NamedPoint));
    p->x = x;
    p->y = y;
    p->name = name;
    return p;
}

void setName(struct NamedPoint* np, char* name) {
    np->name = name;
}

char* getName(struct NamedPoint* np) {
    return np->name;
}
```

main.c

```
#include "point.h"
```

```
#include "namedPoint.h"
#include <stdio.h>

int main(int ac, char** av) {
    struct NamedPoint* origin = makeNamedPoint(0.0, 0.0,
"origin");
    struct NamedPoint* upperRight = makeNamedPoint
(1.0, 1.0, "upperRight");
    printf("distance=%f\n",
    distance(
        (struct Point*) origin,
        (struct Point*) upperRight));
}
```

Внимательно рассмотрев основной код в файле `main.c`, можно заметить, что структура данных `NamedPoint` используется, как если бы она была производной от структуры `Point`. Такое оказалось возможным потому, что первые два поля в `NamedPoint` совпадают с полями в `Point`. Проще говоря, `NamedPoint` может маскироваться под `Point`, потому что `NamedPoint` фактически является надмножеством `Point` и имеет члены, соответствующие структуре `Point`, следующие в том же порядке.

Этот прием широко применялся¹⁵ программистами до появления ОО. Фактически именно так С++ реализует единственное наследование.

То есть можно сказать, что некоторая разновидность наследования у нас имелась задолго до появления языков ОО. Впрочем, это утверждение не совсем истинно. У нас имелся трюк, хитрость, не настолько удобный, как настоящее наследование. Кроме того, с помощью описанного приема

очень сложно получить что-то похожее на множественное наследование.

Обратите также внимание, как в `main.c` мне пришлось приводить аргументы `NamedPoint` к типу `Point`. В настоящем языке ОО такое приведение к родительскому типу производится неявно.

Справедливо ради следует отметить, что языки ОО действительно сделали маскировку структур данных более удобной, хотя это и не совсем новая особенность.

Итак, мы не можем дать идею ОО ни одного очка за инкапсуляцию и можем дать лишь пол-очка за наследование. Пока что общий счет не впечатляет.

Но у нас есть еще одно понятие.

Полиморфизм?

Была ли возможность реализовать полиморфное поведение до появления языков ОО? Конечно! Взгляните на следующую простую программу сору на языке С.

```
#include <stdio.h>

void copy() {
    int c;
    while ((c=getchar()) != EOF)
        putchar(c);
}
```

Функция `getchar()` читает символы из `STDIN`. Но какое устройство в действительности скрыто за ширмой `STDIN`? Функция `putchar()` записывает символы в устройство

STDOUT. Но что это за устройство? Эти функции являются *полиморфными* — их поведение зависит от типов устройств STDIN и STDOUT.

В некотором смысле STDIN и STDOUT похожи на интерфейсы в силе Java, когда для каждого устройства имеется своя реализация этих интерфейсов. Конечно, в примере программы на С нет никаких интерфейсов, но как тогда вызов `getchar()` передается драйверу устройства, который фактически читает символ?

Ответ на этот вопрос прост: операционная система UNIX требует, чтобы каждый драйвер устройства ввода/вывода реализовал пять стандартных функций¹⁶: `open`, `close`, `read`, `write` и `seek`. Сигнатуры этих функций должны совпадать для всех драйверов.

Структура FILE имеет пять указателей на функции. В нашем случае она могла бы выглядеть как-то так:

```
struct FILE {  
    void (*open)(char* name, int mode);  
    void (*close)();  
    int (*read)();  
    void (*write)(char);  
    void (*seek)(long index, int mode);  
};
```

Драйвер консоли определяет эти функции и инициализирует указатели на них в структуре FILE примерно так:

```
#include "file.h"
```

```
void open(char* name, int mode) {/*...*/}
void close() {/*...*/};
int read() {int c; /*...*/ return c;}
void write(char c) {/*...*/}
void seek(long index, int mode) {/*...*/}

struct FILE console = {open, close, read,
write, seek};
```

Если теперь предположить, что символ STDIN определен как указатель FILE* и ссылается на структуру console, тогда getchar() можно реализовать как-то так:

```
extern struct FILE* STDIN;

int getchar() {
    return STDIN->read();
}
```

Иными словами, getchar() просто вызывает функцию, на которую ссылается указатель read в структуре FILE, на которую, в свою очередь, ссылается STDIN.

Этот простой трюк составляет основу полиморфизма в ОО. В C++, например, каждая виртуальная функция в классе представлена указателем в таблице виртуальных методов vtable и все вызовы виртуальных функций выполняются через эту таблицу. Конструкторы производных классов просто инициализируют таблицу vtable объекта указателями на свои версии функций.

Суть полиморфизма заключается в применении указателей на функции. Программисты использовали указатели на функции для достижения полиморфного поведения еще со

времен появления архитектуры фон Неймана в конце 1940-х годов. Иными словами, парадигма ОО не принесла ничего нового.

Впрочем, это не совсем верно. Пусть полиморфизм появился раньше языков ОО, но они сделали его намного надежнее и удобнее.

Проблема явного использования указателей на функции для создания полиморфного поведения в том, что указатели на функции по своей природе опасны. Такое их применение оговаривается множеством соглашений. Вы должны помнить об этих соглашениях и инициализировать указатели. Вы должны помнить об этих соглашениях и вызывать функции посредством указателей. Если какой-то программист забудет о соглашениях, возникшую в результате ошибку будет чертовски трудно отыскать и устраниТЬ.

Языки ОО избавляют от необходимости помнить об этих соглашениях и, соответственно, устраняют опасности, связанные с этим. Поддержка полиморфизма на уровне языка делает его использование тривиально простым. Это обстоятельство открывает новые возможности, о которых программисты на С могли только мечтать. Отсюда можно заключить, что ОО накладывает ограничение на косвенную передачу управления.

Сильные стороны полиморфизма

Какими положительными чертами обладает полиморфизм? Чтобы в полной мере оценить их, рассмотрим пример программы сору. Что случится с программой, если создать новое устройство ввода/вывода? Допустим, мы решили использовать программу сору для копирования данных из устройства распознавания рукописного текста в устройство

синтеза речи: что нужно изменить в программе `copy`, чтобы она смогла работать с новыми устройствами?

Самое интересное, что никаких изменений не требуется! В действительности нам не придется даже перекомпилировать программу `copy`. Почему? Потому что исходный код программы `copy` не зависит от исходного кода драйверов ввода/вывода. Пока драйверы реализуют пять стандартных функций, определяемых структурой `FILE`, программа `copy` сможет с успехом их использовать.

Проще говоря, устройства ввода/вывода превратились в плагины для программы `copy`.

Почему операционная система UNIX превратила устройства ввода/вывода в плагины? Потому что в конце 1950-х годов мы поняли, что наши программы не должны зависеть от конкретных устройств. Почему? Потому что мы успели написать массу программ, зависящих от устройств, прежде чем смогли понять, что в действительности мы хотели бы, чтобы эти программы, выполняя свою работу, могли бы использовать разные устройства.

Например, раньше часто писались программы, читавшие исходные данные из пакета перфокарт¹⁷ и пробивавшие на перфораторе новую стопку перфокарт с результатами. Позднее наши клиенты стали передавать исходные данные не на перфокартах, а на магнитных лентах. Это было неудобно, потому что приходилось переписывать большие фрагменты первоначальных программ. Было бы намного удобнее, если бы та же программа могла работать и с перфокартами, и с магнитной лентой.

Для поддержки независимости от устройств ввода/вывода была придумана архитектура плагинов и реализована практически во всех операционных системах. Но даже после этого большинство программистов не давали распространения

этой идеи в своих программах, потому что использование указателей на функции было опасно.

Объектно-ориентированная парадигма позволила использовать архитектуру плагинов повсеместно.

Инверсия зависимости

Представьте, на что походило программное обеспечение до появления надежного и удобного механизма полиморфизма. В типичном дереве вызовов главная функция вызывала функции верхнего уровня, которые вызывали функции среднего уровня, в свою очередь, вызывавшие функции нижнего уровня. Однако в таком дереве вызовов зависимости исходного кода непреклонно следовали за потоком управления (рис. 5.1).

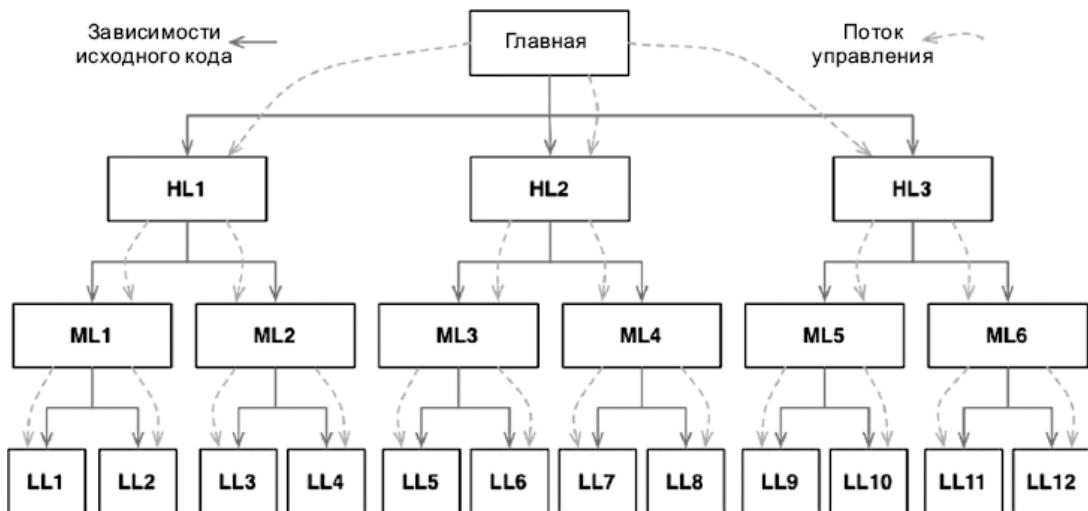


Рис. 5.1. Зависимости исходного кода следуют за потоком управления

Чтобы вызвать одну из функций верхнего уровня, функция `main` должна сослаться на модуль, содержащий эту функцию. В языке C для этой цели используется директива `#include`. В Java — инструкция `import`. В C# — инструкция `using`. В действительности любой вызывающий код был вынужден ссылаться на модуль, содержащий вызываемый код.

Эти требования предоставляли архитектору программного обеспечения несколько вариантов. Поток управления определяется поведением системы, а зависимости исходного кода определяются этим потоком управления.

Однако когда в игру включился полиморфизм, стало возможным нечто совершенно иное (рис. 5.2).

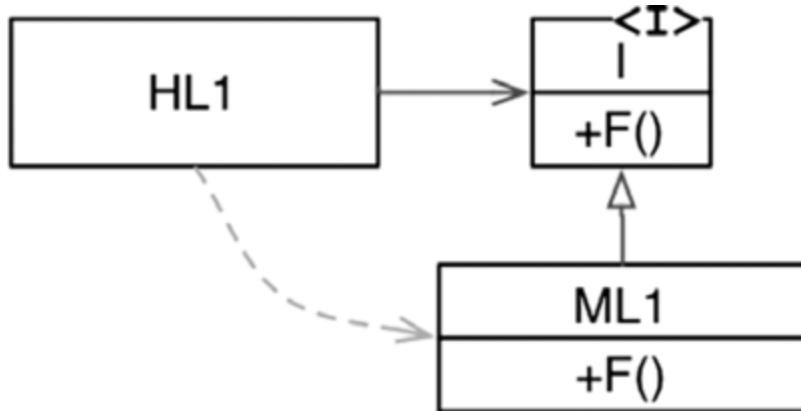


Рис. 5.2. Инверсия зависимости

На рис. 5.2 модуль верхнего уровня *HL1* вызывает функцию *F()* из модуля среднего уровня *ML1*. Вызов посредством интерфейса является уловкой лишь для исходного кода. Во время выполнения интерфейсов не существует. Модуль *HL1* просто вызывает *F()* внутри *ML1*¹⁸.

Но обратите внимание, что направление зависимости в исходном коде (отношение наследования) между *ML1* и интерфейсом *I* поменялось на противоположное по отношению к потоку управления. Этот эффект называют *инверсией зависимости* (dependency inversion), и он имеет далеко идущие последствия для архитекторов программного обеспечения.

Факт поддержки языками ОО надежного и удобного механизма полиморфизма означает, что любую зависимость исходного кода, где бы она ни находилась, можно инвертировать.

Теперь вернемся к дереву вызовов, изображеному на рис. 5.1, и к множеству зависимостей в его исходном коде. Любую из зависимостей в этом исходном коде можно обратить, добавив интерфейс.

При таком подходе архитекторы, работающие в системах, которые написаны на объектно-ориентированных языках, получают *абсолютный контроль* над направлением всех зависимостей в исходном коде. Они не ограничены только направлением потока управления. Неважно, какой модуль вызывает и какой модуль вызывается, архитектор может определить зависимость в исходном коде в любом направлении.

Какая возможность! И эту возможность открывает ОО. Собственно, это все, что дает ОО, — по крайней мере с точки зрения архитектора.

Что можно сделать, обладая этой возможностью? Можно, например, переупорядочить зависимости в исходном коде так, что база данных и пользовательский интерфейс (ПИ) в вашей системе будут зависеть от бизнес-правил (рис. 5.3), а не наоборот.

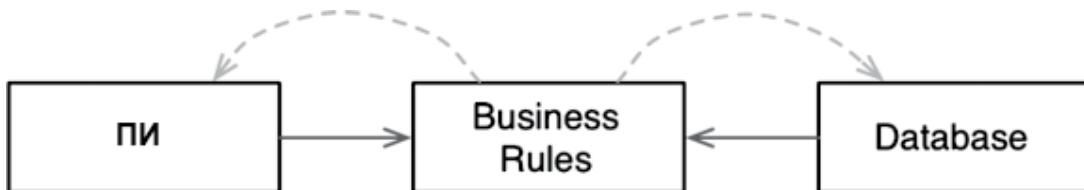


Рис. 5.3. База данных и пользовательский интерфейс зависят от бизнес-правил

Это означает, что ПИ и база данных могут быть плагинами к бизнес-правилам. То есть в исходном коде с реализацией бизнес-правил могут отсутствовать любые ссылки на ПИ или базу данных.

Как следствие, бизнес-правила, ПИ и базу данных можно скомпилировать в три разных компонента или единицы

развертывания (например, jar-файлы, библиотеки DLL или файлы Gem), имеющих те же зависимости, как в исходном коде. Компонент с бизнес-правилами не будет зависеть от компонентов, реализующих ПИ и базу данных.

Как результат, появляется возможность *развертывать бизнес-правила независимо* от ПИ и базы данных. Изменения в ПИ или в базе данных не должны оказывать никакого влияния на бизнес-правила. То есть компоненты можно развертывать отдельно и независимо.

Проще говоря, когда реализация компонента изменится, достаточно повторно развернуть только этот компонент. Это *независимость развертывания*.

Если система состоит из модулей, которые можно развертывать независимо, их можно разрабатывать независимо, разными командами. Это *независимость разработки*.

Заключение

Что такое ОО? Существует много взглядов и ответов на этот вопрос. Однако для программного архитектора ответ очевиден: ОО дает, посредством поддержки полиморфизма, абсолютный контроль над всеми зависимостями в исходном коде. Это позволяет архитектору создать архитектуру со сменными модулями (плагинами), в которой модули верхнего уровня не зависят от модулей нижнего уровня. Низкоуровневые детали не выходят за рамки модулей плагинов, которые можно развертывать и разрабатывать независимо от модулей верхнего уровня.

[12](#) Чтобы иметь возможность определить размер экземпляра каждого класса.

[13](#) Например, Smalltalk, Python, JavaScript, Lua и Ruby.

[14](#) И не только программисты на С: большинство языков той эпохи позволяли маскировать одни структуры данных под другие.

[15](#) И продолжает применяться.

[16](#) В разных версиях UNIX требования разные; это всего лишь пример.

[17](#) Перфокарты IBM Hollerith имели ширину 80 колонок. Я уверен, что многие из вас никогда даже не видели их, но они широко были распространены в 1950-е, 1960-е и даже в 1970-е годы.

[18](#) Хотя и косвенно.

6. Функциональное программирование



Многие идеи функционального программирования появились задолго до появления самого программирования.

Эта парадигма в значительной мере основана на λ -исчислении, изобретенном Алонзо Чёрчем в 1930-х годах.

Квадраты целых чисел

Суть функционального программирования проще объяснить на примерах. С этой целью исследуем решение простой задачи: вывод квадратов первых 25 целых чисел (то есть от 0 до 24).

В языках, подобных Java, эту задачу можно решить так:

```
public class Squint {  
    public static void main(String args[]) {  
        for (int i=0; i<25; i++)  
            System.out.println(i*i);  
    }  
}
```

В Clojure, функциональном языке и производном от языка Lisp, аналогичную программу можно записать так:

```
(println (take 25 (map (fn [x] (* x x))  
(range))))
```

Этот код может показаться немного странным, если вы не знакомы с Lisp. Поэтому я, с вашего позволения, немного переоформлю его и добавлю несколько комментариев.

```
(println ;----- Вывести  
(take 25 ;_____ первые 25  
      (map (fn [x] (* x x)) ;_ квадратов  
            (range)))) ;_____ целых чисел
```

Совершенно понятно, что `println`, `take`, `map` и `range` — это функции. В языке Lisp вызов функции производится помещением ее имени в круглые скобки. Например, `(range)` — это вызов функции `range`.

Выражение `(fn [x] (* x x))` — это анонимная функция, которая, в свою очередь, вызывает функцию умножения и передает ей две копии входного аргумента. Иными словами, она вычисляет квадрат заданного числа.

Взглянем на эту программу еще раз, начав с самого внутреннего вызова функции:

- функция `range` возвращает бесконечный список целых чисел, начиная с 0;
- этот список передается функции `map`, которая вызывает анонимную функцию для вычисления квадрата каждого элемента и производит бесконечный список квадратов;
- список квадратов передается функции `take`, которая возвращает новый список, содержащий только первые 25 элементов;
- функция `println` выводит этот самый список квадратов первых 25 целых чисел.

Если вас напугало упоминание бесконечных списков, не волнуйтесь. В действительности программа создаст только первые 25 элементов этих бесконечных списков. Дело в том, что новые элементы бесконечных списков не создаются, пока программа не обратится к ним.

Если все вышесказанное показалось вам запутанным и непонятным, тогда можете отложить эту книгу и прекрасно

проводи время, изучая функциональное программирование и язык Clojure. В этой книге я не буду рассматривать эти темы, так как не это является моей целью.

Моя цель — показать важнейшее отличие между программами на Clojure и Java. В программе на Java используется изменяющаяся переменная — переменная, состояние которой изменяется в ходе выполнения программы. Это переменная `i` — переменная цикла. В программе на Clojure, напротив, нет изменяющихся переменных. В ней присутствуют инициализируемые переменные, такие как `x`, но они никогда не изменяются.

В результате мы пришли к удивительному утверждению: переменные в функциональных языках *не изменяются*.

Неизменяемость и архитектура

Почему этот аспект важен с архитектурной точки зрения? Почему архитектора должна волновать изменчивость переменных? Ответ на этот вопрос до нелепого прост: все состояния гонки (*race condition*), взаимоблокировки (*deadlocks*) и проблемы параллельного обновления обусловлены изменяемостью переменных. Если в программе нет изменяющихся переменных, она никогда не окажется в состоянии гонки и никогда не столкнется с проблемами одновременного изменения. В отсутствие изменяющихся блокировок программа не может попасть в состояние взаимоблокировки.

Иными словами, все проблемы, характерные для приложений с конкурирующими вычислениями, — с которыми нам приходится сталкиваться, когда требуется организовать многопоточное выполнение и задействовать вычислительную мощность нескольких процессоров, исчезают сами собой в отсутствие изменяющихся переменных.

Вы, как архитектор, обязаны интересоваться проблемами конкуренции. Вы должны гарантировать надежность и устойчивость проектируемых вами систем в многопоточном и многопроцессорном окружении. И один из главных вопросов, которые вы должны задать себе: достижима ли неизменяемость на практике?

Ответ на этот вопрос таков: да, если у вас есть неограниченный объем памяти и процессор с неограниченной скоростью вычислений. В отсутствие этих бесконечных ресурсов ответ выглядит не так однозначно. Да, неизменяемость достижима, но при определенных компромиссах.

Рассмотрим некоторые из этих компромиссов.

Ограничение изменяемости

Один из самых общих компромиссов, на которые приходится идти ради неизменяемости, — деление приложения или служб внутри приложения на изменяемые и неизменяемые компоненты. Неизменяемые компоненты решают свои задачи исключительно функциональным способом, без использования изменяемых переменных. Они взаимодействуют с другими компонентами, не являющимися чисто функциональными и допускающими изменение состояний переменных (рис. 6.1).

Изменяющее состояние этих других компонентов открыто всем проблемам многопоточного выполнения, поэтому для защиты изменяемых переменных от конкурирующих обновлений и состояния гонки часто используется некоторая разновидность *транзакционной памяти*.

Транзакционная память интерпретирует переменные в оперативной памяти подобно тому, как база данных

интерпретирует записи на диске. Она защищает переменные, применяя механизм транзакций с повторениями.

Простым примером могут служить атомы в Clojure:

```
(def counter (atom 0)) ; инициализировать
счетчик нулем
(swap! counter inc) ; безопасно увеличить
счетчик counter.
```

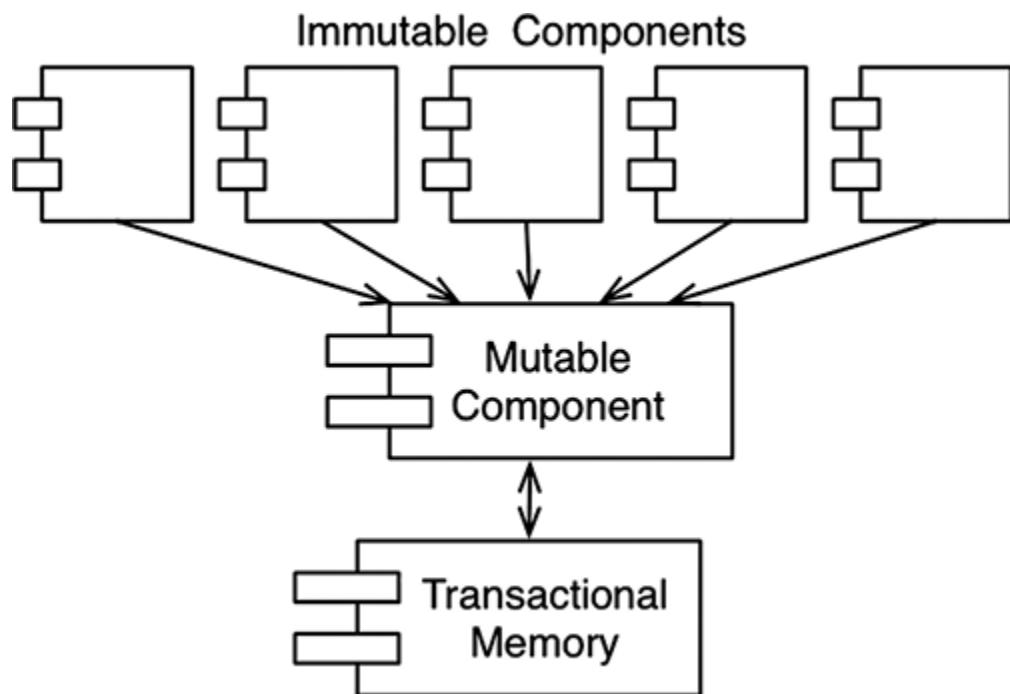


Рис. 6.1. Изменяемое состояние и транзакционная память

В этом фрагменте переменная `counter` определяется как атом (с помощью функции `atom`). Атом в языке Clojure — это особая переменная, способная изменяться при очень ограниченных условиях, соблюдение которых гарантирует функция `swap!`.

Функция `swap!`, показанная в предыдущем фрагменте, принимает два аргумента: атом, подлежащий изменению, и функцию, вычисляющую новое значение для сохранения в атоме. В данном примере атом `counter` получит значение,

вычисленное функцией `inc`, которая просто увеличивает свой аргумент на единицу.

Функция `swap!` реализует традиционный алгоритм *сравнить и присвоить*. Она читает значение `counter` и передает его функции `inc`. Когда `inc` вернет управление, доступ к переменной `counter` блокируется и ее значение сравнивается со значением, переданным в `inc`. Если они равны, в `counter` записывается значение, которое вернула функция `inc`, и блокировка освобождается. Иначе блокировка освобождается и попытка повторяется.

Механизм атомов эффективен для простых приложений. Но, к сожалению, он не гарантирует абсолютную защищенность от конкурирующих обновлений и взаимоблокировок, когда в игру вступает несколько взаимозависимых изменяемых переменных. В таких ситуациях предпочтительнее использовать более надежные средства.

Суть в том, что правильно организованные приложения должны делиться на компоненты, имеющие и не имеющие изменяемых переменных. Такое деление обеспечивается использованием подходящих дисциплин для защиты изменяемых переменных.

Со стороны архитекторов было бы разумно как можно больше кода поместить в неизменяемые компоненты и как можно меньше — в компоненты, допускающие возможность изменения.

Регистрация событий

Проблема ограниченности объема памяти и конечной вычислительной мощности процессоров очень быстро теряет свою актуальность. В настоящее время обычными стали компьютеры с процессорами, выполняющими миллиарды

инструкций в секунду, и имеющие объем оперативной памяти в несколько миллиардов байтов. Чем больше памяти, тем быстрее работает компьютер и тем меньше потребность в изменяемом состоянии.

Как простой пример, представьте банковское приложение, управляющее счетами клиентов. Оно изменяет счета, когда выполняются операции зачисления или списания средств.

Теперь вообразите, что вместо сумм на счетах мы сохраняем только информацию об операциях. Всякий раз, когда кто-то пожелает узнать баланс своего счета, мы просто выполняем все транзакции с состоянием счета от момента его открытия. Эта схема не требует изменяемых переменных.

Очевидно, такой подход кажется абсурдным. С течением времени количество транзакций растет и в какой-то момент объем вычислений, необходимый для определения баланса счета, станет недопустимо большим. Чтобы такая схема могла работать всегда, мы должны иметь неограниченный объем памяти и процессор с бесконечно высокой скоростью вычислений.

Но иногда не требуется, чтобы эта схема работала всегда. Иногда у нас достаточно памяти и вычислительной мощности, чтобы подобная схема работала в течение времени выполнения приложения.

Эта идея положена в основу технологии *регистрации событий* (event sourcing)¹⁹. Регистрация событий (event sourcing) — это стратегия, согласно которой сохраняются транзакции, а не состояние. Когда требуется получить состояние, мы просто применяем все транзакции с самого начала.

Конечно, реализуя подобную стратегию, можно использовать компромиссные решения для экономии ресурсов. Например, вычислять и сохранять состояние каждую полночь.

А затем, когда потребуется получить информацию о состоянии, выполнить лишь транзакции, имевшие место после полуночи.

Теперь представьте хранилище данных, которое потребуется для поддержки этой схемы: оно должно быть о-о-очень большим. В настоящее время объемы хранилищ данных растут так быстро, что, например, триллион байтов мы уже не считаем большим объемом — то есть нам понадобится намного, намного больше.

Что особенно важно, никакая информация не удаляется из такого хранилища и не изменяется. Как следствие, от набора CRUD-операций²⁰ в приложениях остаются только CR. Также отсутствие операций изменения и/или удаления с хранилищем устраняет любые проблемы конкурирующих обновлений.

Обладая хранилищем достаточного объема и достаточной вычислительной мощностью, мы можем сделать свои приложения полностью неизменяемыми — и, как следствие, *полностью функциональными*.

Если это все еще кажется вам абсурдным, вспомните, как работают системы управления версиями исходного кода.

Заключение

Итак:

- Структурное программирование накладывает ограничение на прямую передачу управления.
- Объектно-ориентированное программирование накладывает ограничение на косвенную передачу управления.
- Функциональное программирование накладывает ограничение на присваивание.

Каждая из этих парадигм что-то отнимает у нас. Каждая ограничивает подходы к написанию исходного кода. Ни одна не добавляет новых возможностей.

Фактически последние полвека мы учились тому, *как не надо делать*.

Осознав это, мы должны признать неприятный факт: разработка программного обеспечения не является быстро развивающейся индустрией. Правила остаются теми же, какими они были в 1946 году, когда Аллан Тьюринг написал первый код, который мог выполнить электронный компьютер. Инструменты изменились, аппаратура изменилась, но суть программного обеспечения осталась прежней.

Программное обеспечение — материал для компьютерных программ — состоит из последовательностей, выбора, итераций и косвенности. Ни больше ни меньше.

[19](#) Спасибо Грегу Янгу, что объяснил мне суть этого понятия.

[20](#) CRUD — аббревиатура, обозначающая набор основных операций с данными: Create (создание), Read (чтение), Update (изменение) и Delete (удаление). — Примеч. *пер.*

III. Принципы дизайна

Хорошая программная система начинается с чистого кода. С одной стороны, если здание строить из плохих кирпичей, его архитектура не имеет большого значения. С другой стороны, плохие кирпичи можно перемешать с хорошими. Именно на этом основаны принципы SOLID.

Принципы SOLID определяют, как объединять функции и структуры данных в классы и как эти классы должны сочетаться друг с другом. Использование слова «класс» не означает, что эти принципы применимы только к объектно-ориентированному программному коду. В данном случае «класс» означает лишь инструмент объединения функций и данных в группы. Любая программная система имеет такие объединения, как бы они ни назывались, «класс» или как-то еще. Принципы SOLID применяются к этим объединениям.

Цель принципов — создать программные структуры среднего уровня, которые:

- терпимы к изменениям;
- просты и понятны;
- образуют основу для компонентов, которые могут использоваться во многих программных системах.

Термин «средний уровень» отражает тот факт, что эти принципы применяются программистами на уровне модулей. Они применяются на уровне, лежащем непосредственно над уровнем программного кода, и помогают определять программные структуры, используемые в модулях и компонентах.

Как из хороших кирпичей можно сложить никуда не годную стену, так из хорошо продуманных компонентов среднего уровня можно создать никуда не годную систему. Поэтому сразу после знакомства с принципами SOLID мы перейдем к их аналогам в мире компонентов, а затем к высокоуровневым принципам создания архитектур.

Принципы SOLID имеют долгую историю. Я начал собирать их в конце 1980-х годов, обсуждая принципы проектирования программного обеспечения с другими пользователями USENET (ранняя разновидность Facebook). На протяжении многих лет принципы смешались и изменялись. Некоторые исчезали. Другие объединялись. А какие-то добавлялись. В окончательном виде они были сформулированы в начале 2000-х годов, хотя и в другом порядке, чем я представлял.

В 2004 году или около того Майкл Физерс приспал мне электронное письмо, в котором сообщил, что если переупорядочить мои принципы, из их первых букв можно составить слово SOLID²¹ — так появились принципы SOLID.

Последующие главы подробнее описывают каждый принцип, а пока познакомьтесь с краткой аннотацией:

- **SRP:** Single Responsibility Principle — принцип единственной ответственности.

Действительное следствие закона Конвея: лучшей является такая структура программной системы, которая формируется в основном под влиянием социальной структуры организации, использующей эту систему, поэтому каждый программный модуль имеет одну и только одну причину для изменения.

- **OCP:** Open-Closed Principle — принцип открытости/закрытости.

Этот принцип был сформулирован Берtrandом Мейером в 1980-х годах. Суть его сводится к следующему: простая для изменения система должна предусматривать простую возможность изменения ее поведения добавлением нового, но не изменением существующего кода.

- **LSP:** Liskov Substitution Principle — принцип подстановки Барбары Лисков.

Определение подтипов Барбары Лисков известно с 1988 года. В двух словах, этот принцип утверждает, что для создания программных систем из взаимозаменяемых частей эти части должны соответствовать контракту, который позволяет заменять эти части друг другом.

- **ISP:** Interface Segregation Principle — принцип разделения интерфейсов.

Этот принцип призывает разработчиков программного обеспечения избегать зависимости от всего, что не используется.

- **DIP:** Dependency Inversion Principle — принцип инверсии зависимости.

Код, реализующий высокоуровневую политику, не должен зависеть от кода, реализующего низкоуровневые детали. Напротив, детали должны зависеть от политики.

Эти принципы детально описаны во множестве публикаций²². Последующие главы освещают влияние этих принципов на проектирование архитектур, не повторяя подробное обсуждение из этих публикаций. Если вы еще не знакомы с перечисленными принципами, обсуждения, следующего далее, будет недостаточно, чтобы понять их во всех подробностях, поэтому я рекомендую обратиться к документам, перечисленным в сноске.

²¹ В данном случае слово «SOLID» можно перевести как «прочный», «надежный», «основательный». — Примеч. пер.

²² Например, *Agile Software Development, Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002 (Роберт С. Мартин. Гибкая разработка программ на Java и C++: принципы, паттерны и методики. М.: Вильямс, 2017. — Примеч. пер.), <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod> и [https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)) (можно просто выполнить поиск в Google по слову «SOLID»).

7. Принцип единственной ответственности



Из всех принципов SOLID наиболее трудно понимаемым является принцип единственной ответственности (Single Responsibility Principle, SRP). Это, вероятно, обусловлено выбором названия, недостаточно точно соответствующего сути. Услышав это название, многие программисты решают:

оно означает, что каждый модуль должен отвечать за что-то одно.

Самое интересное, что такой принцип *действительно* существует. Он гласит: *функция* должна делать что-то одно и только одно. Этот принцип мы используем, когда делим большие функции на меньшие, то есть на более низком уровне. Но он не является одним из принципов SOLID — это не принцип единственной ответственности.

Традиционно принцип единственной ответственности описывался так:

Модуль должен иметь одну и только одну причину для изменения.

Программное обеспечение изменяется для удовлетворения нужд пользователей и заинтересованных лиц. Пользователи и заинтересованные лица как раз и есть *та самая «причина для изменения*, о которой говорит принцип. Фактически принцип можно перефразировать так:

Модуль должен отвечать за одного и только за одного пользователя или заинтересованное лицо.

К сожалению, слова «пользователь» и «заинтересованное лицо» не совсем правильно использовать здесь, потому что одного и того же изменения системы могут желать несколько пользователей или заинтересованных лиц. Более правильным выглядит понятие группы, состоящей из одного или нескольких лиц, желающих данного изменения. Мы будем называть такие группы *акторами* (actor).

Соответственно, окончательная версия принципа единственной ответственности выглядит так:

Модуль должен отвечать за одного и только за одного актора.

Теперь определим, что означает слово «модуль». Самое простое определение — файл с исходным кодом. В большинстве случаев это определение можно принять. Однако некоторые языки среды разработки не используют исходные файлы для хранения кода. В таких случаях модуль — это просто связный набор функций и структур данных.

Слово «связный» подразумевает принцип единственной ответственности. Связность — это сила, которая связывает код, ответственный за единственного актора.

Пожалуй, лучший способ понять суть этого принципа — исследовать признаки его нарушения.

Признак 1: непреднамеренное дублирование

Мой любимый пример — класс Employee из приложения платежной ведомости. Он имеет три метода: calculatePay(), reportHours() и save() (рис. 7.1).

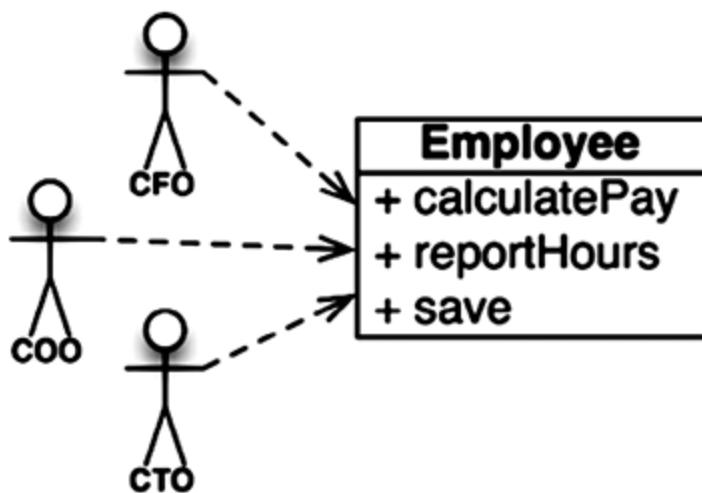


Рис. 7.1. Класс Employee

Этот класс нарушает принцип единственной ответственности, потому что три его метода отвечают за три разных актора.

- Реализация метода `calculatePay()` определяется бухгалтерией.
- Реализация метода `reportHours()` определяется и используется отделом по работе с персоналом.
- Реализация метода `save()` определяется администраторами баз данных.

Поместив исходный код этих трех методов в общий класс `Employee`, разработчики объединили перечисленных акторов. В результате такого объединения действия сотрудников бухгалтерии могут затронуть что-то, что требуется сотрудникам отдела по работе с персоналом.

Например, представьте, что функции `calculatePay()` и `reportHours()` используют общий алгоритм расчета не сверхурочных часов. Представьте также, что разработчики, старающиеся не дублировать код, поместили реализацию этого алгоритма в функцию с именем `regularHours()` (рис. 7.2).

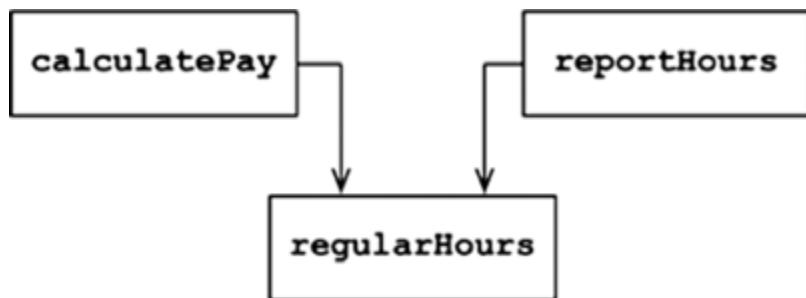


Рис. 7.2. Общий алгоритм

Теперь вообразите, что сотрудники бухгалтерии решили немного изменить алгоритм расчета не сверхурочных часов. Сотрудники отдела по работе с персоналом были бы против такого изменения, потому что вычисленное время они используют для других целей.

Разработчик, которому было поручено внести изменение, заметил, что функция `regularHours()` вызывается методом `calculatePay()`, но, к сожалению, не заметил, что она также вызывается методом `reportHours()`.

Разработчик внес требуемые изменения и тщательно протестировал результат. Сотрудники бухгалтерии проверили и подтвердили, что обновленная функция действует в соответствии с их пожеланиями, после чего измененная версия системы была развернута.

Разумеется, сотрудники отдела по работе с персоналом не знали о произошедшем и продолжали использовать отчеты, генерируемые функцией `reportHours()`, но теперь содержащие неправильные цифры. В какой-то момент проблема вскрылась, и сотрудники отдела по работе с персоналом разом побледнели от ужаса, потому что ошибочные данные обошлись их бюджету в несколько миллионов долларов.

Все мы видели нечто подобное. Эти проблемы возникают из-за того, что мы вводим в работу код, от которого зависят разные акторы. Принцип единственной ответственности требует разделять код, от которого зависят разные акторы.

Признак 2: слияния

Слияния — обычное дело для исходных файлов с большим количеством разных методов. Эта ситуация особенно вероятна, если эти методы отвечают за разных акторов.

Например, представим, что коллектив администраторов баз данных решил внести простое исправление в схему таблицы `Employee`. Представим также, что сотрудники отдела по работе с персоналом пожелали немного изменить формат отчета, возвращаемого функцией `reportHours()`.

Два разных разработчика, возможно, из двух разных команд, извлекли класс `Employee` из репозитория и внесли изменения. К сожалению, их изменения оказались несовместимыми. В результате потребовалось выполнить слияние.

Я думаю, мне не нужно рассказывать вам, что всякое слияние сопряжено с некоторым риском. Современные инструменты довольно совершенны, но никакой инструмент не сможет правильно обработать все возможные варианты слияния. В итоге риск есть всегда.

В нашем примере процедура слияния поставила под удар администраторов баз данных и отдел по работе с персоналом. Вполне возможно, что риску подверглась также бухгалтерия.

Существует много других признаков, которые мы могли бы рассмотреть, но все они сводятся к изменению одного и того же исходного кода разными людьми по разным причинам.

И снова, исправить эту проблему можно, разделив код, предназначенный для обслуживания разных акторов.

Решения

Существует много решений этой проблемы. Но каждое связано с перемещением функций в разные классы.

Наиболее очевидным, пожалуй, является решение, связанное с отделением данных от функций. Три класса, как показано на рис. 7.3, используют общие данные `EmployeeData` — простую структуру без методов. Каждый класс включает только исходный код для конкретной функции. Эти три класса никак не зависят друг от друга. То есть любое непреднамеренное дублирование исключено.

Недостаток такого решения — разработчик теперь должен создавать экземпляры трех классов и следить за ними. Эта

проблема часто решается применением шаблона проектирования «Фасад» (Facade), как показано на рис. 7.4.

Класс EmployeeFacade содержит очень немного кода и отвечает за создание экземпляров трех классов и делегирование вызовов методов.

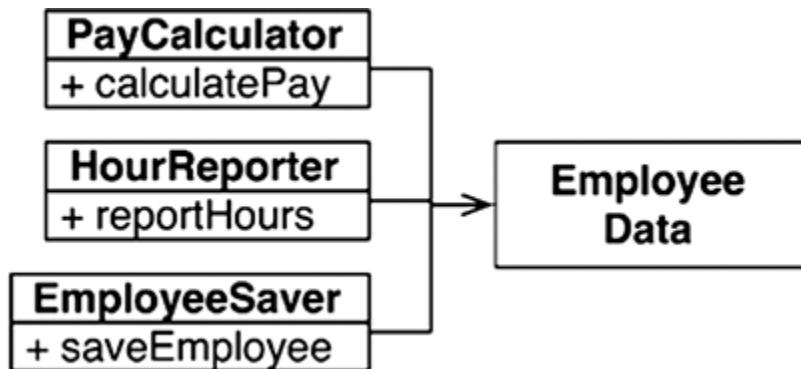


Рис. 7.3. Три класса, не зависящих друг от друга

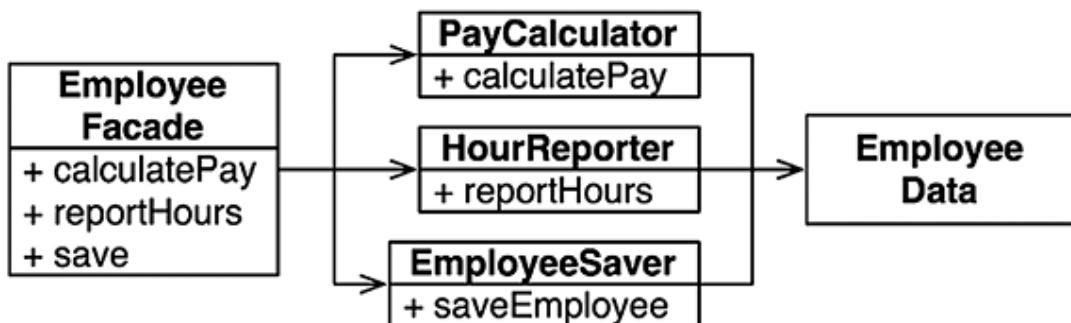


Рис. 7.4. Шаблон «Фасад»

Некоторые разработчики предпочитают держать наиболее важные бизнес-правила как можно ближе к данным. Это можно сделать, сохранив важные методы в оригинальном классе Employee, и затем использовать этот класс как фасад для низкоуровневых функций (рис. 7.5).

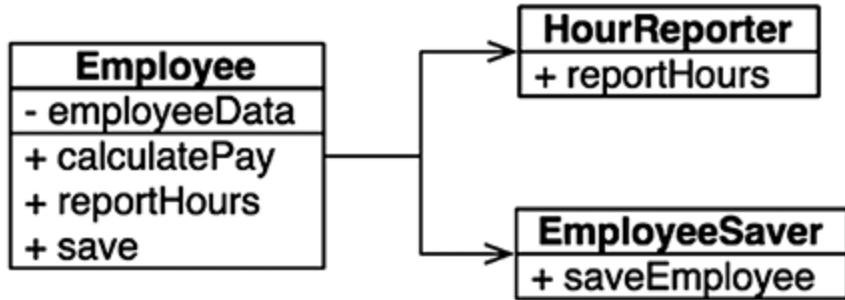


Рис. 7.5. Наиболее важные методы остаются в оригинальном классе Employee и используются как фасад для низкоуровневых функций

Вы можете возразить против такого решения на том основании, что каждый класс содержит только одну функцию. Однако в реальности такое едва ли возможно. Количество функций, необходимых для расчета зарплаты, создания отчета или сохранения данных, наверняка будет больше в любом случае. Каждый из таких классов может иметь также много приватных методов.

Каждый такой класс, содержащий подобное семейство методов, образует свою область видимости. Вне этой области никто не знает о существовании приватных членов семейства.

Заключение

Принцип единственной ответственности (Single Responsibility Principle; SRP) касается функций и классов, но он проявляется в разных формах на еще двух более высоких уровнях. На уровне компонентов он превращается в принцип согласованного изменения (Common Closure Principle; CCP), а на архитектурном уровне — в принцип оси изменения (Axis of Change), отвечающий за создание архитектурных границ. Все эти идеи мы обсудим в последующих главах.

8. Принцип открытости/закрытости



Принцип открытости/закрытости (Open-Closed Principle; OCP) был сформулирован Берtrandом Майером в 1988 году.²³ Он гласит:

Программные сущности должны быть открыты для расширения и закрыты для изменения.

Иными словами, должна иметься возможность расширять поведение программных сущностей без их изменения.

Это одна из основных причин, почему мы изучаем архитектуру программного обеспечения. Очевидно, если простое расширение требований ведет к значительным изменениям в программном обеспечении, значит, архитекторы этой программной системы потерпели сокрушительное фиаско.

Большинство студентов, изучающих проектирование программного обеспечения, признают принцип OCP как руководство по проектированию классов и модулей. Но на уровне архитектурных компонентов этот принцип приобретает еще большую значимость.

Увидеть это поможет простой мысленный эксперимент.

Мысленный эксперимент

Представьте, что у нас есть финансовая сводка. Содержимое страницы прокручивается, и отрицательные значения выводятся красным цветом.

Теперь допустим, что заинтересованные лица попросили нас представить ту же информацию в виде отчета, распечатанного на черно-белом принтере. Отчет должен быть разбит на страницы, включать соответствующие верхний и нижний колонтитулы на каждой странице и колонку меток. Отрицательные значения должны заключаться в круглые скобки.

Очевидно, что для этого придется написать новый код. Но как много старого кода придется изменить?

В программном обеспечении с хорошо проработанной архитектурой таких изменений должно быть очень немного. В идеале их вообще не должно быть.

Как? Правильно разделяя сущности, которые изменяются по разным причинам (принцип единственной ответственности), и затем правильно организуя зависимости между этими сущностями (принцип инверсии зависимостей).

Применяя принцип единственной ответственности, можно прийти к потоку данных, изображенному на рис. 8.1. Некоторая процедура анализирует финансовые данные и производит данные для отчета, которые затем форматируются двумя процедурами формирования отчетов.

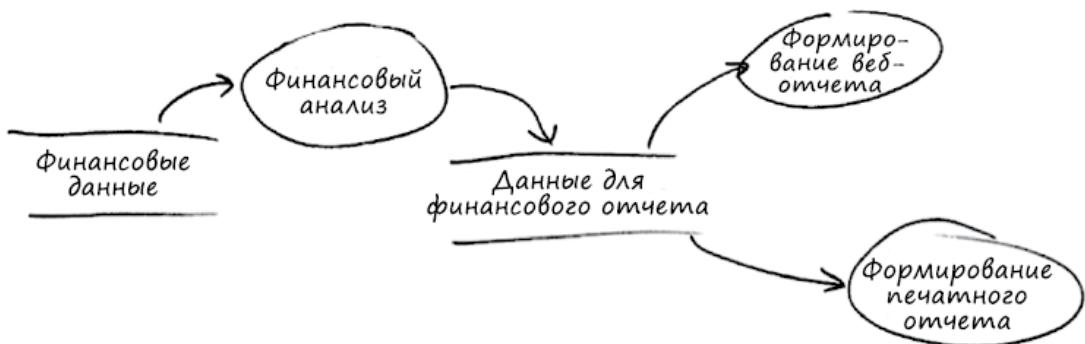


Рис. 8.1. Результат применения принципа единственной ответственности

Самое важное, что нужно понять, — в данном примере в создание отчета вовлечены две отдельные ответственности: вычисление данных для отчета и представление этих данных в форме веб-отчета или распечатанного отчета.

Сделав такое разделение, мы должны организовать зависимости в исходном коде так, чтобы изменения в одной из ответственостей не вызывали необходимости изменений в другой. Кроме того, новая организация должна гарантировать возможность расширения поведения без отмены изменений.

Этого можно добиться, выделив процессы в классы, а классы в компоненты, ограниченные двойными линиями на рис. 8.2. Компонент в левом верхнем углу на этом рисунке — *контроллер*. В правом верхнем углу — *интерактор*, или *посредник*. В правом нижнем углу — *база данных*. Наконец, в левом нижнем углу изображены четыре компонента — *презентаторы и представления*.

Классы, отмеченные символами $\langle I \rangle$, — это интерфейсы; отмеченные символами $\langle DS \rangle$ — это структуры данных (data structures). Простые стрелки соответствуют отношениям *использования*. Стрелки с треугольным наконечником соответствуют отношениям *реализации* или *наследования*.

Первое, на что следует обратить внимание, — все зависимости определены на уровне *исходного кода*. Стрелка, направленная от класса А к классу В, означает, что в исходном коде класса А упоминается имя класса В, но в коде класса В не упоминается имя класса А. Так, на рис. 8.2 диспетчер финансовых данных знает о существовании шлюза через отношение *реализации*, а шлюз финансовых данных ничего не знает о диспетчере.

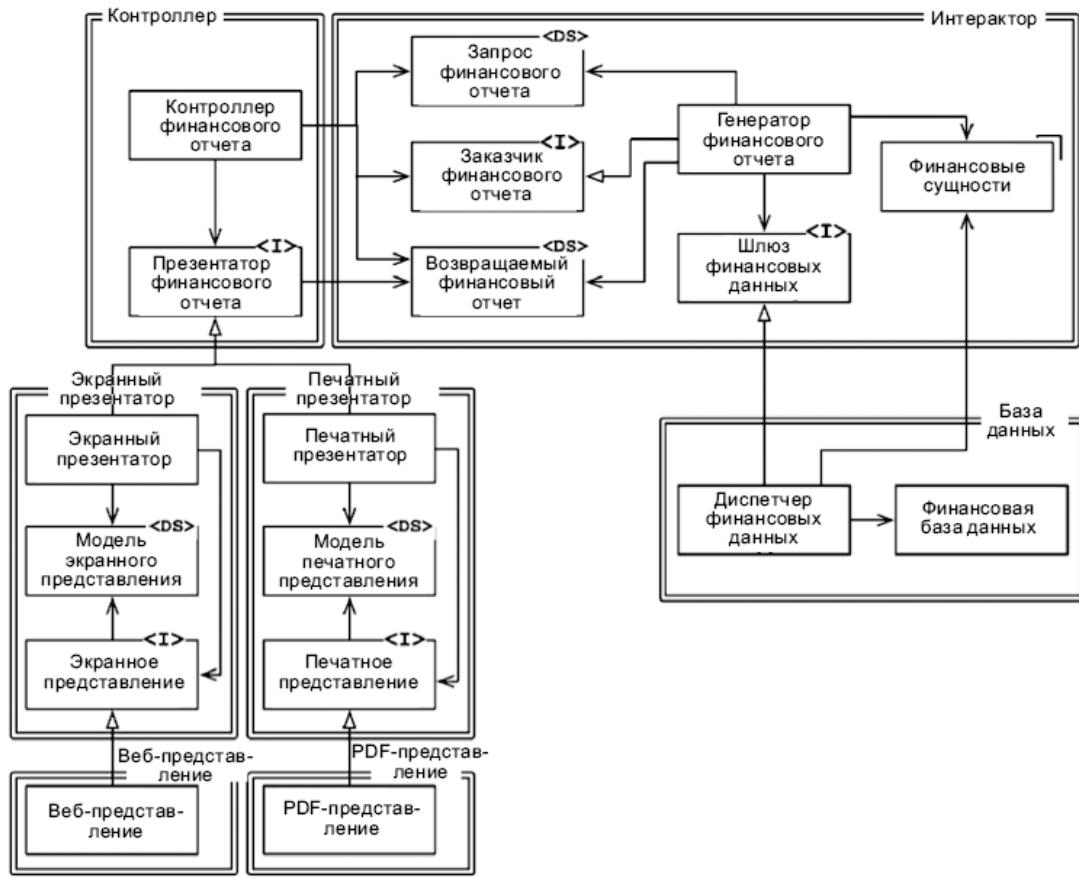


Рис. 8.2. Выделение процессов в классы и выделение классов в компоненты

Также важно отметить, что каждая двойная линия пересекается только в одном направлении. Это означает, что все отношения компонентов односторонние, как показано на графике компонентов (рис. 8.3). Эти стрелки указывают на компоненты, которые мы стремимся защитить от изменения.

Позволю себе повторить еще раз: если компонент А требуется защитить от изменений в компоненте В, компонент В должен зависеть от компонента А.

Нам нужно защитить *контроллер* от изменений в *презентаторах*. Нам нужно защитить *презентаторы* от изменений в *представлениях*. Нам нужно защитить *интерактор* от изменений в... во *всех остальных компонентах*.

Интерактор находится в позиции, лучше соответствующей принципу открытости/закрытости. Изменения в *базе данных*, или в *контроллере*, или в *презентаторах*, или в *представлениях* не должны влиять на *интерактор*.



Рис. 8.3. Отношения компонентов односторонние

Почему *интерактор* должен придерживаться такой привилегированной позиции? Потому что он реализует бизнес-правила. *Интерактор* реализует политики высшего уровня в приложении. Все другие компоненты решают второстепенные задачи. *Интерактор* решает самую главную задачу.

Несмотря на то что контроллер является не таким важным компонентом, как *интерактор*, он важнее *презентаторов* и *представлений*. А *презентаторы*, хотя и менее важные, чем контроллеры, в свою очередь, важнее *представлений*.

Обратите внимание, что в результате выстраивается иерархия защиты, основанная на понятии «уровня». *Интеракторы* занимают самый верхний уровень, поэтому они должны быть самыми защищенными. *Представления* занимают самый низкий уровень, поэтому они наименее защищены. *Презентаторы* находятся уровнем выше *представлений*, но ниже контроллера или *интерактора*.

Именно так работает принцип открытости/закрытости на архитектурном уровне. Архитекторы разделяют функциональные возможности, опираясь на то, как, почему и когда их может потребоваться изменить, и затем организуют их в иерархию компонентов. Компоненты, находящиеся на верхних уровнях в такой иерархии, защищаются от изменений в компонентах на нижних уровнях.

Управление направлением

Если вы испытали шок от схемы классов, представленной выше, взгляните на нее еще раз. Основная сложность в ней заключается в необходимости сориентировать зависимости между компонентами в правильных направлениях.

Например, интерфейс шлюза финансовых данных между генератором финансового отчета и диспетчером финансовых данных добавлен с целью обратить направление зависимости, которая иначе была бы направлена из компонента *интерактора* в компонент *базы данных*. То же относится к интерфейсу презентатора финансового отчета и двум интерфейсам *представлений*.

Сокрытие информации

Интерфейс заказчика финансового отчета служит другой цели — защитить контроллер финансового отчета от необходимости знать внутренние особенности *интерактора*. В отсутствие этого интерфейса контроллер получил бы транзитивные зависимости от финансовых сущностей.

Транзитивные (переходящие) зависимости нарушают общий принцип, согласно которому программные сущности не должны зависеть от того, что они не используют непосредственно. Мы вновь встретимся с этим принципом, когда будем обсуждать принципы разделения интерфейсов и совместного повторного использования (Common Reuse Principle; CRP).

Поэтому, даже при том, что высший приоритет имеет защита *интерактора* от изменений в контроллере, мы также должны защитить контроллер от изменений в *интеракторе*, скрыв детали реализации *интерактора*.

Заключение

Принцип открытости/закрытости — одна из движущих сил в архитектуре систем. Его цель — сделать систему легко расширяемой и обезопасить ее от влияния изменений. Эта цель достигается делением системы на компоненты и упорядочением их зависимостей в иерархию, защищающую компоненты уровнем выше от изменений в компонентах уровнем ниже.

[23](#) Bertrand Meyer. *Object Oriented Software Construction*, Prentice Hall, 1988, p. 23
(Берtrand Мейер. Объектно-ориентированное конструирование программных систем. Русская редакция, 2005. — Примеч. пер.).

9. Принцип подстановки Барбары Лисков



В 1988 году Барбара Лисков написала следующие строки с формулировкой определения подтипов.

Здесь требуется что-то вроде следующего свойства подстановки: если для каждого объекта $o1$ типа S существует такой объект $o2$ типа T , что для всех программ P , определенных в терминах T , поведение P не изменяется при подстановке $o1$ вместо $o2$, то S является подтипом T ²⁴.

Чтобы понять эту идею, известную как принцип подстановки Барбары Лисков (Liskov Substitution Principle; LSP), рассмотрим несколько примеров.

Руководство по использованию наследования

Представьте, что у нас есть класс с именем License, как показано на рис. 9.1. Этот класс имеет метод с именем calcFee(), который вызывается приложением Billing. Существует два «подтипа» класса License: PersonalLicense и BusinessLicense. Они реализуют разные алгоритмы расчета лицензионных отчислений.

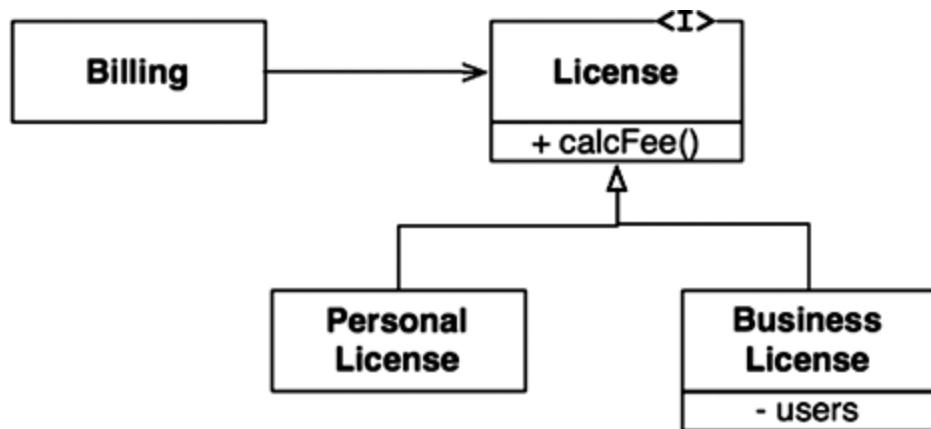


Рис. 9.1. Класс License и его производные, соответствующие принципу LSP

Этот дизайн соответствует принципу подстановки Барбары Лисков, потому что поведение приложения **Billing** не зависит от использования того или иного подтипа. Оба подтипа могут служить заменой для типа **License**.

Проблема квадрат/прямоугольник

Классическим примером нарушения принципа подстановки Барбары Лисков может служить известная проблема квадрат/прямоугольник (рис. 9.2).

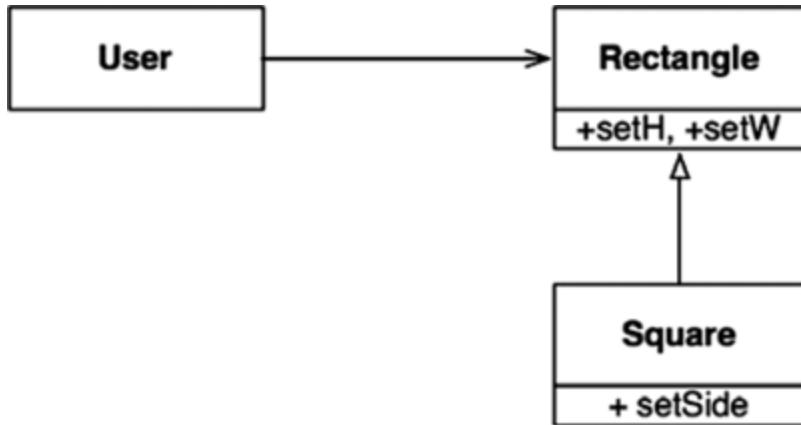


Рис. 9.2. Известная проблема квадрат/прямоугольник

В этом примере класс `Square` (представляющий квадрат) неправильно определен как подтип класса `Rectangle` (представляющего прямоугольник), потому что высоту и ширину прямоугольника можно изменять независимо; а высоту и ширину квадрата можно изменять только вместе. Поскольку класс `User` полагает, что взаимодействует с экземпляром `Rectangle`, его легко можно ввести в заблуждение, как демонстрирует следующий код:

```

Rectangle r = ...
r.setW(5);
r.setH(2);
assert(r.area() == 10);
  
```

Если на место `...` подставить код, создающий экземпляр `Square`, тогда проверка `assert` потерпит неудачу. Единственный способ противостоять такому виду нарушений принципа LSP — добавить в класс `User` механизм (например, инструкцию `if`), определяющий ситуацию, когда прямоугольник фактически является квадратом. Так как поведение `User` зависит от используемых типов, эти типы не являются заменяемыми (совместимыми).

LSP и архитектура

На заре объектно-ориентированной революции принцип LSP рассматривался как руководство по использованию наследования, как было показано в предыдущих разделах. Но со временем LSP был преобразован в более широкий принцип проектирования программного обеспечения, который распространяется также на интерфейсы и реализации.

Подразумеваемые интерфейсы могут иметь множество форм. Это могут быть интерфейсы в стиле Java, реализуемые несколькими классами. Или это может быть группа классов на языке Ruby, реализующих методы с одинаковыми сигнатурами. Или это может быть набор служб, соответствующих общему интерфейсу REST.

Во всех этих и многих других ситуациях применим принцип LSP, потому что существуют пользователи, зависящие от четкого определения интерфейсов и замещаемости их реализаций.

Лучший способ понять значение LSP с архитектурной точки зрения — посмотреть, что случится с архитектурой системы при нарушении принципа.

Пример нарушения LSP

Допустим, что мы взялись за создание приложения, объединяющего несколько служб, предоставляющих услуги такси. Клиенты, как предполагается, будут использовать наш веб-сайт для поиска подходящего такси, независимо от принадлежности к той или иной компании. Как только клиент подтверждает заказ, наша система передает его выбранному такси, используя REST-службу.

Теперь предположим, что URI службы является частью информации, хранящейся в базе данных водителей. Выбрав водителя, подходящего для клиента, наша система извлекает URI из записи с информацией о водителе и использует ее для передачи заказа этому водителю.

Допустим, что для водителя с именем Bob адрес URI отправки заказа выглядит так:

```
purplecab.com/driver/Bob
```

Наша система добавит в конец этого URI информацию о заказе и пошлет его методом PUT:

```
purplecab.com/driver/Bob  
/pickupAddress/24 Maple St.  
/pickupTime/153  
/destination/ORD
```

Это явно означает, что все службы должны соответствовать общему интерфейсу REST. Они должны единообразно интерпретировать поля `pickupAddress`, `pickupTime` и `destination`.

Теперь предположим, что компания такси Асте наняла несколько программистов, которые ознакомились со спецификацией недостаточно внимательно. Они сократили имя поля `destination` до `dest`. Компания Асте — крупнейшая компания такси в нашем регионе, и бывшая жена президента компании Асте вышла замуж за президента нашей компании, и... В общем, вы поняли. Что может произойти с архитектурой нашей системы?

Очевидно, мы должны бы добавить особый случай. Запрос с заказом для любого водителя из Асте должен бы

конструироваться в соответствии с иным набором правил, чем для всех остальных.

Решить поставленную задачу проще всего простым добавлением инструкции `if` в модуль, занимающийся пересылкой заказов:

```
if  
(driver.getDispatchUri().startsWith("acme.com")) ...
```

Конечно, ни один архитектор, дорожащий своей репутацией, не позволил бы добавить такую конструкцию в систему. Появление слова «асме» непосредственно в коде создает возможность появления самых разных неприятностей, не говоря уже о бреши в безопасности.

Например, представьте, что компания Асме добилась большого успеха, купила компанию Purple Taxi и объединенная компания решила сменить имя и адрес веб-сайта и объединить все системы оригинальных компаний. Получается, что теперь мы должны добавить еще одну инструкцию `if` для «purple»?

Архитектор должен изолировать систему от ошибок, подобных этой, и добавить модуль, управляющий созданием команд доставки заказов в соответствии с параметрами, указанным для URI в базе данных с настройками. Настройки могли бы выглядеть как-то так:

URI	Формат команды
Acme.com	/pickupAddress/%s/pickupTime/%s/dest/%s
.	/pickupAddress/%s/pickupTime/%s/destination/%s

В результате архитектор вынужден добавить важный и сложный механизм из-за того, что интерфейсы не всех REST-

служб оказались совместимыми.

Заключение

Принцип подстановки Барбары Лисков может и должен распространяться до уровня архитектуры. Простое нарушение совместимости может вызвать загрязнение архитектуры системы значительным количеством дополнительных механизмов.

[24](#) Barbara Liskov. *Data Abstraction and Hierarchy*, SIGPLAN Notices 23, 5 (May 1988).

10. Принцип разделения интерфейсов



Происхождение названия принципа разделения интерфейсов (Interface Segregation Principle; ISP) наглядно иллюстрирует схема на рис. 10.1.

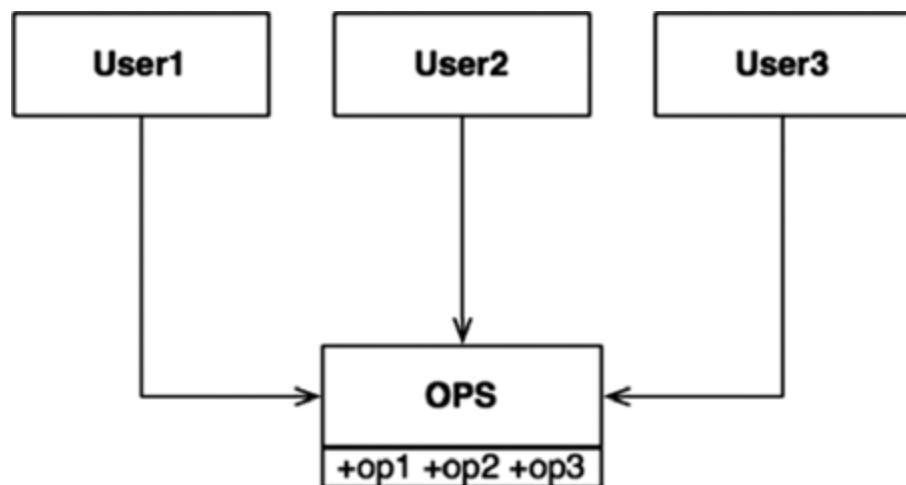


Рис. 10.1. Принцип разделения интерфейсов

В данной ситуации имеется несколько классов, пользующихся операциями в классе OPS. Допустим, что User1 использует только операцию op1, User2 – только op2 и User3 – только op3.

Теперь представьте, что OPS – это класс, написанный на таком языке, как Java. Очевидно, что в такой ситуации исходный код User1 непреднамеренно будет зависеть от op2 и op3, даже при том, что он не пользуется ими. Эта зависимость означает, что изменения в исходном коде метода op2 в классе OPS потребуют повторной компиляции и развертывания класса User1, несмотря на то что для него ничего не изменилось.

Эту проблему можно решить разделением операций по интерфейсам, как показано на рис. 10.2.

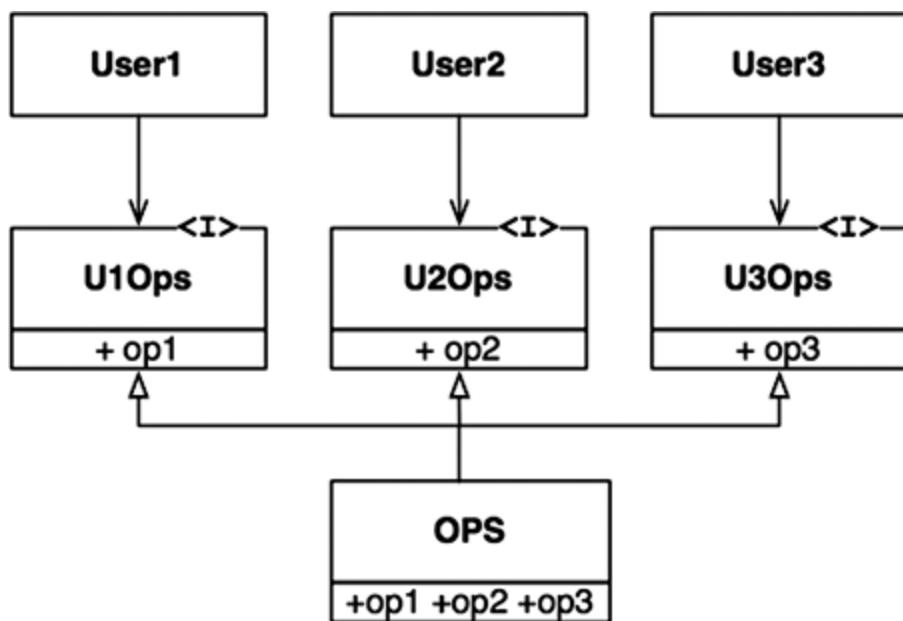


Рис. 10.2. Разделение операций

Если снова представить, что этот интерфейс реализован на языке со строгим контролем типов, таком как Java, исходный код User1 будет зависеть от U1Ops и op1, но не от OPS. То есть

изменения в OPS, которые не касаются User1, не потребуют повторной компиляции и развертывания User1.

Принцип разделения интерфейсов и язык

Очевидно, что описание выше в значительной степени зависит от типа языка. Языки со статическими типами, такие как Java, вынуждают программистов создавать объявления, которые должны импортироваться или подключаться к исходному коду пользователя как-то иначе. Именно эти инструкции подключения в исходном коде пользователя создают зависимости и вынуждают выполнять повторную компиляцию и развертывание.

В языках с динамической типизацией, таких как Ruby или Python, подобные объявления отсутствуют в исходном коде — они определяются автоматически во время выполнения. То есть в исходном коде отсутствуют зависимости, вынуждающие выполнять повторную компиляцию и развертывание. Это главная причина, почему системы на языках с динамической типизацией получаются более гибкими и с меньшим количеством строгих связей.

Этот факт ведет нас к заключению, что принцип разделения интерфейсов является проблемой языка, а не архитектуры.

Принцип разделения интерфейсов и архитектура

Если отступить на шаг назад и взглянуть на коренные мотивы, стоящие за принципом разделения интерфейсов, можно заметить более глубинные проблемы. В общем случае опасно создавать зависимости от модулей, содержащих больше, чем требуется. Это справедливо не только в отношении зависимостей в исходном коде, которые могут вынуждать

выполнять без необходимости повторную компиляцию и развертывание, но также на более высоком уровне — на уровне архитектуры.

Рассмотрим, например, действия архитектора, работающего над системой S. Он пожелал включить в систему некоторый фреймворк F. Теперь представьте, что авторы F связали его с поддержкой конкретной базы данных D. То есть S зависит от F, который зависит от D (рис. 10.3).



Рис. 10.3. Проблемная архитектура

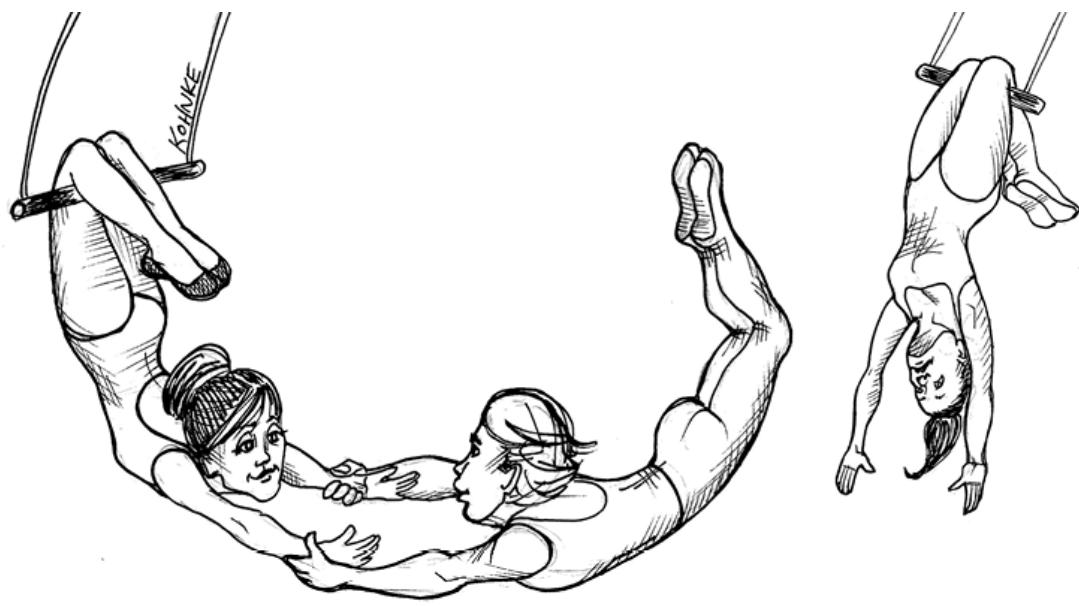
Теперь представьте, что D включает функции, которые не используются фреймворком F и, соответственно, не используются системой S. Изменения в этих функциях внутри D могут вынудить повторно развернуть F и, соответственно, повторно развернуть S. Хуже того, ошибка в одной из таких функций внутри D может спровоцировать появление ошибок в F и S.

Заключение

Из вышесказанного следует вывод: зависимости, несущие лишний груз ненужных и неиспользуемых особенностей, могут стать причиной неожиданных проблем.

Мы развернем эту мысль подробнее при обсуждении принципа совместного использования (Common Reuse Principle; CRP) в главе 13 «Связность компонентов».

11. Принцип инверсии зависимости



Принцип инверсии зависимости (Dependency Inversion Principle; DIP) утверждает, что наиболее гибкими получаются системы, в которых зависимости в исходном коде направлены на абстракции, а не на конкретные реализации.

В языках со статической системой типов, таких как Java, это означает, что инструкции `use`, `import` и `include` должны ссылаться только на модули с исходным кодом, содержащим интерфейсы, абстрактные классы и другие абстрактные объявления. Никаких зависимостей от конкретных реализаций не должно быть.

То же правило действует для языков с динамической системой типов, таких как Ruby или Python. Исходный код не должен зависеть от модулей с конкретной реализацией. Однако в этих языках труднее определить, что такое конкретный модуль. В частности, это любой модуль, в котором реализованы вызываемые функции.

Очевидно, что принять эту идею за правило практически невозможно, потому что программные системы должны зависеть от множества конкретных особенностей. Например, `String` в Java — это конкретный класс и его невозможно сделать абстрактным. Зависимости исходного кода от конкретного модуля `java.lang.string` невозможно и не нужно избегать.

С другой стороны, класс `String` очень стабилен. Изменения в этот класс вносятся крайне редко и жестко контролируются. Программистам и архитекторам не приходится беспокоиться о частых и непредсказуемых изменениях в `String`.

По этим причинам мы склонны игнорировать стабильный фундамент операционной системы и платформы, рассуждая о принципе инверсии зависимости. Мы терпим эти конкретные зависимости, потому что уверенно можем положиться на их постоянство.

Мы должны избегать зависимости от неустойчивых конкретных элементов системы. То есть от модулей, которые продолжают активно разрабатываться и претерпевают частые изменения.

Стабильные абстракции

Каждое изменение абстрактного интерфейса вызывает изменение его конкретной реализации. Изменение конкретной реализации, напротив, не всегда сопровождается изменениями и даже обычно не требует изменений в соответствующих интерфейсах. То есть интерфейсы менее изменчивы, чем реализации.

Действительно, хорошие дизайнеры и архитекторы программного обеспечения всеми силами стремятся ограничить изменчивость интерфейсов. Они стараются найти

такие пути добавления новых возможностей в реализации, которые не потребуют изменения интерфейсов. Это основа проектирования программного обеспечения.

Как следствие, стабильными называются такие архитектуры, в которых вместо зависимостей от переменчивых конкретных реализаций используются зависимости от стабильных абстрактных интерфейсов. Это следствие сводится к набору очень простых правил:

- **Не ссылайтесь на изменчивые конкретные классы.** Ссылайтесь на абстрактные интерфейсы. Это правило применимо во всех языках, независимо от устройства системы типов. Оно также накладывает важные ограничения на создание объектов и определяет преимущественное использование шаблона «*Абстрактная фабрика*».
- **Не наследуйте изменчивые конкретные классы.** Это естественное следствие из предыдущего правила, но оно достойно отдельного упоминания. Наследование в языках со статической системой типов является самым строгим и жестким видом отношений в исходном коде; следовательно, его следует использовать с большой осторожностью. Наследование в языках с динамической системой типов влечет меньшее количество проблем, но все еще остается зависимостью, поэтому дополнительная предосторожность никогда не помешает.
- **Не переопределяйте конкретные функции.** Конкретные функции часто требуют зависимостей в исходном коде. Переопределяя такие функции, вы не устраняете эти зависимости — фактически вы *наследуете* их. Для

управления подобными зависимостями нужно сделать функцию абстрактной и создать несколько ее реализаций.

- **Никогда не ссылайтесь на имена конкретных и изменчивых сущностей.** В действительности это всего лишь перефразированная форма самого принципа.

Фабрики

Чтобы соблюсти все эти правила, необходимо предусмотреть особый способ создания изменчивых объектов. Это объясняется тем, что практически во всех языках создание объектов связано с образованием зависимостей на уровне исходного кода от конкретных определений этих объектов.

В большинстве объектно-ориентированных языков, таких как Java, для управления подобными нежелательными зависимостями можно использовать шаблон «*Абстрактная фабрика*».

Рисунок 11.1 демонстрирует, как работает такая схема. Приложение Application использует конкретную реализацию ConcreteImpl через интерфейс Service. Однако приложению требуется каким-то образом создавать экземпляры ConcreteImpl. Чтобы решить эту задачу без образования зависимости от ConcreteImpl на уровне исходного кода, приложение вызывает метод makeSvc интерфейса фабрики ServiceFactory. Этот метод реализован в классе ServiceFactoryImpl, наследующем ServiceFactory. Эта реализация создает экземпляр ConcreteImpl и возвращает его как экземпляр интерфейса Service.

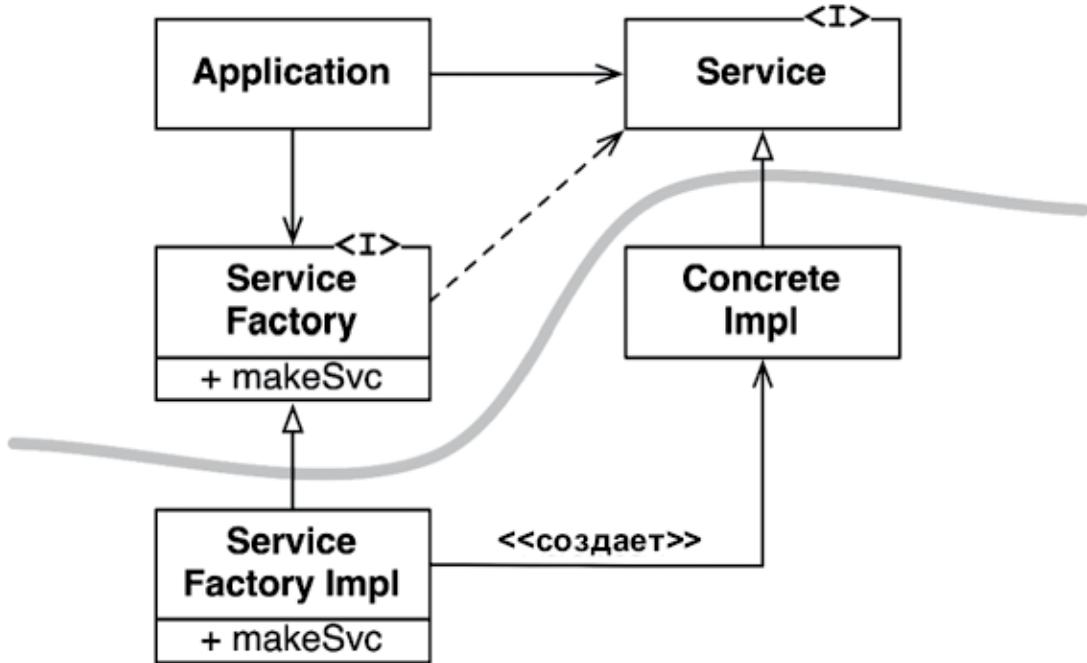


Рис. 11.1. Использование шаблона «Абстрактная фабрика» для управления зависимостями

Извилистая линия на рис. 11.1 обозначает архитектурную границу. Она отделяет абстракцию от конкретной реализации. Все зависимости в исходном коде пересекают эту границу в одном направлении — в сторону абстракции.

Извилистая линия делит систему на два компонента: абстрактный и конкретный. Абстрактный компонент содержит все высокоуровневые бизнес-правила приложения. Конкретный компонент содержит детали реализации этих правил.

Обратите внимание, что поток управления пересекает извилистую линию в направлении, обратном направлению зависимостей в исходном коде. Зависимости следуют в направлении, противоположном направлению потока управления — именно поэтому принцип получил название принципа инверсии зависимости.

Конкретные компоненты

Конкретный компонент `ConcreteImpl` на рис. 11.1 имеет единственную зависимость, то есть он нарушает принцип DIP. Это нормально. Полностью устраниить любые нарушения принципа инверсии зависимости невозможно, но их можно сосредоточить в узком круге конкретных компонентов и изолировать от остальной системы.

Большинство систем будет содержать хотя бы один такой конкретный компонент — часто с именем `main`, потому что включает функцию `main`²⁵. В схеме, изображенной на рис. 11.1, функция `main` могла бы создавать экземпляр `ServiceFactoryImpl` и сохранять ссылку на него в глобальной переменной типа `ServiceFactory`. Благодаря этому приложение `Application` сможет использовать данную глобальную переменную для обращения к фабрике.

Заключение

По мере продвижения вперед и знакомства с высокоуровневыми архитектурными принципами мы снова и снова будем сталкиваться с принципом инверсии зависимостей. Он будет самым заметным организационным принципом в наших архитектурных диаграммах. Извилистая линия на рис. 11.1 часто будет обозначать архитектурные границы в последующих главах. Зависимости будут пересекать эту извилистую линию в одном направлении, в сторону более абстрактных сущностей, и это станет для нас новым правилом, которое мы будем называть *правилом зависимостей*.

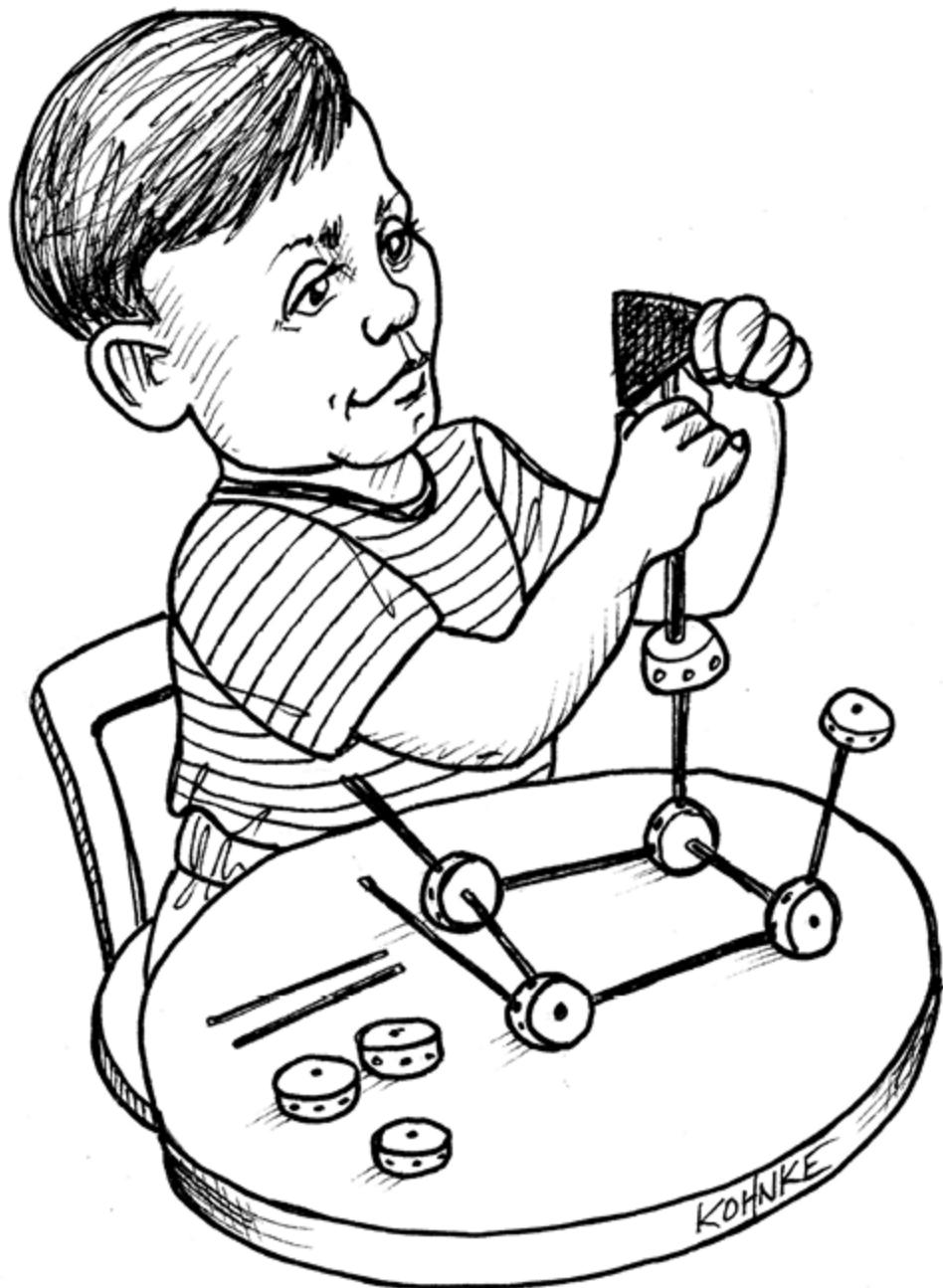
²⁵ То есть функцию, которая вызывается операционной системой в момент запуска приложения.

IV. Принципы организации компонентов

Принципы SOLID определяют, как выкладывать кирпичами стены, образующие комнаты, а принципы организации компонентов — как размещать комнаты в зданиях. Большие программные системы, подобно большим зданиям, строятся из меньших компонентов.

В части IV мы познакомимся с программными компонентами, узнаем, из каких элементов они состоят и как конструировать системы из них.

12. Компоненты



Компоненты — это единицы развертывания. Они представляют наименьшие сущности, которые можно

развертывать в составе системы. В Java — это jar-файлы. В Ruby — gem-файлы. В .Net — библиотеки DLL. В компилирующих языках — комплексы двоичных файлов. В интерпретирующих языках — комплексы файлов с исходным кодом. Во всех языках — элементарная единица развертывания.

Компоненты могут объединяться в один выполняемый файл, собираясь в один архив, например файл .war, или развертываться независимо, как отдельные плагины, загружаемые динамически, такие как файлы .jar, .dll или .exe. Но независимо от способа развертывания, правильно спроектированные компоненты всегда сохраняют возможность независимого развертывания и, соответственно, могут разрабатываться независимо.

Краткая история компонентов

На заре разработки программного обеспечения программисты сами определяли организацию памяти в своих программах. В первых строках кода часто присутствовала инструкция *origin*, объявлявшая начальный адрес в памяти для загрузки программы.

Взгляните на следующую простую программу для PDP-8. Она состоит из подпрограммы с именем GETSTR, которая принимает ввод с клавиатуры в виде строки и сохраняет его в буфер. В ней также имеется короткий модульный тест для проверки GETSTR.

*200

TLS

START, CLA

TAD BUFR

JMS GETSTR

CLA
TAD BUFR
JMS PUTSTR
JMP START

BUFR, 3000

GETSTR, 0
DCA PTR
NXTCH, KSF
JMP -1
KRB
DCA I PTR
TAD I PTR
AND K177
ISZ PTR
TAD MCR
SZA
JMP NXTCH

K177, 177
MCR, -15

Обратите внимание на команду *200 в начале программы. Она сообщает компилятору, что сгенерированный им код будет загружаться в память, начиная с адреса 200g (в восьмеричной системе счисления).

Такой способ программирования чужд современным программистам. Они редко задумываются, в какую область

памяти будет загружаться программа. Но давным-давно это было одним из первых решений, которые программист должен был принять. В ту пору программы были неперемещаемыми.

Как осуществлялся доступ к библиотечным функциям в те дни? Это иллюстрирует предыдущий пример. Программисты включали исходный код библиотек в свои программы и компилировали их как одно целое²⁶. Библиотеки хранились в исходном коде, а не в двоичном.

Проблема такого подхода в ту эпоху состояла в том, что устройства были медленными, а память стоила дорого, и поэтому ее объем был ограничен. Компиляторам требовалось выполнить несколько проходов по исходному коду, но памяти было недостаточно, чтобы уместить в ней весь исходный код. Как следствие, компилятору приходилось несколько раз читать исходный код, используя медленные устройства.

Это требовало много времени, и чем больше была библиотека, тем дольше работал компилятор. Компиляция большой программы могла длиться часами.

Чтобы сократить время компиляции, программисты отделяли исходный код библиотек от приложений. Компилировали эти библиотеки отдельно и загружали готовый двоичный код в известный адрес — например, 2000g. Они создавали таблицу символов для библиотеки и компилировали ее со своим прикладным кодом. Когда им требовалось запустить приложение, они загружали двоичный код библиотеки²⁷, а затем загружали приложение. Память была организована, как показано на рис. 12.1.

	x000-x177	x200-x377	x400-x577	x600-x777
0000-0777	[]	[]	[]	[]
1000-1777	[]	[]	[]	[]
2000-2777	[]	[]	[]	[]
3000-3777	[]	[]	[]	[]
4000-4777	[]	[]	[]	[]
5000-5777	[]	[]	[]	[]
6000-6777	[]	[]	[]	[]
7000-7777	[]	[]	[]	[]

Рис. 12.1. Организация памяти на заре программирования

Такой прием прекрасно работал, пока приложение умещалось в объем между адресами 0000₈ и 1777₈. Но если размер приложения оказывался больше отведенного адресного пространства, программисту приходилось разбивать программу на два сегмента, располагавшихся по обеим сторонам сегмента с библиотекой (рис. 12.2).

	x000-x177	x200-x377	x400-x577	x600-x777
0000-0777	[]	[]	[]	[]
1000-1777	[]	[]	[]	[]
2000-2777	[]	[]	[]	[]
3000-3777	[]	[]	[]	[]
4000-4777	[]	[]	[]	[]
5000-5777	[]	[]	[]	[]
6000-6777	[]	[]	[]	[]
7000-7777	[]	[]	[]	[]

Рис. 12.2. Деление приложения на два сегмента

Очевидно, что так не могло продолжаться вечно. Добавляя новые функции в библиотеку, программисты выходили за

границы объема, отведенного для нее, и были вынуждены выделять дополнительный сегмент (в этом примере начинающийся с адреса 7000g). Такое фрагментирование программ и библиотек продолжалось с увеличением объемов памяти в компьютерах.

Совершенно понятно, что с этим нужно было что-то делать.

Перемещаемость

Решение проблемы было найдено в создании перемещаемого двоичного кода. Идея была проста. Компилятор изменили так, чтобы он производил двоичный код, который мог перемещаться в памяти специальным загрузчиком. Этому загрузчику можно было сообщить, с какого адреса тот должен загрузить перемещаемый код. Код снабжался флагами, сообщающими загрузчику, какие части загружаемых данных нужно изменить, чтобы загрузить в выбранный адрес. Обычно это означало простое добавление начального адреса к любым ссылкам в двоичном коде.

Теперь программист мог указать загрузчику начальный адрес для загрузки библиотеки и адрес для загрузки приложения. Фактически загрузчик мог принять несколько фрагментов двоичного кода и просто загрузить их в память друг за другом, корректируя адреса ссылок в них. Это позволило программистам загружать только необходимые функции.

В компилятор также была встроена возможность, позволяющая генерировать имена функций в виде метаданных. Если программа вызывала библиотечную функцию, компилятор определял ее имя как *внешнюю ссылку*. Если программа объявляла библиотечную функцию, компилятор определял ее имя как *внешнее определение*. После

этого, зная адреса загрузки функций, загрузчик *связывал* внешние ссылки с внешними определениями.

Так родился связывающий загрузчик.

Компоновщики

Связывающий загрузчик дал программистам возможность делить свои программы на сегменты, компилируемые и загружаемые по отдельности. Такое решение оставалось жизнеспособным, пока относительно небольшие программы компоновались с относительно небольшими библиотеками. Однако в конце 1960-х — начале 1970-х годов программисты стали более честолюбивыми и их программы значительно выросли в размерах.

В результате связывающий загрузчик оказался слишком медлительным. Библиотеки хранились на устройствах с медленным доступом, таких как накопители на магнитной ленте. Даже дисковые устройства в ту пору были слишком медленными. Связывающие загрузчики должны были читать с этих медленных устройств десятки, а то и сотни библиотек в двоичном коде и корректировать внешние ссылки. С увеличением размеров программ и количества функций в библиотеках связывающему загрузчику часто требовалось больше часа, чтобы загрузить программу.

В конечном итоге загрузка и компоновка (связывание) были разделены на два отдельных этапа. Программисты выделили самый медленный этап — этап компоновки (связывания) — в отдельное приложение, получившее название *компоновщик*, или *редактор связей* (linker). В результате работы компоновщика получался скомпонованный и перемещаемый двоичный код, который загружался загрузчиком очень быстро. Это позволило программистам создавать выполняемый код,

используя медленный компоновщик, который можно было быстро загрузить в любой момент.

Затем наступили 1980-е годы. Программисты писали программы на высокоуровневых языках, таких как С. С ростом их амбиций росли и создаваемые ими программы. Программы с сотнями и тысячами строк кода стали обычным делом.

Модули с исходным кодом в файлах .c компилировались в файлы .o и затем передавались компоновщику для создания выполняемых файлов, которые можно было быстро загрузить. Компиляция каждого отдельного модуля выполнялась относительно быстро, но компиляция всех модулей порой требовала значительного времени. Для корректировки связей компоновщику могло потребоваться еще больше времени. Во многих случаях длительность этого процесса снова стала достигать часа и даже больше.

Казалось, что программисты обречены всю жизнь ходить по кругу. Все усилия, сделанные в 1960-х, 1970-х и 1980-х годах для достижения высокой скорости рабочего процесса, пошли прахом из-за амбиций программистов, создававших большие программы. Казалось, нет выхода из замкнутого круга многочасового ожидания. Этап загрузки выполнялся быстро, но этап компиляции/компоновки оставался узким местом.

Мы на практике испытывали действие закона Мерфи о размере программы:

Любая программа растет, пока не заполнит все доступное время на компиляцию и компоновку.

Но Мерфи не был единственным на этом ринге. Вскоре появился Мур²⁸, и в конце 1980-х годов они схлестнулись. Мур вышел из этой схватки победителем. Диски начали уменьшаться в размерах и стали намного быстрее. Компьютерная память стала настолько дешевой, что большую

часть данных на диске можно было уместить в кэше, в оперативной памяти. Тактовая частота процессоров выросла с 1 до 100 МГц.

К середине 1990-х годов время, необходимое на компоновку, сокращалось быстрее, чем росли наши программы. Во многих случаях время на компоновку сократилось до нескольких секунд. Для маленьких заданий вновь обрела жизнеспособность идея связывающего загрузчика.

Это была эпоха Active-X, динамических библиотек и первых файлов `.jar`. Компьютеры и периферийные устройства стали настолько быстрыми, что мы снова получили возможность выполнять компоновку (связывание) во время загрузки. Мы можем связать несколько файлов `.jar` или несколько динамических библиотек за секунды и выполнить получившуюся программу. Так родилась архитектура сменных компонентов (плагинов).

Сегодня мы обычно поставляем файлы `.jar` или динамические библиотеки как плагины к существующим приложениям. Чтобы создать новую модель для *Minecraft*, достаточно просто включить свои файлы `.jar` в определенную папку. Чтобы подключить плагин *Resharper* к среде разработки *Visual Studio*, достаточно добавить соответствующие библиотеки DLL.

Заключение

Динамически связываемые файлы, которые можно подключать во время выполнения, являются программными компонентами в наших архитектурах. Потребовалось 50 лет, но мы достигли такой степени развития, когда архитектура сменных плагинов

стала применяться по умолчанию и без титанических усилий, как когда-то.

26 Мой первый работодатель хранил в шкафу несколько десятков колод перфокарт с исходным кодом библиотек подпрограмм. Когда кто-то писал новую программу, он просто брал требуемую колоду и добавлял ее в конец колоды со своей программой.

27 В действительности на многих старых ЭВМ использовалась энергонезависимая оперативная память, которая не очищалась при выключении питания. Поэтому мы часто в течение нескольких дней использовали библиотеку, загруженную однажды.

28 Закон Мура: быстродействие компьютеров, объем памяти и компактность удваиваются каждые 18 месяцев. Этот закон удерживался с 1950-х до 2000 годов, но затем утратил силу, по крайней мере в отношении роста тактовой частоты.

13. Связность компонентов



К какому компоненту отнести тот или иной класс? Это важное решение должно приниматься в соответствии с зарекомендовавшими себя принципами разработки программного обеспечения. К сожалению, подобные решения носят особый характер и принимаются почти исключительно исходя из контекста.

В этой главе мы обсудим три принципа, определяющих связность компонентов:

- **REP:** Reuse/Release Equivalence Principle — принцип эквивалентности повторного использования и выпусков;
- **CCP:** Common Closure Principle — принцип согласованного изменения;

- CRP: Common Reuse Principle — принцип совместного повторного использования.

Принцип эквивалентности повторного использования и выпусков

Единица повторного использования есть единица выпуска.

За последнее десятилетие появилось множество инструментов управления модулями, таких как Maven, Leiningen и RVM. Эти инструменты приобрели особую важность, потому что за это же время создано огромное количество многократно используемых компонентов и библиотек компонентов. Сейчас мы живем в эпоху программного обеспечения многократного использования, когда исполнилось одно из самых древних обещаний объектно-ориентированной модели.

Принцип эквивалентности повторного использования и выпусков (Reuse/Release Equivalence Principle; REP) выглядит очевидным, по крайней мере сейчас. Люди, нуждающиеся в программных компонентах многократного использования, не смогут и не будут пользоваться компонентами, не прошедшими процесс выпуска и не получившими номер версии.

И совсем не потому, что без номера версии невозможно гарантировать совместимость всех повторно используемых компонентов. А потому, что разработчики программного обеспечения желают знать, когда появится новая версия и какие изменения в этой версии произойдут.

Нередко разработчики, знающие о приближающемся выпуске новой версии, оценивают грядущие изменения и принимают решение о продолжении использования старой

версии или переходе на новую. Поэтому в процессе выпуска создатели компонента должны рассылать соответствующие извещения и описание новой версии, чтобы пользователи могли принимать обоснованные решения о том, когда и следует ли переходить на новую версию.

С точки зрения архитектуры и дизайна этот принцип означает, что классы и модули, составляющие компонент, должны принадлежать связной группе. Компонент не может просто включать случайную смесь классов и модулей; должна быть какая-то тема или цель, общая для всех модулей.

Все это, безусловно, очевидно. Однако есть еще один аспект, возможно, не такой очевидный. Классы и модули, объединяемые в компонент, должны выпускаться вместе. Объединение их в один выпуск и включение в общую документацию с описанием этой версии должны иметь смысл для автора и пользователей.

Впрочем, говорить, что что-то должно «иметь смысл» — это все равно, что размахивать руками и пытаться выглядеть авторитетно. Это не самый лучший совет, потому что сложно точно описать, что объединяет классы и модули в единый компонент. Однако слабость совета не умаляет важности самого принципа, потому что его нарушение легко определяется по «отсутствию смысла». Если вы нарушите принцип REP, ваши пользователи узнают об этом и усомнятся в вашей компетентности как архитектора.

Слабость этого принципа с лихвой компенсируется силой двух следующих принципов. В действительности принципы CCP CRP строго определяют этот принцип, хотя и в негативном ключе.

Принцип согласованного изменения

В один компонент должны включаться классы, изменяющиеся по одним причинам и в одно время. В разные компоненты должны включаться классы, изменяющиеся в разное время и по разным причинам.

Это принцип единственной ответственности (SRP), перефразированный для компонентов. Так же, как принцип SRP, гласящий, что класс не должен иметь нескольких причин для изменения, принцип согласованного изменения (CCP) требует, чтобы компонент не имел нескольких причин для изменения.

Для большинства приложений простота сопровождения важнее возможности повторного использования. Если возникнет необходимость изменить код приложения, предпочтительнее, если все изменения будут сосредоточены в одном месте, а не разбросаны по нескольким компонентам²⁹. Если изменения ограничиваются единственным компонентом, нам потребуется развернуть только один, изменившийся компонент. Другие компоненты, не зависящие от измененного, не придется повторно проверять и развертывать.

Принцип CCP требует от нас собирать вместе все классы, которые может понадобиться изменить по одной, общей причине. Если два класса тесно связаны, физически или концептуально, настолько, что всегда будут изменяться вместе, они должны принадлежать одному компоненту. Это поможет уменьшить трудозатраты, имеющие отношение к повторному выпуску, тестированию и развертыванию программного обеспечения.

Этот принцип тесно связан с принципом открытости/закрытости (Open Closed Principle; OCP). Фактически он означает «закрытость» в смысле принципа OCP. Принцип открытости/закрытости (OCP) требует, чтобы классы были

закрыты для изменений, но открыты для расширения. Так как 100% закрытость невозможна, она должна носить скорее стратегический характер. Мы проектируем свои классы так, чтобы они были закрыты для изменений, наиболее типичных из ожидаемых по опыту.

Принцип согласованного изменения (CCP) развивает эту идею, предписывая объединять в один компонент только классы, закрытые для одного и того же вида изменений. То есть увеличивает вероятность, что изменение требований повлечет необходимость изменения минимального количества компонентов.

Сходство с принципом единственной ответственности

Как отмечалось выше, принцип согласованного изменения (CCP) есть форма принципа единственной ответственности (SRP) для компонентов. Принцип SRP требует выделять методы в разные классы, если они изменяются по разным причинам.

Принцип CCP аналогично требует выделять классы в разные компоненты, если они изменяются по разным причинам. Оба принципа можно привести к общей формуле:

Собирайте вместе все, что изменяется по одной причине и в одно время. Разделяйте все, что изменяется в разное время и по разным причинам.

Принцип совместного повторного использования

Не вынуждайте пользователей компонента зависеть от того, чего им не требуется.

Принцип совместного повторного использования (Common Reuse Principle; CRP) — еще один принцип, помогающий

определять, какие классы и модули должны включаться в компонент. Он указывает, что в компонент должны включаться классы и модули, используемые совместно.

Классы редко используются по отдельности. Обычно многократно используемые классы взаимодействуют с другими классами, являющимися частью многократно используемой абстракции. Принцип CRP указывает, что такие классы должны включаться в один компонент. Мы надеемся увидеть в компонентах классы, имеющие множественные зависимости друг от друга.

Простейшим примером могут служить класс коллекции и связанные с ним итераторы. Эти классы используются вместе, потому что они тесно связаны друг с другом. Соответственно, должны находиться в одном компоненте.

Но принцип CRP говорит не только о том, какие классы должны включаться в компонент; он также сообщает, какие классы *не должны* объединяться. Когда один компонент использует другой компонент, между ними образуется зависимость. *Использующий* компонент может нуждаться только в одном классе из *используемого* компонента, но это не ослабляет зависимости. *Использующий* компонент все так же зависит от *используемого* компонента.

Из-за этой зависимости изменение *используемого* компонента часто влечет необходимость соответствующих изменений в *использующем* компоненте. Даже если в *использующем* компоненте ничего не нужно изменять, его почти наверняка потребуется повторно скомпилировать, протестировать и развернуть. Это верно, даже если реализация *использующего* компонента совершенно не зависит от изменений в *используемом* компоненте.

То есть когда образуется зависимость от компонента, желательно, чтобы она распространялась на все классы в этом

компоненте. Иначе говоря, классы, включаемые в компонент, должны быть неотделимы друг от друга — чтобы нельзя было зависеть от одних и не зависеть от других. Иначе нам придется повторно развертывать больше компонентов, чем требуется, и тратить существенно больше усилий.

Итак, принцип совместного повторного использования (CRP) в большей степени говорит о том, какие классы *не должны* объединяться, чем какие *должны* объединяться. Принцип CRP указывает, что классы, не имеющие тесной связи, не должны включаться в один компонент.

Связь с принципом разделения интерфейсов

Принцип совместного повторного использования (CRP) является обобщенной версией принципа разделения интерфейсов (ISP). Принцип ISP советует не создавать зависимостей от классов, методы которых не используются. Принцип CRP советует не создавать зависимостей от компонентов, имеющих неиспользуемые классы.

Обобщая, эти советы можно объединить в один:

Не создавайте зависимостей от чего-либо неиспользуемого.

Диаграмма противоречий для определения связности компонентов

Возможно, вы уже заметили, что три принципа связности компонентов вступают в противоречие друг с другом. Принципы эквивалентности повторного использования (REP) и согласованного изменения (CCP) являются *включительными*: оба стремятся сделать компоненты как можно крупнее. Принцип повторного использования (CRP) — *исключительный*,

стремящийся сделать компоненты как можно мельче. Задача хорошего архитектора — разрешить это противоречие.

На рис. 13.1 изображена диаграмма противоречий³⁰, показывающая, как три принципа связности влияют друг на друга. Ребра на диаграмме описывают цену нарушения принципа на противоположной вершине.

Архитектор, уделяющий внимание только принципам REP и CRP, обнаружит, что простое изменение вовлекает слишком большое количество компонентов. С другой стороны, архитектор, уделяющий особое внимание принципам CCP и REP, вынужден будет выпускать слишком много ненужных версий.



Рис. 13.1. Диаграмма противоречий принципов связности

Хороший архитектор найдет в этом треугольнике противоречий золотую середину, отвечающую текущим

нуждам разработчиков, а также подумает об изменениях, которые могут произойти в будущем. Например, на ранних этапах разработки проекта принцип CCP намного важнее, чем REP, удобство разработки важнее удобства повторного использования.

Вообще говоря, в начале разработки наибольшую важность имеет правая сторона треугольника, когда единственной жертвой является повторное использование. Но по мере развития и интеграции в другие проекты фокус начинает смещаться влево. То есть организация компонентов в проекте может изменяться с течением времени. Это больше связано с тем, как проект разрабатывается и используется, нежели с тем, что фактически этот проект делает.

Заключение

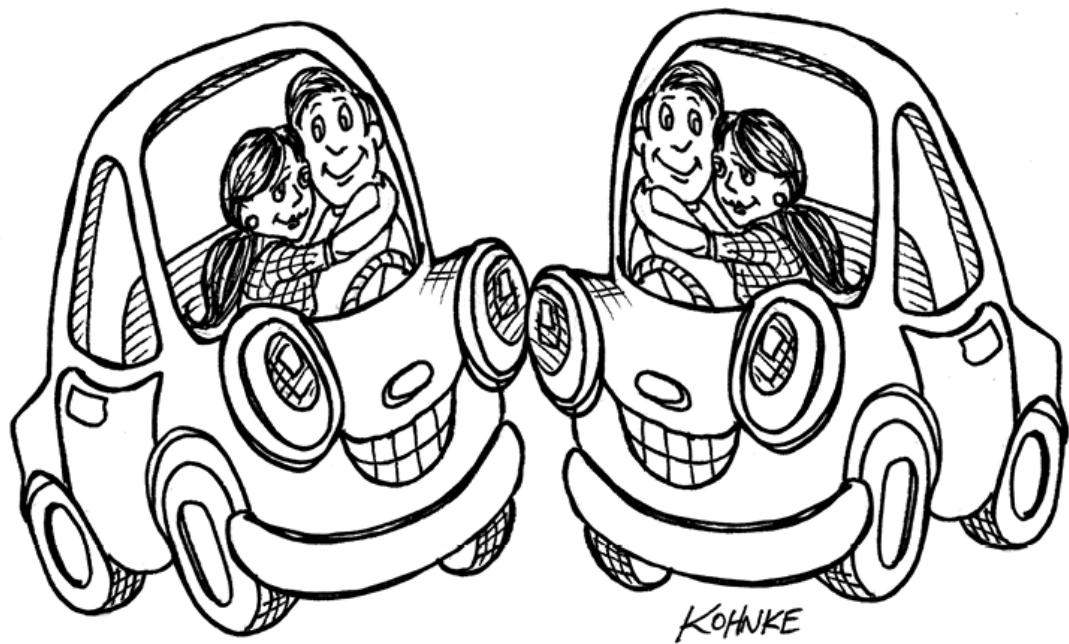
В прошлом мы смотрели на связность проще, чем предполагают принципы эквивалентности повторного использования (REP), согласованного изменения (CCP) и совместного повторного использования (CRP). Когда-то мы думали, что связность — это просто атрибут, что модуль выполняет одну и только одну функцию. Однако три принципа связности компонентов описывают намного более сложное многообразие. Выбирая классы для включения в компоненты, нужно учитывать противодействующие силы, связанные с удобством повторного использования и разработки. Поиск баланса этих сил, исходя из потребностей приложения, — непростая задача. Кроме того, баланс практически всегда постоянно смещается. То есть разбиение, считающееся удачным сегодня, может оказаться неудачным через год. Как следствие, состав компонентов почти наверняка будет

изменяться с течением времени и смещением фокуса проекта с удобства разработки к удобству повторного использования.

[29](#) См. раздел «Проблема с животными» в главе 27 «Службы: большие и малые».

[30](#) Спасибо Тиму Оттингеру за эту идею.

14. Сочетаемость компонентов



Следующие три принципа определяют правила взаимоотношений между компонентами. И снова мы сталкиваемся с противоречиями между удобством разработки и логической организацией. Силы, действующие на архитектуру структуры компонентов, носят технический, политический и непостоянный характер.

Принцип ацикличности зависимостей

Циклы в графе зависимостей компонентов недопустимы.

Бывало ли у вас так, что вы целый день проработали, заставили что-то заработать, а на следующее утро, прия на работу, обнаружили, что опять ничего не работает? Почему это произошло? Потому что кто-то задержался на работе дольше

вас и изменил что-то, от чего зависит ваш код! Я называю это «синдромом следующего утра».

«Синдром следующего утра» возникает, когда одни и те же файлы с исходным кодом правят сразу несколько разработчиков. В относительно небольших проектах, над которыми трудится малое количество разработчиков, эта проблема не доставляет особых хлопот. Но в крупных проектах с многочисленным коллективом следующее утро может превратиться в настоящий кошмар. Нередко проходят неделя за неделей, а команда все не в состоянии собрать стабильную версию проекта. Вместо этого разработчики правят и правят свой код, пытаясь заставить его работать с изменениями, сделанными кем-то другим.

За последние десятилетия было выработано два решения этой проблемы, и оба пришли из телекоммуникационной отрасли. Первое: «еженедельные сборки». И второе: соблюдение принципа ацикличности зависимостей (Acyclic Dependencies Principle; ADP).

Еженедельные сборки

Еженедельные сборки часто используются в проектах среднего размера. Это решение действует так: все разработчики работают независимо первые четыре дня в неделю. Они изменяют собственные копии кода и не заботятся об интеграции результатов своего труда в коллективную основу. Затем, в пятницу, они объединяют свои изменения и пытаются собрать систему.

Этот подход имеет замечательное преимущество, позволяя разработчикам работать в изоляции четыре дня из пяти. Недостаток, конечно же, — большие трудозатраты на интеграцию в пятницу.

К сожалению, с развитием проекта становится все сложнее завершить интеграцию в пятницу. Бремя интеграции продолжает расти, пока не начинает захватывать субботу. Нескольких таких суббот достаточно, чтобы разработчики пришли к выводу, что интеграция должна начинаться в четверг — так начало интеграции постепенно переползает ближе к середине недели.

Одновременно с уменьшением отношения продолжительности разработки к продолжительности интеграции снижается эффективность команды. В конечном итоге ситуация становится настолько удручающей, что разработчики и руководители проекта заявляют о необходимости перехода на двухнедельный цикл сборки. Это ослабляет проблему на какое-то время, но время, затрачиваемое на интеграцию, продолжает расти вместе с размерами проекта.

В конечном счете этот сценарий приводит к кризису. Для поддержания эффективности на высоком уровне график сборки должен постоянно удлиняться, но такое удлинение увеличивает риски. Интеграция и тестирование становятся все сложнее, а команда теряет преимущества, которые дает быстрая обратная связь.

Устранение циклических зависимостей

Решение этой проблемы заключается в разделении проекта на компоненты, которые могут выпускаться независимо. Компоненты становятся единицами работы, ответственность за которые можно возложить на одного разработчика или на небольшую группу. Когда разработчики добиваются работоспособности компонента, они выпускают новую версию для использования другими разработчиками. Они присваивают этой версии номер и помещают в каталог,

доступный другим разработчикам. Затем продолжают разработку компонента, изменяя свои локальные копии. А все остальные используют выпущенную версию.

Когда появляется новая версия компонента, другие команды могут выбирать — сразу же задействовать новую версию или подождать. Если принято решение подождать, они просто продолжают использовать предыдущую версию. Но как только они решат, что готовы, они начинают использовать новую версию.

В результате ни одна команда не отдается на милость другим. Изменения в одном компоненте не оказывают немедленного влияния на другие команды. Каждая команда сама решает, когда начать адаптацию своего компонента для использования новой версии другого компонента. Кроме того, интеграция происходит небольшими шагами. Нет единого момента времени, когда все разработчики должны собраться вместе и интегрировать все, что они создали.

Этот очень простой и рациональный процесс получил широкое распространение. Однако чтобы добиться успеха, вы должны управлять структурой зависимостей компонентов. *В ней не должно быть циклических зависимостей*. Если в структуре зависимостей появятся циклы, «синдрома следующего утра» не избежать.

Взгляните на диаграмму компонентов на рис. 14.1. Она демонстрирует типичную структуру компонентов, собранных в приложение. В данном случае назначение приложения не играет роли — важна сама структура зависимостей компонентов. Обратите внимание, что структура имеет вид *ориентированного (направленного) графа*. Компоненты играют роль узлов, а зависимости между ними — ориентированных ребер.

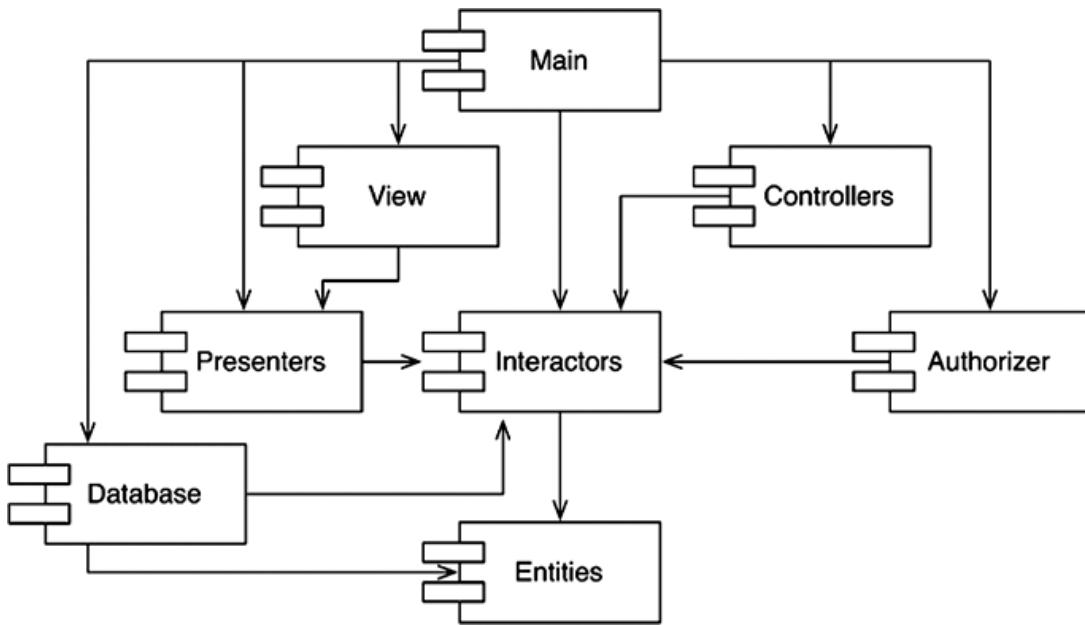


Рис. 14.1. Типичная диаграмма компонентов

Обратите внимание на одну важную особенность: с какого бы компонента вы ни начали, вы не сможете пройти по связям-зависимостям и вернуться обратно в этот же компонент. Эта структура не имеет циклов. Это *ациклический ориентированный граф* (Directed Acyclic Graph; DAG).

Теперь представьте, что произойдет, когда одна из команд выпустит новую версию компонента **Presenters**. Определить, кого затронет этот факт, нетрудно, нужно лишь проследовать по стрелкам входящих зависимостей. То есть затронуты будут компоненты **View** и **Main**. В этот момент разработчики, развивающие эти компоненты, должны решить, когда начать их интеграцию с новой версией **Presenters**.

Отметьте также, что выпуск новой версии компонента **Main** не затронет никакой другой компонент в системе. Их разработчики вообще могут не знать о существовании компонента **Main** и никак не учитывать изменения, происходящие в нем. Это замечательно. Это означает, что выпуск новой версии **Main** оказывает минимальное влияние.

Когда разработчики компонента `Presenters` пожелают протестировать его, им просто нужно собрать свою версию `Presenters` с версиями компонентов `Interactors` и `Entities`, используемыми в данный момент. Никакой другой компонент в системе им не потребуется для этого. Это замечательно. Это означает, что разработчикам `Presenters` не придется прилагать значительных усилий для подготовки к тестированию и им достаточно учесть небольшое количество переменных.

Когда придет время выпустить новую версию всей системы, процесс будет протекать снизу вверх. Сначала будет скомпилирован, протестирован и выпущен компонент `Entities`. Затем те же действия будут выполнены с компонентами `Database` и `Interactors`. За ними последуют `Presenters`, `View`, `Controllers` и затем `Authorizer`. И наконец, очередь дойдет до `Main`. Это очевидный и легко воспроизводимый процесс. Мы знаем, в каком порядке собирать систему, потому что понимаем, как связаны зависимостями отдельные ее части.

Влияние циклов в графе зависимостей компонентов

Предположим, что появление новых требований вынудило нас изменить один из классов в компоненте `Entities` так, что он стал использовать класс из компонента `Authorizer`. Например, допустим, что класс `User` из `Entities` стал использовать класс `Permissions` из `Authorizer`. В результате образовалась циклическая зависимость, как показано на рис. 14.2.

Этот цикл немедленно приводит к появлению проблем. Например, разработчики, развивающие компонент `Database`, знают, что для выпуска новой версии они должны проверить

совместимость с компонентом Entities. Но из-за образовавшегося цикла компонент Database теперь также должен быть совместим с Authorizer. Но Authorizer зависит от Interactors. Все это усложняет выпуск новой версии Database, Entities, Authorizer и Interactors фактически превращаются в один большой компонент — это означает, что всех разработчиков, развивающих эти компоненты, будет преследовать «синдром следующего утра». Они будут постоянно наступать друг другу на пятки из-за необходимости использования одних и тех же версий компонентов друг друга.

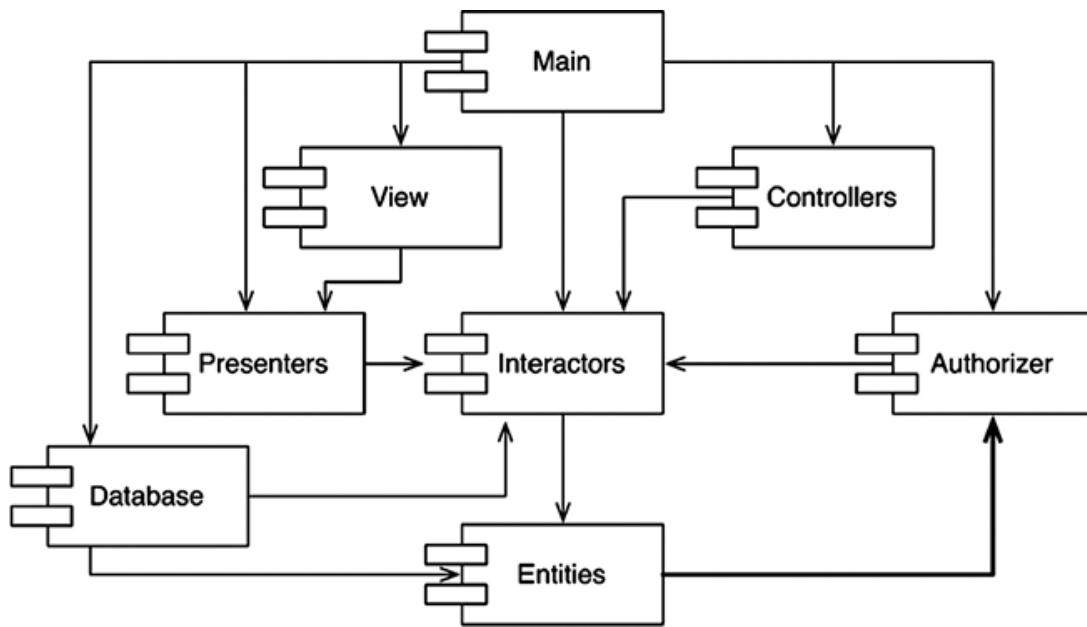


Рис. 14.2. Циклическая зависимость

Но список проблем этим не исчерпывается. Подумайте, что произойдет, когда нам потребуется протестировать компонент Entities. К нашему глубокому огорчению обнаружится, что мы должны собрать и интегрировать его с Authorizer и Interactors. Такая степень зависимости компонентов вызывает беспокойство, если она вообще допустима.

Возможно, вам уже приходилось задаваться вопросом, почему для простого модульного тестирования одного из классов приходится подключать так много библиотек и всякой другой всячины. Если бы вы копнули глубже, то наверняка бы обнаружили циклы в графе зависимостей. Такие циклы существенно усложняют изоляцию компонентов. Модульное тестирование и выпуск новой версии превращается в очень сложную задачу. Кроме того, проблемы, проявляющиеся во время сборки, начинают нарастать в геометрической прогрессии от количества модулей.

Более того, наличие циклов в графе зависимостей усложняет определение порядка сборки компонентов. И действительно, в этом случае нет правильного порядка. Это может повлечь другие неприятности в языках, таких как Java, которые извлекают объявления из скомпилированных двоичных файлов.

Разрыв цикла

Образовавшуюся циклическую зависимость всегда можно разорвать и привести график зависимостей к форме ациклического ориентированного графа (DAG). Для этого используются два основных механизма:

1. Применить принцип инверсии зависимостей (Dependency Inversion Principle; DIP). В этом случае, как показано на рис. 14.3, можно было бы создать интерфейс, определяющий методы, необходимые классу `User`, затем поместить этот интерфейс в `Entities` и унаследовать его в `Authorizer`. Это обратило бы зависимость между `Entities` и `Authorizer` и разорвало цикл.

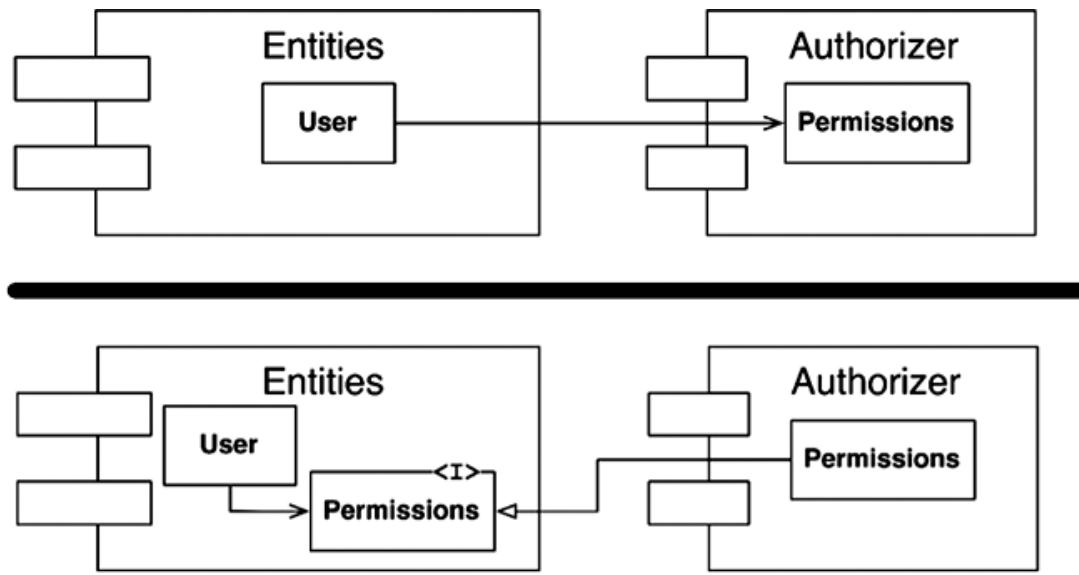


Рис. 14.3. Инверсия зависимости между Entities и Authorizer

2. Создать новый компонент, от которого зависят **Entities** и **Authorizer**. Поместить в новый компонент класс(ы), от которых они оба зависят (рис. 14.4).

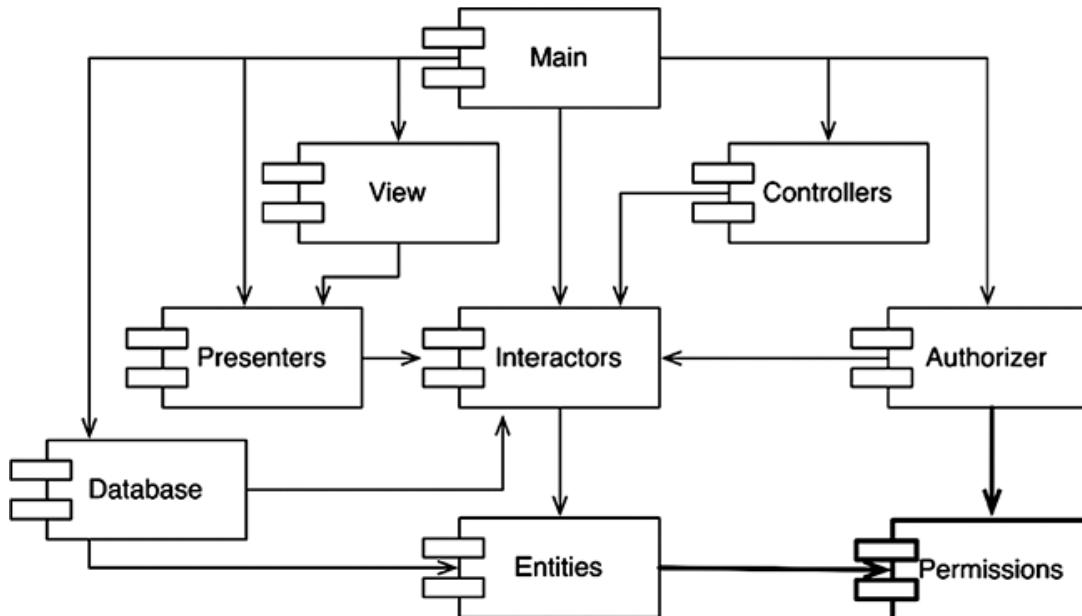


Рис. 14.4. Новый компонент, от которого зависят Entities и Authorizer

«Изменчивость»

Второе решение предполагает зависимость структуры компонентов от изменения требований. И действительно, с ростом приложения структура зависимостей компонентов растет и изменяется. Поэтому ее постоянно нужно проверять на предмет появления циклов. Когда образуются циклы, их нужно разрывать тем или иным способом. Иногда для этого приходится создавать новые компоненты, что заставляет разрастаться структуру зависимостей.

Проектирование сверху вниз

Проблемы, что мы обсудили к данному моменту, ведут к однозначному заключению: структура компонентов не может проектироваться сверху вниз. К этому выводу приходят не сразу, как только начинают проектировать систему, но это неизбежно случается с ростом и изменением системы.

Некоторые читатели могут счесть это утверждение нелогичным. Мы обычно ожидаем, что разложение на крупные составляющие, такие как компоненты, будет также соответствовать разложению по функциональному признаку.

Рассматривая крупноблочную диаграмму, такую как структура зависимостей компонентов, мы полагаем, что компоненты должны каким-то образом представлять функции системы. Но в действительности это не является непременным атрибутом диаграмм зависимостей компонентов.

Фактически диаграммы зависимостей компонентов слабо отражают функции приложения. В большей степени они являются отражением удобства сборки и сопровождения приложения. В этом главная причина, почему они не проектируются в начале разработки проекта. В этот период нет программного обеспечения, которое требуется собирать и

сопровождать, поэтому нет необходимости составлять карту сборки и сопровождения. Но с появлением все большего числа модулей на ранних стадиях реализации и проектирования возрастаёт потребность управлять зависимостями, чтобы проект можно было разрабатывать, не опасаясь «синдрома следующего утра». Кроме того, появляется желание максимально ограничить влияние изменений, поэтому мы начинаем обращать внимание на принципы единственной ответственности (SRP) и согласованного изменения (CCP) и объединять классы, которые наверняка будут изменяться вместе.

Одной из главных задач такой структуры зависимостей является изоляция изменчивости. Нам не нужны компоненты, часто изменяющиеся по самым мелким причинам и влияющие на другие компоненты, которые иначе были бы вполне стабильными. Например, косметические изменения в графическом интерфейсе не должны влиять на бизнес-правила. Добавление и изменение отчетов не должно влиять на высокоуровневые политики. Следовательно, граф зависимостей компонентов создается и формируется архитекторами для защиты стабильных и ценных компонентов от влияния изменчивых компонентов.

По мере развития приложения мы начинаем беспокоиться о создании элементов многократного пользования. На этом этапе на состав компонентов начинает влиять принцип совместного повторного использования (CRP). Наконец, с появлением циклов мы начинаем применять принцип ацикличности зависимостей (ADP), в результате начинает изменяться и разрастаться граф зависимостей компонентов.

Попытка спроектировать структуру зависимостей компонентов раньше любых классов, скорее всего, потерпит неудачу. На этом этапе мы почти ничего не знаем о

согласовании изменений, не представляем, какие элементы можно использовать многократно и почти наверняка создадим компоненты, образующие циклические зависимости. Поэтому структура зависимостей компонентов должна расти и развиваться вместе с логическим дизайном системы.

Принцип устойчивых зависимостей

Зависимости должны быть направлены в сторону устойчивости.

Дизайн не может оставаться статичным. Некоторая изменчивость все равно необходима, если предполагается сопровождать дизайн. Следуя принципу согласованного изменения (CCP), мы создаем компоненты, чувствительные к одним изменениям и невосприимчивые к другим. Некоторые из компонентов *изначально проектируются* как изменчивые. То есть мы *ожидаем*, что они будут изменяться.

Компоненты, с большим трудом поддающиеся изменению, не должны зависеть от любых изменчивых компонентов. Иначе изменчивый компонент тоже трудно будет изменять.

Это одна из превратностей программного обеспечения, когда модуль, проектировавшийся специально, чтобы упростить возможность изменений, становится сложно изменять просто из-за того, что от него зависит другой модуль. Представьте: вы не изменили ни строчки кода в своем модуле, и вдруг его стало сложно изменять. Следуя принципу устойчивых зависимостей (Stable Dependencies Principle; SDP), мы гарантируем, что модули, с трудом поддающиеся изменению, не будут зависеть от модулей, спроектированных для упрощения изменений.

Устойчивость

Что подразумевается под «устойчивостью»? Представьте себе монету, стоящую на ребре. Является ли такое ее положение устойчивым? Скорее всего, вы ответите «нет». Однако если оградить ее от вибраций и дуновений ветра, она может оставаться в таком положении сколь угодно долго. То есть устойчивость напрямую не связана с частотой изменений. Монета не изменяется, но едва ли кто-то скажет, что, стоя на ребре, она находится в устойчивом положении.

В толковом словаре говорится, что устойчивость — это «способность сохранять свое состояние при внешних воздействиях». Устойчивость связана с количеством работы, которую требуется проделать, чтобы изменить состояние. С одной стороны, монета, стоящая на ребре, находится в неустойчивом состоянии, потому что требуется приложить крошечное усилие, чтобы опрокинуть ее. С другой стороны, стол находится в очень устойчивом состоянии, потому что для его опрокидывания требуются намного более существенные усилия.

Какое отношение все это имеет к программному обеспечению? Существует множество факторов, усложняющих изменение компонента, например его размер, сложность и ясность. Но мы оставим в стороне все эти факторы и сосредоточим внимание на кое-чем другом. Есть один верный способ сделать программный компонент сложным для изменения — создать много других компонентов, зависящих от него. Компонент с множеством входящих зависимостей очень устойчив, потому что согласование изменений со всеми зависящими компонентами требует значительных усилий.

На рис. 14.5 представлена диаграмма с устойчивым компонентом X. От него зависят три других компонента, то есть имеется три веские причины не изменять его. Мы

говорим, что X несет ответственность за эти три компонента. Сам компонент X , напротив, ни от чего не зависит, то есть на него не оказывается никаких внешних воздействий, которые могли бы привести к изменению. Мы говорим, что он *независим*.

На рис. 14.6 изображен очень неустойчивый компонент Y . От него не зависит никакой другой компонент, поэтому мы говорим, что он лишен ответственности. Имеется также три компонента, от которых зависит Y , поэтому необходимость его изменения может проистекать из трех внешних источников. Мы говорим, что Y зависит.

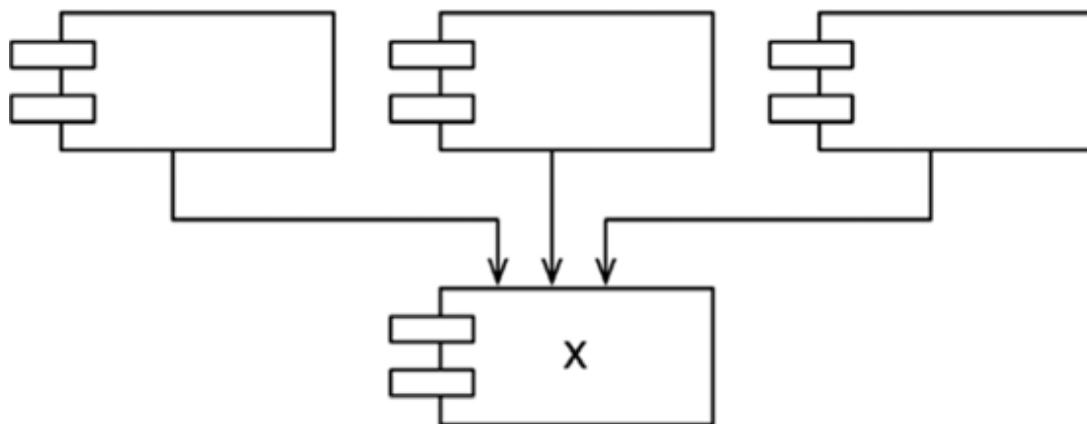


Рис. 14.5. X : устойчивый компонент

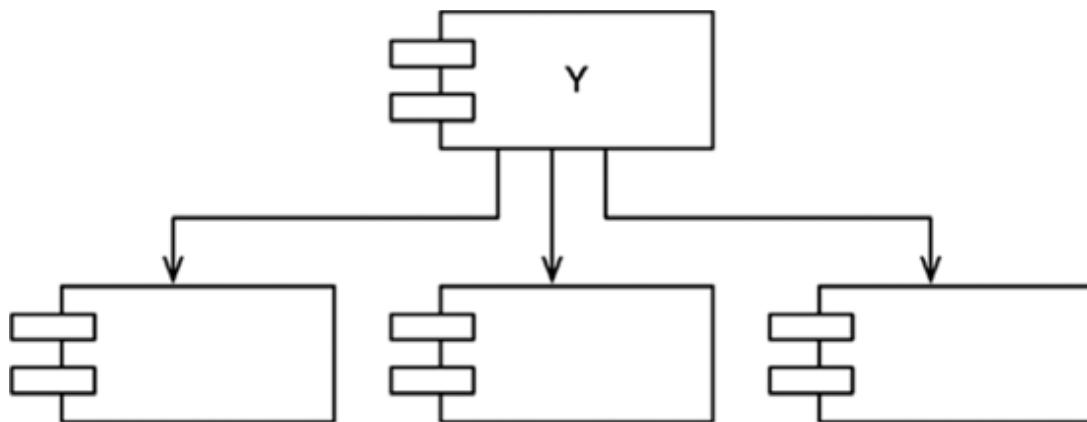


Рис. 14.6. Y : очень неустойчивый компонент

Метрики устойчивости

Как оценить устойчивость компонента? Один из способов — подсчитать количество входящих и исходящих зависимостей этого компонента. Эти числа позволяют вычислить меру его устойчивости.

- *Fan-in* (число входов): количество входящих зависимостей. Эта метрика определяет количество классов вне данного компонента, которые зависят от классов внутри компонента.
- *Fan-out* (число выходов): количество исходящих зависимостей. Эта метрика определяет количество классов внутри данного компонента, зависящих от классов за его пределами.
- I : неустойчивость: $I = \text{Fan-out} \div (\text{Fan-in} + \text{Fan-out})$. Значение этой метрики изменяется в диапазоне $[0, 1]$. $I = 0$ соответствует максимальной устойчивости компонента, $I = 1$ — максимальной неустойчивости.

Метрики *Fan-in* (число входов) и *Fan-out* (число выходов)³¹ определяются как количество классов за пределами компонентов, связанных зависимостями с классами внутри компонента. Рассмотрим пример, изображенный на рис. 14.7.

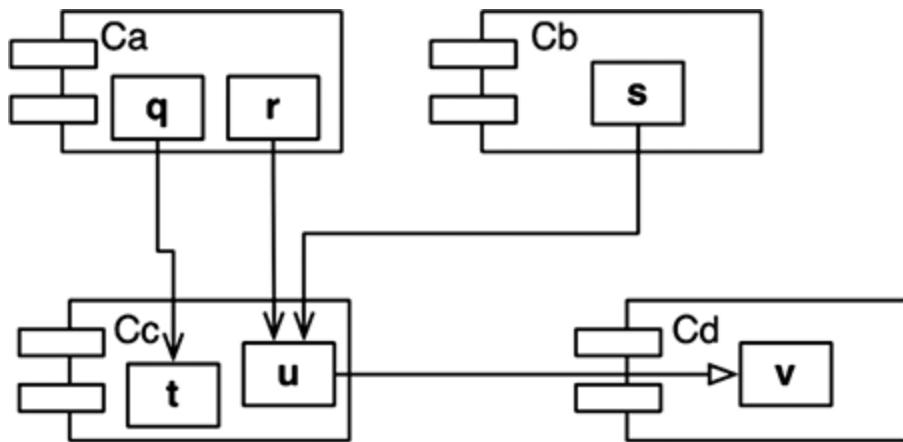


Рис. 14.7. Пример

Рассчитаем устойчивость компонента Сс. Вне компонента Сс имеется три класса, зависящих от классов внутри Сс. То есть $Fan-in = 3$. Кроме того, вне Сс имеется один класс, от которого зависят классы внутри Сс. То есть $Fan-out = 1$ и $I = 1/4$.

В C++ эти зависимости обычно представлены инструкциями `#include`. В действительности метрику I проще вычислять, когда исходный код организован так, что каждый класс хранится в отдельном файле. В Java метрику I можно вычислить, подсчитав количество инструкций `import` и полных квалифицированных имен.

Если метрика I равна 1, это означает, что никакой другой компонент не зависит от данного компонента ($Fan-in = 0$) и данный компонент зависит от других компонентов ($Fan-out > 0$). Это признак неустойчивости компонента; он безответствен и зависим. Отсутствие зависящих компонентов означает, что он не может служить причиной изменения других компонентов, а его собственная зависимость может послужить веским основанием для изменения самого компонента.

Напротив, когда метрика I равна 0, это означает, что от компонента зависят другие компоненты ($Fan-in > 0$), но сам он не зависит от других компонентов ($Fan-out = 0$). Такой компонент *ответствен и независим*. Он занимает максимальную

устойчивое положение. Зависимости от него усложняют изменение компонента, а отсутствие компонентов, от которых он зависит, означает отсутствие сил, которые могли бы заставить его измениться.

Принцип устойчивых зависимостей (SDP) говорит, что метрика I компонента должна быть больше метрик I компонентов, которые от него зависят. То есть метрики I должны уменьшаться в направлении зависимости.

Не все компоненты должны быть устойчивыми

Если все компоненты в системе будут иметь максимальную устойчивость, такую систему невозможно будет изменить. Это нежелательная ситуация. В действительности структура компонентов должна проектироваться так, чтобы в ней имелись и устойчивые, и неустойчивые компоненты. Диаграмма на рис. 14.8 демонстрирует идеальную организацию системы с тремя компонентами.

Изменяемые компоненты находятся вверху и зависят от устойчивого компонента внизу. Размещение неустойчивых компонентов в верхней части диаграммы — общепринятое и очень удобное соглашение, потому что любые стрелки, направленные вверх, ясно покажут нарушение принципа устойчивых зависимостей (и, как вы убедитесь далее, принципа ацикличности зависимостей).

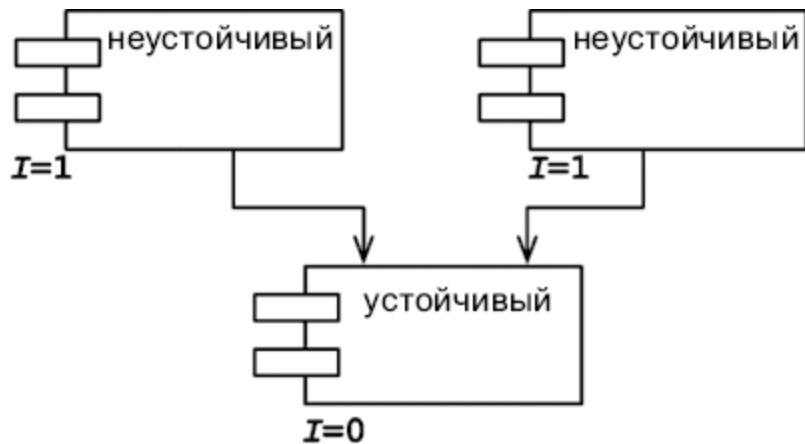


Рис. 14.8. Идеальная организация системы с тремя компонентами

Диаграмма на рис. 14.9 демонстрирует нарушение принципа SDP.

Компонент **Flexible** специально проектировался так, чтобы его было легко изменять. Предполагалось, что он будет неустойчивым. Но кто-то из разработчиков, работающих над компонентом **Stable**, создал зависимость от компонента **Flexible**. Это явное нарушение принципа SDP, потому что

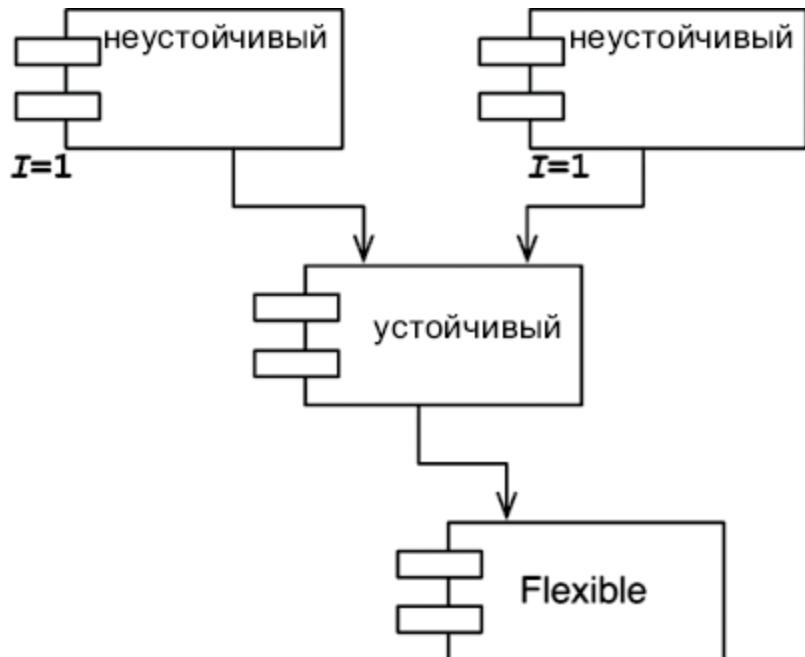


Рис. 14.9. Нарушение принципа SDP

метрика I компонента **Stable** намного меньше метрики I компонента **Flexible**. Как результат, создание такой зависимости усложнило возможное изменение компонента **Flexible**. Теперь любые изменения в компоненте **Flexible** придется согласовывать с компонентом **Stable** и всеми компонентами, зависящими от него.

Чтобы исправить проблему, нужно разорвать зависимость **Stable** от **Flexible**. Зачем нужна эта зависимость? Допустим, что в компоненте **Flexible** имеется класс **C**, который используется другим классом **U** из компонента **Stable** (рис. 14.10).



Рис. 14.10. Класс U в компоненте Stable использует класс C в компоненте Flexible

Исправить ситуацию можно, применив принцип инверсии зависимостей (DIP). Для этого определим интерфейс **US** и поместим его в компонент с именем **UServer**. Этот интерфейс должен объявлять все методы, используемые классом **U**. Затем реализуем этот интерфейс в классе **C**, как показано на рис. 14.11. Это разорвет зависимость **Stable** от **Flexible** и вынудит оба компонента зависеть от **UServer**. **UServer** очень устойчив ($I = 0$), а **Flexible** сохранит желаемую неустойчивость ($I = 1$). Теперь все зависимости простираются в сторону уменьшения I .

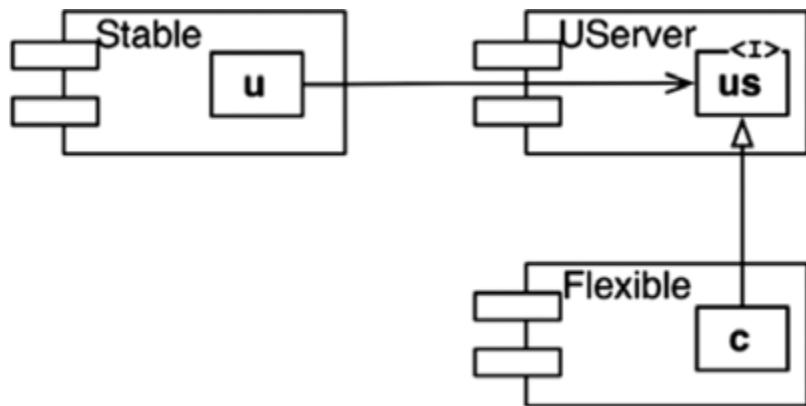


Рис. 14.11. Класс С реализует интерфейс US

Абстрактные компоненты

Кому-то может показаться странным, что мы создали компонент — в данном примере `UService`, — не содержащий ничего, кроме интерфейса. То есть компонент не содержит выполняемого кода! Однако, как оказывается, это весьма распространенная и единственная возможная тактика в языках со статической системой типов, таких как Java и C#. Такие абстрактные компоненты очень устойчивы и поэтому служат идеальной целью для зависимостей в менее устойчивых компонентах.

В языках с динамической системой типов, таких как Ruby или Python, подобные абстрактные компоненты вообще отсутствуют, так же как зависимости, которые можно было бы нацелить на них. Структура зависимостей в этих языках намного проще, потому что для инверсии зависимостей не требуется объявлять или наследовать интерфейсы.

Принцип устойчивости абстракций

Устойчивость компонента пропорциональна его абстрактности.

Куда поместить высокоуровневые правила?

Некоторые части программных систем должны меняться очень редко. Эти части представляют высокоуровневые архитектурные и другие важные решения. Никто не желает, чтобы такие решения были изменчивыми. Поэтому программное обеспечение, инкапсулирующее высокоуровневые правила, должно находиться в устойчивых компонентах ($I = 0$). Неустойчивые ($I = 1$) должны содержать только изменчивый код — код, который можно было бы легко и быстро изменить.

Но если высокоуровневые правила поместить в устойчивые компоненты, это усложнит изменение исходного кода, реализующего их. Это может сделать всю архитектуру негибкой. Как компонент с максимальной устойчивостью ($I = 0$) сделать гибким настолько, чтобы он сохранял устойчивость при изменениях? Ответ заключается в соблюдении принципа открытости/закрытости (OCP). Этот принцип говорит, что можно и нужно создавать классы, достаточно гибкие, чтобы их можно было наследовать (расширять) без изменения. Какие классы соответствуют этому принципу? *Абстрактные*.

Введение в принцип устойчивости абстракций

Принцип устойчивости абстракций (Stable Abstractions Principle; SAP) устанавливает связь между устойчивостью и абстрактностью. С одной стороны, он говорит, что устойчивый компонент также должен быть абстрактным, чтобы его устойчивость не препятствовала расширению, с другой — он говорит, что неустойчивый компонент должен быть конкретным, потому что неустойчивость позволяет легко изменять его код.

То есть стабильный компонент должен состоять из интерфейсов и абстрактных классов, чтобы его легко было расширять. Устойчивые компоненты, доступные для расширения, обладают достаточной гибкостью, чтобы не накладывать чрезмерные ограничения на архитектуру.

Принципы устойчивости абстракций (SAP) и устойчивых зависимостей (SDP) вместе соответствуют принципу инверсии зависимостей (DIP) для компонентов. Это верно, потому что принцип SDP требует, чтобы зависимости были направлены в сторону устойчивости, а принцип SAP утверждает, что устойчивость подразумевает абстрактность. То есть зависимости должны быть направлены в сторону абстрактности.

Однако принцип DIP сформулирован для классов, и в случае с классами нет никаких полутонов. Класс либо абстрактный, либо нет. Принципы SDP и SAP действуют в отношении компонентов и допускают ситуацию, когда компонент частично абстрактный или частично устойчивый.

Мера абстрактности

Мерой абстрактности компонента служит метрика A. Ее значение определяется простым отношением количества интерфейсов и абстрактных классов к общему числу классов в компоненте.

- Nc : число классов в компоненте.
- Na : число абстрактных классов и интерфейсов в компоненте.
- A : абстрактность. $A = Na \div Nc$.

Значение метрики A изменяется в диапазоне от 0 до 1. 0 означает полное отсутствие абстрактных классов в компоненте, а 1 означает, что компонент не содержит ничего, кроме абстрактных классов.

Главная последовательность

Теперь мы можем определить зависимость между устойчивостью (I) и абстрактностью (A). Для этого нарисуем график со значениями A по вертикальной оси и значениями I — по горизонтальной (рис. 14.12). Если нанести на график «хорошие» компоненты обоих видов, обнаружится, что максимально устойчивые и абстрактные находятся слева вверху, в точке с координатами $(0, 1)$, а максимально неустойчивые и конкретные — справа внизу, в точке $(1, 0)$.

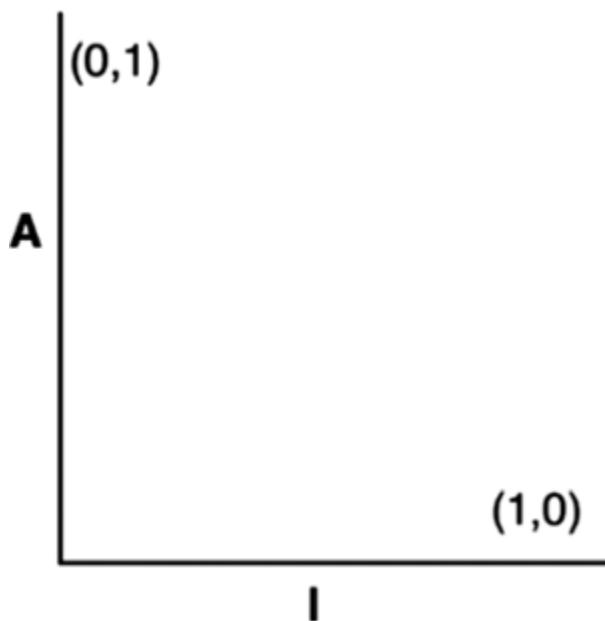


Рис. 14.12. График I/A

Но не все компоненты попадают в эти две точки, потому что компоненты имеют разные *степени* абстрактности и устойчивости. Например, очень часто один абстрактный класс

наследует другой абстрактный класс. В результате получается абстрактный класс, имеющий зависимость. Поэтому, несмотря на абстрактность, он не будет максимально устойчивым. Зависимость уменьшает устойчивость.

Так как нельзя потребовать, чтобы все компоненты находились в двух точках $(0, 1)$ или $(1, 0)$, мы должны предположить, что на графике A/I имеется некоторое множество точек, определяющих оптимальные позиции для компонентов. Вывести это множество можно, определив области, где компоненты *не должны* находиться, — иными словами, определив зоны исключения (рис. 14.13).

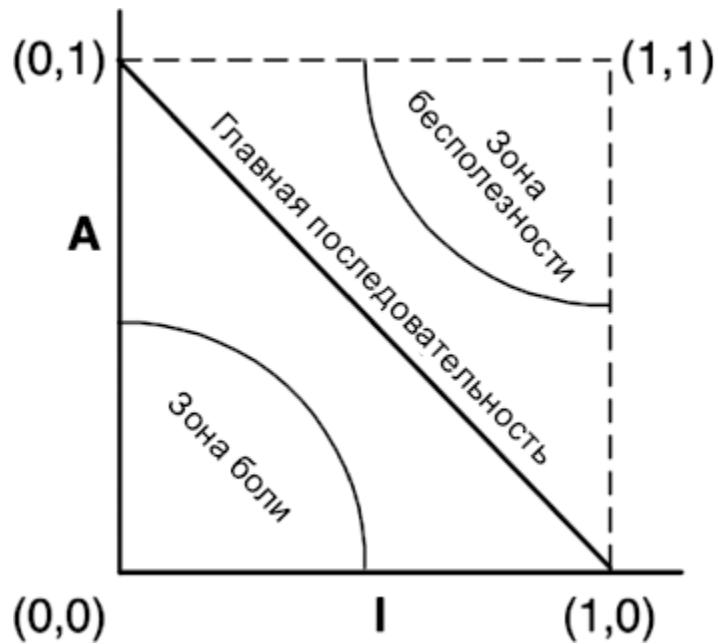


Рис. 14.13. Зоны исключения

Зона боли

Рассмотрим компонент в точке $(0, 0)$. Это очень устойчивый и конкретный компонент. Такие компоненты нежелательны, потому что слишком жесткие. Их нельзя расширить, потому что они неабстрактные, и очень трудно изменить из-за

большой устойчивости. Поэтому правильно спроектированные компоненты обычно не должны находиться рядом с точкой $(0, 0)$. Область вокруг точки $(0, 0)$ — это зона исключения, которую называют *зоной боли*.

Некоторые программные сущности действительно попадают в зону боли. Примером может служить схема базы данных. Схемы баз данных печально известны своей изменчивостью, до предела конкретны и к ним тянутся множество зависимостей. Это одна из причин, почему так сложно управлять интерфейсом между объектно-ориентированными приложениями и базами данных и почему изменение схемы обычно связано с большой болью.

Другой пример программного обеспечения, лежащего поблизости от точки $(0, 0)$ — конкретная библиотека вспомогательных функций. Хотя такая библиотека имеет метрику I со значением 1, в действительности она может быть очень негибкой. Возьмем для примера компонент `String`. Даже при том, что все классы в нем конкретны, он используется настолько широко, что его изменение может породить хаос. Поэтому `String` — негибкий.

Негибкие компоненты в зоне, окружающей точку $(0, 0)$, безопасны, потому что, скорее всего, не будут изменяться. По этой причине проблемы вызывают только изменчивые программные компоненты, находящиеся в зоне боли. Чем более изменчив компонент, находящийся в зоне боли, тем больше «боли» он доставляет. Фактически изменчивость можно рассматривать как третью ось графика. С этой точки зрения на рис. 14.13 изображена самая болезненная плоскость, где изменчивость = 1.

Зона бесполезности

Рассмотрим компонент рядом с точкой $(1, 1)$. Такие компоненты также нежелательны, потому что они максимально абстрактны и не имеют входящих зависимостей. Они бесполезны. Поэтому данная область так и называется: *зона бесполезности*.

Программные сущности, находящиеся в этой области, являются своего рода осколками. Часто это оставшиеся абстрактные классы, которые так и не были реализованы. Нам иногда доводится натыкаться на них в системах, где они лежат без использования.

Компонент, находящийся глубоко в зоне бесполезности, должен содержать значительную долю таких сущностей. Очевидно, что присутствие таких бесполезных сущностей нежелательно.

Как не попасть в зоны исключения

Кажется очевидным, что наиболее изменчивые компоненты должны находиться как можно дальше от зон исключения. Точки, максимально удаленные от обеих зон, лежат на прямой, соединяющей точки $(1, 0)$ и $(0, 1)$. Я называю эту прямую *главной последовательностью*³².

Компонент, располагающийся на главной последовательности, не «слишком абстрактный» для своей устойчивости и не «слишком неустойчив» для своей абстрактности. Он не бесполезен и не доставляет особенной боли. От него зависят другие компоненты в меру его абстрактности, и сам он зависит от других в меру конкретности.

Самыми желательными позициями для компонента являются конечные точки главной последовательности. Хорошие архитекторы стремятся разместить подавляющее

большинство компонентов в этих точках. Однако, по моему опыту, в большой системе всегда найдется несколько компонентов, недостаточно абстрактных и недостаточно устойчивых. Такие компоненты обладают великолепными характеристиками, когда располагаются *на* или *вблизи* главной последовательности.

Расстояние до главной последовательности

Мы подошли к последней нашей метрике. Коль скоро желательно, чтобы компонент располагался на или вблизи главной последовательности, можно определить метрику, выражющую удаленность компонента от идеала.

- D^{33} : расстояние. $D = |A+I-1|$. Эта метрика принимает значения из диапазона $[0, 1]$. Значение 0 указывает, что компонент находится прямо на главной последовательности. Значение 1 сообщает, что компонент располагается на максимальном удалении от главной последовательности.

Взяв на вооружение эту метрику, можно исследовать весь дизайн на близость к главной последовательности. Метрику D можно вычислить для любого компонента. Любой компонент со значением метрики D , далеким от нуля, требует пересмотра и реструктуризации.

Также можно выполнить статистический анализ дизайна. Можно найти среднее и дисперсию всех метрик D для компонентов в нашей архитектуре. Если среднее и дисперсия близки к нулю, значит, весь дизайн находится рядом с главной последовательностью. Для дисперсии можно установить «контрольные границы» и рассматривать выход за эти границы

как признак появления компонентов, оказавшихся в «исключительной» зоне, в сравнении с остальными.

На диаграмме рассеяния, изображенной на рис. 14.14, можно видеть, что основная масса компонентов располагается вдоль главной последовательности, но некоторые отклоняются от среднего более чем на одно стандартное отклонение ($Z = 1$). На эти отклоняющиеся компоненты следует обратить особое внимание. По каким-то причинам они оказались слишком абстрактными и с малым количеством входящих зависимостей или слишком конкретными и с большим количеством входящих зависимостей.

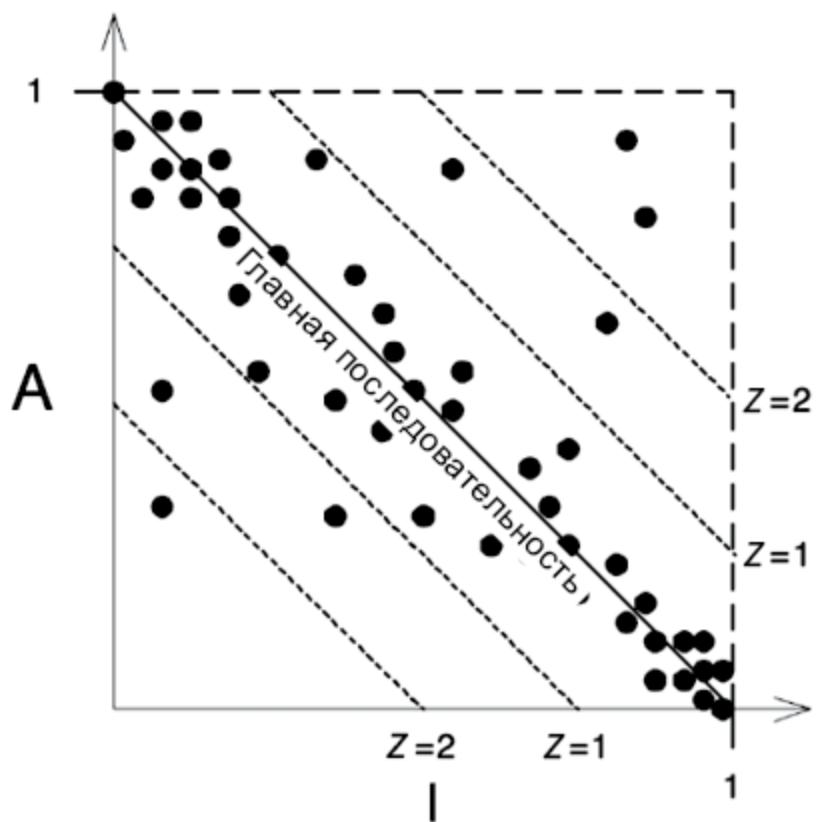


Рис. 14.14. Диаграмма рассеяния компонентов

Другой интересный способ использования метрик — построение графика изменения метрики D каждого компонента с течением времени. График на рис. 14.15

демонстрирует изменение метрики D вымышленного компонента. Можно заметить, что в последних нескольких выпусках в компонент Payroll проникли какие-то странные зависимости. На график нанесен контрольный порог для $D = 0.1$. В точке R2.1 порог оказался превышен, поэтому стоит побеспокоиться и узнать, почему компонент отдалился от главной последовательности.

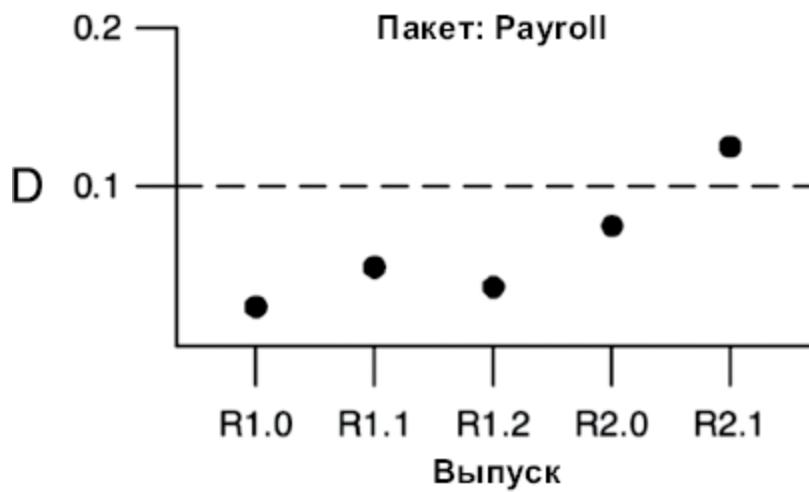


Рис. 14.15. График изменения метрики D компонента с течением времени

Заключение

Метрики управления зависимостями, описанные в этой главе, помогают получить количественную оценку соответствия структуры зависимостей и абстракции дизайна тому, что я называю «хорошим» дизайном. Как показывает опыт, есть хорошие зависимости, есть плохие. Данная оценка отражает этот опыт. Однако метрики не являются истиной в последней инстанции; это всего лишь измерения на основе произвольного стандарта. Эти метрики несовершены — в лучшем случае, но я надеюсь, что вы найдете их полезными.

31 В предыдущих публикациях я использовал для связей имена *Efferent* (центробежный) и *Afferent* (центростремительный), *Ce* и *Ca*, вместо *Fan-out* и *Fan-in* соответственно. Это был всего лишь каприз с моей стороны: мне нравилось сравнение с центральной нервной системой.

32 Автор просит у читателей снисходительности за такое высокопарное заимствование из астрономии.

33 В предыдущих публикациях я дал этой метрике имя B' . Теперь я не вижу причин продолжать эту практику.

V. Архитектура

15. Что такое архитектура



Слово «архитектура» вызывает ассоциации с такими понятиями, как власть и тайна. Оно заставляет нас воображать себе важные решения и высочайшее техническое мастерство. Архитектура программного обеспечения находится на вершине технических достижений. Думая об архитекторе программного

обеспечения, мы представляем кого-то, обладающего властью и пользующегося непререкаемым авторитетом. Какой молодой и целеустремленный разработчик не мечтает стать архитектором?

Но что такое «архитектура программного обеспечения»? Что делает архитектор программного обеспечения и когда он это делает?

Прежде всего, архитектор программного обеспечения — это программист; и он продолжает оставаться таковым. Не верьте, если кто-то вам скажет, что архитекторы не занимаются программированием и решают проблемы более высокого уровня. Это не так! Архитекторы — это лучшие программисты, и они продолжают заниматься программированием, но при этом направляют других членов команды в сторону дизайна, максимизирующего продуктивность. Архитекторы могут писать не так много кода, как другие программисты, но они продолжают программировать. Они делают это, потому что не смогут справиться со своей работой, если не будут испытывать проблем, которые сами создают для других программистов.

Архитектура программной системы — это форма, которая придается системе ее создателями. Эта форма образуется делением системы на компоненты, их организацией и определением способов взаимодействий между ними.

Цель формы — упростить разработку, развертывание и сопровождение программной системы, содержащейся в ней.

Главная стратегия такого упрощения в том, чтобы как можно дольше иметь как можно больше вариантов.

Возможно, это утверждение удивило вас. Может быть, вы считали, что главной целью архитектуры программного обеспечения является правильное функционирование системы. Конечно, все мы хотим, чтобы наши системы

работали правильно, и для архитектуры это должно быть одним из высших приоритетов.

Однако архитектура системы слабо влияет на работу системы. Существует масса систем с ужасной архитектурой, которые прекрасно работают. Их проблемы кроются не в функционировании; чаще они возникают в процессе развертывания, сопровождения и продолжительного развития.

Это не означает, что архитектура не играет никакой роли в поддержании правильного функционирования системы. Эта роль присутствует и, безусловно, имеет важнейшее значение. Но она носит скорее пассивный, косметический характер. Есть несколько вариантов *поведения*, которые архитектура системы может оставить открытыми.

Главное предназначение архитектуры — поддержка жизненного цикла системы. Хорошая архитектура делает систему легкой в освоении, простой в разработке, сопровождении и развертывании. Конечная ее цель — минимизировать затраты на протяжении срока службы системы и максимизировать продуктивность программиста.

Разработка

Программной системе, которую трудно развивать, едва ли стоит рассчитывать на долгую и здоровую жизнь. Поэтому архитектура системы должна делать ее простой в развитии для тех, кто ее разрабатывает.

Разные организации коллективов разработчиков предполагают разные архитектурные решения. С одной стороны, маленькая команда из пяти разработчиков может эффективно развивать монолитную систему без четкого определения границ между компонентами или интерфейсами. Фактически, на ранних этапах разработки такая команда

может посчитать архитектурные рамки чем-то вроде препятствия. Вероятно, в этом заключается причина, почему многие системы имеют плохую архитектуру: их архитектура не была продумана с самого начала, потому что разработчиков было немного и они не желали отвлекаться на создание общей структуры.

С другой стороны, система, разрабатываемая пятью разными командами по семь разработчиков в каждой, едва ли добьется прогресса, если не разделить ее на четко очерченные компоненты с надежными и устойчивыми интерфейсами. Если оставить без внимания другие факторы, архитектура такой системы, скорее всего, превратится в пять компонентов — по одному для каждой команды.

Такая архитектура «по одному компоненту на команду» вряд ли будет лучшей для развертывания, эксплуатации и сопровождения. Тем не менее именно к такой архитектуре будет стремиться группа команд, если они руководствуются исключительно графиком разработки.

Развертывание

Чтобы достичь высокой эффективности, программная система должна легко разворачиваться. Чем выше стоимость развертывания, тем ниже эффективность системы. Соответственно, целью архитектуры является создание системы, которую можно развернуть в одно действие.

К сожалению, стратегия развертывания редко принимается во внимание на начальных этапах разработки. В результате может получиться архитектура, обеспечивающая простоту разработки, но существенно усложняющая развертывание.

Например, на ранних порах разработчики системы могут решить использовать «архитектуру микрослужб», посчитав, что

такая архитектура значительно облегчает разработку, потому что компоненты имеют четко очерченные границы, а интерфейсы относительно устойчивы. Однако когда дело доходит до развертывания системы, они могут обнаружить, что количество микрослужб оказалось слишком велико; настройка соединений между ними и синхронизация их запуска могут оказаться неисчерпаемым источником ошибок.

Если бы архитекторы учитывали проблемы развертывания на самых ранних этапах, они, возможно, приняли бы решение создать меньшее количество служб, разбить систему на блоки, являющиеся гибридами служб и компонентов, и использовать более интегрированные средства управления взаимодействиями.

Эффективность работы

Влияние архитектуры системы на ее эффективную работу менее драматично, чем на разработку, развертывание и сопровождение. Практически любые проблемы эффективности можно решить вводом в систему нового аппаратного обеспечения без существенного влияния на ее архитектуру.

Подтверждение этому мы видим снова и снова. Системы с неэффективной архитектурой могут работать вполне эффективно за счет простого увеличения объема памяти или количества серверов. Из-за дешевизны аппаратного обеспечения и дороговизны человеческих трудозатрат архитектуры, ограничивающие эффективность работы, оказываются менее дорогостоящими, чем архитектуры, усложняющие разработку, развертывание и сопровождение.

Это не означает, что архитектура, нацеленная на эффективную работу системы, нежелательна. Она желательна!

Просто уравнение затрат больше ориентируется на разработку, развертывание и сопровождение.

Между тем существует еще одна роль, которую архитектура играет в работе системы: хорошая архитектура способствует высокой эффективности.

Можно даже сказать, что хорошая архитектура делает работу системы понятной разработчикам. Архитектура должна отражать особенности работы. Архитектура должна возводить варианты использования, особенности и ожидаемые реакции системы до уровня полноправных сущностей, которые послужат видимыми ориентирами для разработчиков. Это упростит понимание системы и, соответственно, окажет значительную помощь в разработке и сопровождении.

Сопровождение

Из всех аспектов программной системы сопровождение является самым дорогостоящим. Бесконечный парад новых особенностей и неизбежная череда дефектов и исправлений требуют огромных человеческих трудозатрат.

Основная стоимость сопровождения складывается из стоимости исследования и рисков. Затраты на исследование обусловлены необходимостью поиска мест в существующем программном обеспечении и стратегий для добавления новых особенностей или устранения дефектов. При внесении таких изменений всегда существует вероятность что-то испортить, что увеличивает стоимость рисков.

Продуманная архитектура значительно снижает эти затраты. Разделив систему на компоненты и изолировав их за устойчивыми интерфейсами, можно осветить пути к будущим особенностям и существенно уменьшить риск непреднамеренных ошибок.

Сохранение разнообразия вариантов

Как говорилось в главе 2, всякая программная система имеет две ценности: поведение и структуру. Вторая из них больше, потому что именно она делает программное обеспечение *гибким*.

Программное обеспечение было изобретено, потому что требовался способ легко и быстро менять поведение машин. Но гибкость в решающей степени зависит от формы системы, организации ее компонентов и способов взаимодействий между ними.

Основной способ сохранить гибкость программного обеспечения заключается в том, чтобы как можно дольше иметь как можно больше вариантов. Что это за варианты, которые нужно иметь? *Это детали, не имеющие значения.*

Любую программную систему можно разложить на два основных элемента: политику и детали. Политика воплощает все бизнес-правила и процедуры. Политика — вот истинная ценность системы.

Детали — это все остальное, что позволяет людям, другим системам и программистам взаимодействовать с политикой, никак не влияя на ее поведение. К ним можно отнести устройства ввода/вывода, базы данных, веб-системы, серверы, фреймворки, протоколы обмена данными и т.д.

Цель архитектора — создать такую форму для системы, которая сделает политику самым важным элементом, а детали — *не относящимися к политике*. Это позволит *откладывать и задерживать* принятие решений о деталях.

Например:

- Не нужно на ранних этапах разработки выбирать тип базы данных, потому что высокоуровневая политика не должна зависеть от этого выбора. У мудрого архитектора

высокоуровневая политика не зависит от того, какая база данных будет выбрана впоследствии: реляционная, распределенная, иерархическая или просто набор файлов.

- Не нужно на ранних этапах разработки выбирать тип веб-сервера, потому что высокоуровневая политика не должна даже знать, что она доставляется через Веб. Если высокоуровневая политика не зависит от HTML, AJAX, JSP, JSF и прочих технологий, применяемых в веб-разработке, вы сможете отложить принятие решения о выборе веб-системы до более поздних этапов. Фактически *вы даже не должны принимать решения, что система будет предоставлять свои услуги через Веб*.
- Не нужно на ранних этапах разработки приспосабливать интерфейс REST, потому что высокоуровневая политика не должна зависеть от интерфейса с внешним миром. Также не нужно приспосабливаться под архитектуру микрослужб или SOA³⁴. Высокоуровневая политика не должна зависеть от подобных деталей.
- Не нужно на ранних этапах разработки приспосабливать механизмы внедрения зависимостей, потому что высокоуровневая политика не должна зависеть от способа разрешения зависимостей.

Думаю, вы поняли. Сумев разработать высокоуровневую политику, не связывая ее с окружающими деталями, вы сможете откладывать и задерживать принятие решений об этих деталях на более поздний срок. И чем дольше вы сможете откладывать такие решения, *тем большим объемом информации вы будете обладать, чтобы сделать правильный выбор*.

Такой подход дает также возможность экспериментировать. Если уже имеется действующая часть высокоуровневой политики, не зависящая от типа базы данных, можно попробовать связать ее с разными базами данных, чтобы проверить их применимость и производительность. То же верно в отношении веб-систем, веб-фреймворков и даже самого Веб.

Чем дольше варианты будут оставаться открытыми, тем больше экспериментов можно провести, тем больше способов опробовать и тем большим объемом информации вы будете обладать, подойдя к моменту, когда принятие решения уже нельзя отложить.

Что делать, если решение уже принято кем-то другим? Что делать, если ваша компания взяла на себя обязательство перед кем-то использовать определенную базу данных, веб-сервер или фреймворк? Хороший архитектор сделает вид, что такого решения не было, и будет формировать систему так, чтобы эти решения можно было принять или изменить как можно позже.

Хороший архитектор максимизирует количество непринятых решений.

Независимость от устройства

Чтобы понять суть, вернемся ненадолго в 1960-е годы, когда компьютеры были совсем юными, а большинство программистов были математиками или инженерами из других областей (и третья или даже больше были женщинами).

В те дни допускалась масса ошибок. Конечно, в те дни мы не знали, что это ошибки. Да и не могли знать.

Одна из таких ошибок — образование прямой зависимости кода от устройств ввода/вывода. Если нам требовалось вывести

что-то на принтер, мы писали код, использующий инструкции ввода/вывода, который генерировал команды управления принтером. Наш код зависел от устройства.

Например, когда я писал программы для PDP-8, печатавшие что-то на телепринтере, я использовал примерно такой набор машинных инструкций:

```
PRTCHR, 0
TSF
JMP .-1
TLS
    JMP I PRTCHR
```

PRTCHR — это подпрограмма, печатающая один символ на телепринтере. Ячейка памяти с нулем в начале подпрограммы использовалась для хранения адреса возврата. (Не спрашивайте, что это такое.) Инструкция TSF пропускает следующую за ней инструкцию, если телепринтер готов напечатать символ. Если телепринтер занят, TSF просто передавала управление следующей за ней инструкции JMP .-1, выполняющей переход обратно на инструкцию TSF. Если телепринтер готов, TSF передавала управление инструкции TLS, которая посыпала символ в регистр А телепринтера. Затем инструкция JMP I PRTCHR возвращала управление вызывающему коду.

Первое время этот прием работал безупречно. Если нам нужно было прочитать данные с перфокарт, мы использовали код, напрямую обращающийся к устройству ввода с перфокарт. Если нужно было набить перфокарты, мы писали код, напрямую обращающийся к перфоратору. Такие программы работали замечательно. Могли ли мы знать, что это была ошибка?

Но с большими колодами перфокарт было трудно управляться. Карты могли теряться, мяться, рваться, перемешиваться или проскакивать мимо. Могли теряться отдельные карты или добавляться лишние. То есть целостность данных была большой проблемой.

Магнитные ленты решили эту проблему. Мы получили возможность переносить образы карт на ленту. Послеброса данных на ленту записи уже не могли перемещаться. Они не могли потеряться по неосторожности, а также не могли добавиться новые пустые записи. Лента оказалась гораздо надежнее. А кроме того, операции чтения/записи с лентой выполнялись намного быстрее, и намного проще стало делать резервные копии.

К сожалению, все наше программное обеспечение было написано для работы с устройствами чтения перфокарт и с перфораторами. Эти программы пришлось переписывать, чтобы они могли работать с магнитной лентой. Это была большая работа.

К концу 1960-х годов мы выучили урок и придумали, как обрести *независимость* от устройств. Операционные системы, современные в те дни, абстрагировали устройства ввода/вывода за фасадами программных функций, которые обрабатывали отдельные записи, выглядящие как перфокарты. Программы должны были обращаться к службам операционной системы, представляющим абстрактные устройства хранения единичных записей. Операторы могли сообщить операционной системе, к каким физическим устройствам должны подключаться эти абстрактные службы — устройствам чтения перфокарт, накопителям на магнитной ленте или любым другим устройствам хранения записей.

Теперь одна и та же программа могла читать и записывать данные на перфокарты или магнитные ленты без всяких

изменений. Так родился принцип открытости/закрытости (Open-Closed Principle; OCP), но в те дни он еще не имел названия.

Нежелательная почта

В конце 1960-х годов я работал в компании, которая печатала рекламные письма для своих клиентов. Клиенты присыпали магнитные ленты с записями, содержащими имена и адреса своих клиентов, а мы должны были писать программы, печатающие персонализированные рекламные объявления.

Примерно такого содержания:

Здравствуйте, мистер Мартин,

Поздравляем!

Мы выбрали ВАС из всех живущих на Вичвуд-лейн, чтобы сделать фантастическое предложение...

Клиенты присыпали нам огромные рулоны с заготовками писем, куда мы должны были впечатать имена и адреса или любой другой текст, которые передавались на магнитной ленте. Мы писали программы, читающие эти имена, адреса и другой текст с магнитной ленты и печатающие их точно в отведенных полях.

Такие рулоны весили по 500 фунтов и содержали тысячи писем. Клиенты могли присыпать сотни рулонов, и каждый приходилось печатать отдельно.

Сначала у нас была машина IBM 360, печатающая на единственном последовательном принтере. За смену мы могли напечатать несколько тысяч писем. К сожалению, печать таких писем стоила очень дорого. В те дни аренда IBM 360 обходилась в десятки тысяч долларов в месяц.

Поэтому мы настраивали операционную систему так, чтобы вместо принтера она использовала магнитную ленту. Это никак не сказывалось на наших программах, потому что они выводили данные, используя абстракции ввода/вывода операционной системы.

360-я могла заполнить ленту за десять минут или что-то около того — вполне достаточно, чтобы напечатать несколько рулонов писем. Эти ленты затем выносились из машинного зала и монтировались в приводы, подключенные к автономным принтерам. У нас их было пять штук, и эти пять принтеров работали круглые сутки без перерывов и выходных, печатая сотни тысяч рекламных писем каждую неделю.

Ценность независимости от устройства оказалась огромной! Мы могли писать программы, не зная и не заботясь о том, какие устройства будут использоваться, и отлаживать их, используя локальный принтер, подключенный к компьютеру. После этого мы могли просто приказать операционной системе выполнить «печать» на магнитную ленту и получить сотни тысяч готовых форм.

Наши программы обрели форму. Эта форма отделила политику от деталей. Политика заключалась в форматировании записей с именами и адресами. А деталями стали устройства. Мы смогли отложить принятие решения о выборе устройства.

Физическая адресация

В начале 1970-х годов я работал над большой системой учета для местного профсоюза водителей грузовиков. У нас имелся диск емкостью 25 Мбайт, на котором хранились записи с информацией об агентах, работодателях и членах профсоюза. Разные записи имели разные размеры, поэтому первые несколько цилиндров на диске форматировались так, чтобы

размер одного сектора в точности соответствовал размеру записи с информацией об агенте. Следующие несколько цилиндров форматировались под записи с информацией о работодателях. И последние несколько цилиндров — под записи с информацией о членах профсоюза.

Мы писали программы, детально зная структуру диска. Мы знали, что на диске имеется 200 цилиндров и 10 головок, что каждый цилиндр имеет несколько десятков секторов на головку. Мы знали, какие цилиндры отводятся для хранения записей того или иного вида. И все это жестко «зашивалось» в код.

Мы также хранили на диске оглавление, позволявшее отыскать каждого агента, работодателя и члена профсоюза. Это оглавление хранилось в еще одной, специально отформатированной группе цилиндров. Оглавление для поиска агентов состояло из записей, включавших идентификационный номер агента, номер цилиндра, номер головки и номер сектора, где хранится запись с информацией об агенте. Для поиска работодателей и членов имелись аналогичные оглавления. Кроме того, информация о членах профсоюза хранилась на диске в форме двусвязного списка: каждая запись хранила номер цилиндра, головки и сектора следующей и предыдущей записи.

Представляете, во что выливался переход на использование нового дискового устройства, в котором на одну головку или на один цилиндр больше? Мы писали специальную программу, которая читала данные в старом формате со старого диска и записывала их на новый диск, преобразуя номера цилиндров/головок/секторов. Нам также приходилось изменять все жестко зашитые номера в коде, которые были *повсюду!* Все бизнес-правила скрупулезно учитывали схему цилиндр/головка/сектор.

Однажды в наши ряды влился более опытный программист. Увидев наше «творчество», он побледнел и уставился на нас как на инопланетян. Затем осторожно посоветовал нам изменить схему адресации и использовать относительные адреса.

Наш более умудренный опытом коллега предложил рассматривать диск как большой линейный массив секторов с последовательными номерами. Такая интерпретация позволяла написать крошечную подпрограмму преобразования, которая, зная физическую структуру диска, могла бы «на лету» транслировать относительные адреса в номера физических цилиндров/головок/секторов.

К счастью для нас, мы приняли его совет. Мы изменили высокоуровневую политику системы, добавив независимость от физической структуры диска. Это позволило нам отделить решение об организации записей на диске от прикладной задачи.

Заключение

Две истории, рассказанные в этой главе, наглядно демонстрируют один маленький принцип, который широко используют архитекторы. Хорошие архитекторы скрупулезно отделяют детали от политики и затем не менее скрупулезно отделяют политику от деталей, чтобы политика никак не зависела от деталей. Хорошие архитекторы проектируют политику так, чтобы решения о деталях можно было отложить и отодвинуть на как можно более поздний срок.

[34](#) SOA (Service-Oriented Architecture): сервис-ориентированная архитектура (https://ru.wikipedia.org/wiki/Сервис-ориентированная_архитектура). — Примеч. пер.

16. Независимость



Как отмечалось выше, хорошая архитектура должна обеспечивать:

- Разнообразие вариантов использования и эффективную работу системы.
- Простоту сопровождения системы.
- Простоту разработки системы.
- Простоту развертывания системы.

Варианты использования

Первый пункт в списке — разнообразие вариантов использования — означает, что архитектура системы должна поддерживать назначение системы. Если система — это онлайн-магазин, архитектура должна поддерживать разные варианты использования онлайн-магазина. Фактически в этом заключается главная задача архитектора и главная цель архитектуры. Архитектура должна поддерживать разнообразие вариантов использования.

Однако, как обсуждалось выше, архитектура не оказывает большого влияния на поведение системы. Существует очень мало вариантов поведения, которые архитектура может оставить открытыми. Но влияние — это еще не все. Самое важное, что хорошая архитектура может сделать для поддержки поведения, — раскрывать и прояснять это поведение, чтобы назначение системы было видимо на архитектурном уровне.

Приложение онлайн-магазина с хорошей архитектурой будет *выглядеть* как приложение онлайн-магазина. Варианты использования такой системы будут ясно видны в ее структуре. Разработчикам не придется искать черты поведения, потому что они будут представлены полноценными элементами, видимыми на верхнем уровне системы. Такими элементами могут быть классы, функции или модули, занимающие видные позиции в архитектуре и имеющие имена, ясно описывающие их назначение.

В главе 21 «Кричащая архитектура» мы подробнее рассмотрим этот пункт.

Эффективность работы

Более существенную и менее косметическую роль архитектура играет в эффективной работе системы. Если система должна обслуживать 100 000 клиентов в секунду, архитектура должна поддерживать такую пропускную способность и время отклика для каждого возможного варианта. Если система должна запрашивать большие массивы данных за миллисекунды, архитектура должна быть структурирована так, чтобы обеспечивать возможность подобных операций.

Для одних систем это означает создание массивов маленьких служб, выполняющихся параллельно на множестве разных серверов. Для других — создание множества легковесных потоков выполнения, действующих в общем адресном пространстве, внутри одного процесса, выполняющегося на одном процессоре. Для третьих достаточно запустить несколько процессов, действующих в изолированных адресных пространствах. А для некоторых вполне подойдут простые монолитные программы.

Как ни странно, это решение относится к числу тех, которые хороший архитектор стремится оставить открытыми. Систему с монолитной организацией и зависящую от этой монолитной структуры трудно будет реорганизовать для поддержки выполнения в нескольких процессах, нескольких потоках или нескольких микрослужбах, если это потребуется. Напротив, архитектуру, предусматривающую надлежащую изоляцию компонентов и не предполагающую конкретных средств взаимодействий между компонентами, будет намного проще провести через спектр нескольких потоков, процессов и служб по мере изменения потребностей.

Разработка

Архитектура играет важную роль в поддержке разработки. В данном случае начинает действовать закон Конвея:

Любая организация, разрабатывающая систему, невольно будет формировать дизайн, структура которого повторяет структуру взаимодействий внутри этой организации.

В организации с множеством команд, решающей множество задач, должна получиться архитектура, которая обеспечивает независимую работу этих команд, чтобы команды не мешали друг другу во время разработки. Это достигается делением системы на изолированные компоненты, которые можно разрабатывать независимо. Затем эти компоненты распределяются между командами, которые могут работать независимо друг от друга.

Развертывание

Огромную роль архитектура играет также в простоте развертывания системы. Главная цель — «немедленное развертывание». Хорошая архитектура не полагается на десятки маленьких сценариев и файлов с определениями настроек. Она не требует вручную создавать каталоги или файлы и располагать их в особом порядке. Хорошая архитектура помогает системе достичь уровня немедленной готовности к развертыванию сразу после сборки.

И снова это достигается за счет разделения системы на компоненты и их изоляции, включая специальные компоненты, которые связывают систему воедино и гарантируют правильный запуск, интеграцию и контроль за работой каждого компонента.

Сохранение разнообразия вариантов

Хорошая архитектура стремится достичь баланса всех этих потребностей компонентной организации с целью удовлетворить их. Звучит просто, правда? Согласен, мне легко писать.

В действительности добиться такого баланса очень трудно. Проблема в том, что зачастую мы не знаем всех возможных вариантов использования, эксплуатационных ограничений, структуры команды разработчиков или требований к развертыванию. Хуже того, даже если бы мы все это знали, нет никаких гарантий, что ничего из этого не изменится в течение жизненного цикла системы. Проще говоря, мы преследуем нечеткие и переменчивые цели. Добро пожаловать в реальный мир.

Но не все потеряно: некоторые принципы строительства архитектуры относительно недороги в реализации и могут помочь сбалансировать эти нужды, даже когда нет четкой картины целей, к которым вы стремитесь. Эти принципы помогают нам делить системы на хорошо изолированные компоненты, что позволяет максимально долго оставлять как можно больше вариантов открытыми.

Хорошая архитектура упрощает возможность изменения системы во всех ее вариантах, оставляя возможности открытыми.

Разделение уровней

Поговорим о вариантах использования. Архитектор стремится получить структуру системы, поддерживающую необходимые варианты использования, но не знает их все. Зато он знает основное назначение системы. Это может быть онлайн-магазин, система учета материалов или система обработки

заказов. Поэтому, опираясь на контекст назначения системы, архитектор может использовать принципы единственной ответственности и согласованного изменения для отделения друг от друга всего, что изменяется по разным причинам, и объединения всего, что изменяется по одной причине.

Что изменяется по разным причинам? Кое-что из этого сразу бросается в глаза. Пользовательский интерфейс изменяется по причинам, не имеющим ничего общего с бизнес-правилами. Все варианты использования включают оба этих элемента. Очевидно, что поэтому хороший архитектор отделит часть, отвечающую за пользовательский интерфейс, от части, реализующей бизнес-правила, так, что их можно будет изменять независимо друг от друга, сохраняя при этом ясную видимость вариантов использования.

Сами бизнес-правила могут делиться на более или менее тесно связанные с приложением. Например, проверка полей ввода — это бизнес-правило, тесно связанное с приложением. Вычисление процентов по вкладу и подсчет запасов, напротив, — это бизнес-правила, более тесно связанные с предметной областью. Эти два разных вида бизнес-правил будут изменяться с разной скоростью и по разным причинам, поэтому их следует разделить так, чтобы их можно было изменять независимо.

База данных, язык запросов и даже схема — все это технические детали, которые не имеют ничего общего с бизнес-правилами и пользовательским интерфейсом. Они будут изменяться со скоростью и по причинам, не зависящим от других аспектов системы. Следовательно, в архитектуре они должны отделяться от остальной системы, чтобы их можно было изменять независимо.

То есть систему можно разделить на горизонтальные уровни: пользовательский интерфейс, бизнес-правила,

характерные для приложения, бизнес-правила, не зависящие от приложения, и база данных — кроме всего прочего.

Разделение вариантов использования

Что еще меняется по разным причинам? Сами варианты использования! Добавление заказов в систему сопровождения заказов почти наверняка будет изменяться с иной скоростью и по иным причинам, чем удаление заказов из системы. Варианты использования дают очень естественный способ деления системы.

В то же время варианты использования образуют узкие вертикальные срезы, пересекающие горизонтальные слои системы. Каждый вариант использования включает какую-то долю пользовательского интерфейса, часть бизнес-правил, характерных для приложения, какие-то бизнес-правила, не зависящие от приложения, и некоторые функции базы данных. То есть вместе с делением системы на горизонтальные слои мы одновременно делим ее на тонкие вертикальные варианты использования, пересекающие эти слои.

Чтобы получить желаемое разделение, мы должны отделить пользовательский интерфейс добавления заказа от пользовательского интерфейса удаления заказа. То же самое нужно проделать с бизнес-правилами и базой данных. Вертикальные срезы случаев использования системы мы сохраняем отдельными.

Здесь можно увидеть определенный шаблон. Разделив элементы системы, изменяющиеся по разным причинам, вы сможете добавлять новые варианты использования, не влияя на имеющиеся. Если при этом вы сгруппируете поддержку этих случаев использования в пользовательском интерфейсе и в базе данных так, что для всех них в пользовательском

интерфейсе и базе данных будут предусмотрены свои аспекты, тогда добавление новых случаев использования почти гарантированно не повлияет на имеющиеся.

Режим разделения

Теперь задумаемся, какое значение имеет разделение для эффективной работы. Если разные варианты использования разделены, тогда те, что должны работать с максимальной пропускной способностью, уже наверняка отделены от работающих с низкой пропускной способностью. Если пользовательский интерфейс и база данных отделены от бизнес-правил, значит, все они могут выполняться на разных серверах. Части, которые должны выполняться с максимальной пропускной способностью, можно запустить на нескольких серверах.

Проще говоря, разделение вариантов использования также помогает повысить эффективность работы. Однако для этого разделение должно иметь соответствующий режим. Чтобы разделенные компоненты могли выполняться на собственных серверах, они не должны зависеть от наличия общего адресного пространства в памяти. Они должны быть независимыми службами, взаимодействующими по сети.

Многие архитекторы называют такие компоненты «службами» или «микрослужбами», в зависимости от некоторого неопределенного понятия величины. Архитектуру, основанную на службах (или сервисах), часто называют сервис-ориентированной архитектурой.

Если в ответ на увиденные термины в вашей голове зазвенел звоночек, не волнуйтесь. Я не собираюсь убеждать вас, что SOA³⁵ — лучшая из возможных архитектур или что за микрослужбами будущее. Я лишь хотел отметить, что иногда

мы должны разделять свои компоненты вплоть до создания отдельных служб.

Помните, что хорошая архитектура оставляет возможности открытыми. *Режим разделения — одна из таких возможностей*.

Прежде чем продолжить обсуждение этой темы, рассмотрим два других пункта.

Возможность независимой разработки

В третьем пункте, в начале главы, говорилось о простоте разработки. Очевидно, что полное разделение компонентов ослабляет взаимовлияние команд разработчиков. Если бизнес-правила никак не зависят от пользовательского интерфейса, тогда команда, занятая его разработкой, едва ли сможет как-то помешать команде, занятой реализацией бизнес-правил. Если варианты использования отделены друг от друга, тогда команда, сосредоточенная на реализации варианта `addOrder`, не сможет помешать команде, сосредоточенной на варианте `deleteOrder`.

Архитектура системы, разделенная на уровни и варианты, будет поддерживать любую организацию команд, по какому бы признаку они ни были организованы — функциональному, по компонентам, по уровням или как-то иначе.

Возможность независимого развертывания

Разделение на уровни и варианты использования помогает также добиться максимальной гибкости при развертывании. Фактически при эффективном разделении должна открываться возможность «горячей» замены уровней и вариантов использования в действующей системе. Добавление нового варианта использования может заключаться в простом

копировании нескольких jar-файлов или служб, без влияния на что-то другое.

Дублирование

Архитекторы часто попадают в ловушку, зависящую от их страха перед дублированием.

Вообще в программном обеспечении считается, что дублирования следует избегать. Мы стараемся предотвратить дублирование программного кода. Когда обнаруживается действительно повторяющийся код, мы с благоговением смотрим, как профессионалы устраниют его.

Но есть разные виды дублирования. Есть истинное дублирование, когда любое изменение в одной копии требует того же изменения во всех остальных копиях. А есть ложное или случайное дублирование. Если два фрагмента кода, кажущиеся одинаковыми, развиваются разными путями — если они изменяются с разной скоростью и по разным причинам — *этот случай не является истинным дублированием*. Вернитесь к ним через несколько лет, и вы увидите, что они совершенно не похожи друг на друга.

Теперь представьте два варианта использования, одинаково отображающиеся на экране. Многие архитекторы будут испытывать непреодолимое желание использовать общую реализацию отображения. Но правы ли они? Действительно ли это истинное дублирование? Или сходство получилось случайным?

Скорее всего, это случайность. С течением времени эти два экрана будут приобретать все больше и больше различий и, наконец, будут выглядеть совершенно непохоже. По этой причине желательно избежать их объединения. Иначе потом разделить их будет очень сложно.

В процессе создания вертикального деления на варианты использования могут возникать проблемы и желание объединить какие-то варианты со схожими отображениями на экране, или алгоритмами, или запросами к базе данных и/или схемами. Будьте осторожны. Сопротивляйтесь рефлекторному желанию устраниить дублирование. Убедитесь прежде, что дублирование действительно истинное.

Точно так же, выполняя горизонтальное деление на уровни, можно заметить, что структура данных в какой-то записи в базе данных очень похожа на структуру данных в экранной форме. В результате может возникнуть желание просто передать запись из базы данных в пользовательский интерфейс, не создавая промежуточную модель представления, которая, казалось бы, просто копирует элементы. Будьте осторожны: эта схожесть почти всегда оказывается случайной. Создание отдельной модели представления не требует много усилий и поможет вам правильно отделить уровни.

Режимы разделения (еще раз)

Вернемся к режимам. Существует много разных способов деления на уровни и варианты использования. Деление можно выполнить на уровне исходного кода, на уровне двоичного кода (развертывания) и на уровне единиц выполнения (служб).

- **Уровень исходного кода.** Мы можем так настроить зависимости между модулями с исходным кодом, чтобы изменения в одном модуле не вынуждали производить изменения в других (например, Ruby Gems).

При использовании этого режима разделения все компоненты выполняются в общем адресном пространстве и взаимодействуют, просто вызывая функции друг друга. То есть имеется единственный выполняемый файл, загружаемый в память компьютера. Люди часто называют это монолитной структурой.

- **Уровень развертывания.** Мы можем так настроить зависимости между единицами развертывания, jar-файлами или динамически загружаемыми библиотеками, чтобы изменения в исходном коде в одном модуле не вынуждали производить повторную сборку и развертывание других.

Многие компоненты могут находиться в общем адресном пространстве и взаимодействовать, вызывая функции друг друга. Другие компоненты могут выполняться в других процессах на той же машине и взаимодействовать посредством механизмов межпроцессных взаимодействий, сокетов или разделяемой памяти. Важно отметить, что в этом случае разделенные компоненты находятся в независимых единицах развертывания, таких как jar-, gem-файлы или динамически загружаемые библиотеки.

- **Уровень служб.** Мы можем ограничить зависимости до уровня структур данных и взаимодействовать, обмениваясь исключительно сетевыми пакетами, чтобы каждая единица выполнения была по-настоящему независимой от изменений в исходном и двоичном коде в других (как, например, службы и микрослужбы).

Какой режим лучше?

Ответ заключается в следующем: на ранних этапах разработки проекта трудно понять, какой режим лучше. В действительности по мере взросления проекта оптимальный режим может изменяться.

Например, нетрудно представить, что система, вполне комфортно чувствовавшая себя на одном сервере, может разрастись до масштабов, когда удобнее перенести некоторые ее компоненты на отдельные серверы. Пока система действует на одном сервере, разделения на уровне исходного кода может быть вполне достаточно. Но позднее может понадобиться разделить ее на единицы развертывания или даже на службы.

Одно из решений (набирающее популярность в наши дни) заключается в изначальном разделении на службы. Однако такой подход дорог и способствует разделению на слишком крупные блоки. Какими бы «микро» ни получились микрослужбы, маловероятно, что разделение будет выполнено на достаточно мелкие блоки.

Еще одна проблема разделения на службы — дороговизна, и в отношении разработки, и в отношении системных ресурсов. Для преодоления границ между службами, когда в действительности в них нет необходимости, требуются дополнительные усилия, память и машинные такты. И да, я помню, что последние два дешевы, но первое — нет.

Я предпочитаю доводить деление до того состояния, когда при необходимости легко можно сформировать отдельные службы, но оставлять компоненты в общем адресном пространстве до тех пор, пока это возможно. Это оставляет открытой возможность организации служб.

При таком подходе компоненты первоначально разделяются на уровне исходного кода. Этого может оказаться достаточно для продолжительного развития проекта. Если возникают проблемы с развертыванием или разработкой,

разделения до уровня развертывания может быть достаточно — по крайней мере на какое-то время.

По мере накопления проблем с разработкой, развертыванием и эффективной работой я тщательно отбираю единицы развертывания, которые можно преобразовать в службы, и постепеннодвигаю систему в этом направлении.

Со временем требования к эффективности работы могут снижаться. И для системы, прежде требовавшей разделения до уровня служб, теперь может оказаться достаточно разделения до уровня развертывания или даже исходного кода.

Хорошая архитектура позволит создать монолитную систему, развертываемую как один файл, а затем превратить ее в набор независимых единиц развертывания и далее в независимые службы и/или микрослужбы. Позднее, когда что-то изменится, она должна позволить обратить все вспять и вновь вернуться к монолитной структуре.

Хорошая архитектура защищает большую часть исходного кода от таких изменений. Она оставляет режим разделения открытым, поэтому для крупномасштабных вариантов развертывания может использоваться один режим, а для мелкомасштабных — другой.

Заключение

Да, все это непросто. Впрочем, я и не говорю, что изменение режимов разделения должно сводиться к тривиальным изменениям в конфигурации (хотя иногда такое возможно). Я говорю, что режим разделения системы на компоненты относится к числу вариантов, которые, скорее всего, будут изменяться со временем, и хороший архитектор должен предвидеть такие изменения и стремиться к упрощению.

[35](#) SOA (Service-Oriented Architecture): сервис-ориентированная архитектура. —
Примеч. пер.

17. Границы: проведение разделяющих линий



Разработка архитектуры программного обеспечения — это искусство проведения разделяющих линий, которые я называю *границами*. Границы отделяют программные элементы друг от друга и избавляют их от необходимости знать, что находится по ту сторону. Некоторые из этих линий можно провести на самых ранних этапах развития проекта — даже до появления первого программного кода. Другие проводятся намного позже. Границы, проводимые на ранних этапах, призваны отложить принятие решений на как можно долгий срок и предотвратить загрязнение основной бизнес-логики этими решениями.

Напомню, что целью архитектора является минимизация трудозатрат на создание и сопровождение системы. Что может помешать достижению этой цели? *Зависимость* — и особенно зависимость от преждевременных решений.

Какие решения можно назвать преждевременными? Решения, не имеющие ничего общего с бизнес-требованиями — вариантами использования — системы. К ним можно отнести решения о выборе фреймворка, базы данных, веб-сервера, вспомогательных библиотек, механизма внедрения зависимостей и т.п. В хорошей архитектуре подобные решения носят вспомогательный характер и откладываются на потом. Хорошая архитектура не зависит от таких решений. Хорошая архитектура позволяет принимать эти решения в самый последний момент без существенного влияния на саму архитектуру.

Пара печальных историй

Расскажу одну печальную историю о компании Р, которая послужит предупреждением всем, кто торопится принимать решения. В 1980-х годах основатели компании Р написали простое монолитное приложение для настольного компьютера. Оно пользовалось большим успехом, и в 1990-е годы этот продукт превратили в популярное и успешное приложение с графическим интерфейсом.

Но потом, в конце 1990-х годов, началось бурное развитие Всемирной паутины. Многие внезапно решили, что у них должны быть свои веб-решения, и Р не стала исключением. Клиенты компании Р настойчиво требовали написать версию продукта для Веб. Чтобы удовлетворить это требование, компания наняла двадцать с лишним крутых программистов

на Java и приступила к проектированию веб-версии своего продукта.

Парни грезили о фермах серверов, поэтому решили выбрать богатую трехуровневую «архитектуру»³⁶, которую они могли бы развернуть на такой ферме. Они предполагали, что у них будут отдельные серверы для обслуживания интерфейса с пользователем, серверы для промежуточного программного обеспечения и серверы для баз данных. Но красиво было на бумаге, да забыли про овраги.

Программисты слишком рано решили, что все предметные объекты должны иметь по три экземпляра: один для уровня графического интерфейса, один для промежуточного уровня и один для уровня базы данных. Так как все эти экземпляры находились на разных серверах, была создана разветвленная система взаимодействий между процессами и уровнями. Вызовы методов между уровнями преобразовывались в объекты, которые подвергались сериализации и передавались по сети.

Теперь представьте, во что выливается простое изменение, такое как добавление нового поля в существующую запись. Поле нужно добавить в классы на всех трех уровнях, а также в некоторые сообщения, циркулирующие между уровнями. Так как передача данных происходит в обоих направлениях, необходимо спроектировать четыре протокола обмена сообщениями. Каждый протокол имеет отправляющую и принимающую стороны, итого необходимо реализовать восемь обработчиков протоколов. Нужно собрать три выполняемых файла, в каждом из которых находятся три измененных бизнес-объекта, четыре новых сообщения и восемь новых обработчиков.

А теперь задумайтесь, что все эти выполняемые файлы должны делать, чтобы выполнить простейшее действие. Они

создают экземпляры объектов, выполняют их сериализацию, маршалинг и демаршалинг, конструируют и анализируют сообщения, выполняют операции с сокетами, обрабатывают тайм-ауты и производят повторные попытки и многое еще только ради выполнения одной простой операции.

Конечно, во время разработки у программистов не было фермы серверов. В действительности они просто запускали все три выполняемых файла в трех разных процессах на одной машине. Так они развивали проект в течение нескольких лет, но были уверены, что их архитектура правильная. И поэтому даже когда система выполнялась на одной машине, она по-прежнему продолжала создавать объекты, выполнять их сериализацию, маршалинг и демаршалинг, конструировать и анализировать сообщения, оперировать сокетами и многое чего еще, ненужного на одной машине.

По иронии ни одна из систем, проданных компанией Р, не требовала фермы серверов. Все системы, когда-либо развертывавшиеся компанией, размещались на единственном сервере. И на этом единственном сервере три выполняемых файла продолжали создавать объекты, выполнять их сериализацию, маршалинг и демаршалинг, конструировать и анализировать сообщения, оперировать сокетами и многое чего еще в ожидании фермы серверов, которая никогда не существовала и никогда не будет существовать.

Трагедия в том, что архитекторы, приняв преждевременное решение, чрезмерно увеличили трудозатраты на разработку.

История с компанией Р не единственная. Я наблюдал нечто подобное много раз и во многих местах. Фактически компания Р является лишь одним из примеров.

Но есть еще более печальные примеры, чем пример компании Р.

Представьте себе компанию W, местное предприятие, управляющее парками служебных автомобилей. Недавно они наняли «Архитектора», чтобы взять под контроль разрозненные усилия по разработке программного обеспечения. И, хочу вам сообщить, «Контроль» — это второе имя того парня. Он быстро решил, что требуется создать полномасштабную, корпоративную, сервис-ориентированную **«АРХИТЕКТУРУ»**. Он создал гигантскую предметную модель всех возможных «бизнес-объектов», спроектировал набор служб для управления этими объектами и направил разработчиков по пути в Ад. Чтобы было понятнее, представьте, что вам понадобилось добавить имя, адрес и номер телефона контактного лица в запись о продаже. Для этого нужно обратиться в `ServiceRegistry`, запросить идентификатор службы `ContactService`. Затем послать сообщение `CreateContact` в `ContactService`. Конечно, это сообщение имеет десятки полей, каждое из которых должно содержать достоверные данные — данные, к которым у программиста не было доступа, потому что программист имел только имя, адрес и номер телефона. После заполнения полей ложными данными программист должен был вставить идентификатор вновь созданного контакта в запись и послать сообщение `UpdateContact` службе `SaleRecordService`.

Чтобы протестировать все это, нужно по порядку запустить все необходимые службы, поднять шину сообщений и сервер Bpel и... И затем все эти сообщения перемещались от службы к службе и ждали обработки то в одной очереди, то в другой.

Чтобы добавить что-то новое, представьте только зависимости между всеми этими службами и объем кода WSDL, который нужно изменить, а потом повторно развернуть модули, в которые были внесены изменения...

В сравнении с этим ад начинает казаться не таким плохим местом.

В программной системе, организованной в виде набора служб, нет ничего принципиально неправильного. Ошибка компании W заключалась в преждевременном решении внедрить комплекс инструментов для поддержки SOA — то есть массивного набора служб для работы с предметными объектами. За эту ошибку пришлось заплатить человеко-часами — большим количеством человеко-часов, — сброшенными с вершины SOA.

Я мог бы описывать архитектурные провалы один за другим. Но давайте лучше поговорим об успешных примерах.

FitNesse

Мой сын Майка и я начали работу над проектом FitNesse в 2001 году. Мы намеревались создать простую вики-страницу, обертывающую инструмент FIT Уорда Каннингема для разработки приемочных тестов.

Это было еще до того, как в *Maven* «решили» проблему jar-файла. Я был абсолютно уверен: все, что мы производим, не должно вынуждать людей загружать больше одного jar-файла. Я назвал это правило «Загрузи и вперед». Это правило управляло многими нашими решениями.

Одним из первых решений было создание собственного веб-сервера, отвечающего потребностям FitNesse. Это решение может показаться абсурдным. Даже в 2001 году существовало немало веб-серверов с открытым исходным кодом, которые мы могли бы использовать. Тем не менее решение написать свой сервер оправдало себя, потому что реализовать простой веб-сервер совсем несложно и это позволило нам отложить решение о выборе веб-фреймворка на более поздний срок³⁷.

Еще одно решение, принятое на ранней стадии, — не думать о базе данных. У нас была задумка использовать MySQL, но мы намеренно отложили это решение, использовав дизайн, сделавший это решение несущественным. Суть его заключалась в том, чтобы вставить интерфейс между всеми обращениями к данным и самим хранилищем.

Мы поместили методы доступа к данным в интерфейс с именем `WikiPage`. Эти методы обеспечивали все необходимое для поиска, извлечения и сохранения страниц. Конечно, мы не реализовали эти методы с самого начала, а просто добавили «заглушки», пока работали над функциями, не связанными с извлечением и сохранением данных.

В действительности в течение трех месяцев мы работали над переводом текста вики-страницы в HTML. Это не потребовало использования какого-либо хранилища данных, поэтому мы создали класс с именем `MockWikiPage`, содержащий простые заглушки методов доступа к данным.

В какой-то момент этих заглушек оказалось недостаточно для новых функций, которые мы должны были написать. Нам понадобился настоящий доступ к данным, без заглушек. Поэтому мы создали новый класс `InMemoryPage`, производный от `WikiPage`. Этот класс реализовал методы доступа к данным в хеш-таблице с вики-страницами, хранящейся в ОЗУ.

Это позволило нам целый год писать функцию за функцией. В результате мы получили первую версию программы FitNesse, действующую именно так. Мы могли создавать страницы, ссылаться на другие страницы, применять самое причудливое форматирование и даже выполнять тесты под управлением FIT. Мы не могли только сохранять результаты нашего труда.

Когда пришло время реализовать долговременное хранение, мы снова подумали о MySQL, но решили, что в краткосрочной перспективе это необязательно, потому что

хеш-таблицы было очень легко записывать в простые файлы. Как результат, мы реализовали класс `FileSystemWikiPage`, который работал с простыми файлами, и продолжили работу над созданием новых возможностей.

Три месяца спустя мы пришли к заключению, что решение на основе простых файлов достаточно хорошее, и решили вообще отказаться от идеи использовать MySQL. Мы отложили это решение до неопределенного будущего и никогда не оглядывались назад.

На этом история закончилась бы, если бы один из наших клиентов, решивший, что ему очень нужно сохранить свою вики-страницу в MySQL. Мы показали ему архитектуру `WikiPages`, позволившую нам отложить решение. *Через день* он вернулся с законченной системой, работавшей в MySQL. Он просто написал производный класс `MySqlWikiPage` и получил необходимое.

Мы включили этот вариант в FitNesse, но никто больше не использовал его, по крайней мере, поэтому мы отбросили его. Даже клиент, написавший производный класс, в конечном счете отказался от него.

Начиная работу над FitNesse, мы *провели границу* между бизнес-правилами и базами данных. Эта граница позволила реализовать бизнес-правила так, что они вообще никак не зависели от выбора базы данных, им требовалось только методы доступа к данным. Это решение позволило нам больше года откладывать выбор и реализацию базы данных. Оно позволило опробовать вариант с файловой системой и изменить направление, когда мы увидели лучшее решение. Кроме того, оно не препятствовало и даже не мешало движению в первоначальном направлении (к MySQL), когда кто-то пожелал этого.

Факт отсутствия действующей базы данных в течение 18 месяцев разработки означал, что 18 месяцев мы не испытывали проблем со схемами, запросами, серверами баз данных, паролями, тайм-аутами и прочими неприятностями, которые непременно начинают проявляться, как только вы включаете в работу базу данных. Это также означало, что все наши тесты выполнялись очень быстро, потому что не было базы данных, тормозившей их.

Проще говоря, проведение границ помогло нам задержать и отложить принятие решений, и это сэкономило нам немало времени и нервов. Именно такой должна быть хорошая архитектура.

Какие границы проводить и когда?

Отделять линиями нужно все, что не имеет значения. Графический интерфейс не имеет значения для бизнес-правил, поэтому между ними нужно провести границу. База данных не имеет значения для графического интерфейса, поэтому между ними нужно провести границу. База данных не имеет значения для бизнес-правил, поэтому между ними нужно провести границу.

Возможно, кто-то из вас пожелал бы возразить против этих заявлений, особенно против заявления о независимости бизнес-правил от базы данных. Многие из нас привыкли считать, что база данных неразрывно связана с бизнес-правилами. Кое-кто может быть даже уверен, что база данных является воплощением бизнес-правил.

Но, как будет показано в другой главе, эта идея ошибочна. База данных — это инструмент, который бизнес-правила могут использовать опосредованно. Бизнес-правила не зависят от конкретной схемы, языка запросов и любых других деталей

организации базы данных. Бизнес-правилам требуется только набор функций для извлечения и сохранения данных. Это позволяет нам скрыть базу данных за интерфейсом.

Это ясно видно на рис. 17.1. Класс `BusinessRules` использует интерфейс `DatabaseInterface` для загрузки и сохранения данных. Класс `DatabaseAccess` реализует интерфейс и выполняет операции в фактической базе данных `Database`.

Классы и интерфейсы на этой диаграмме носят символический характер. В настоящем приложении может иметься несколько классов бизнес-правил, несколько интерфейсов доступа данным и несколько классов, реализующих этот доступ. Но все они будут следовать примерно одному и тому же шаблону.

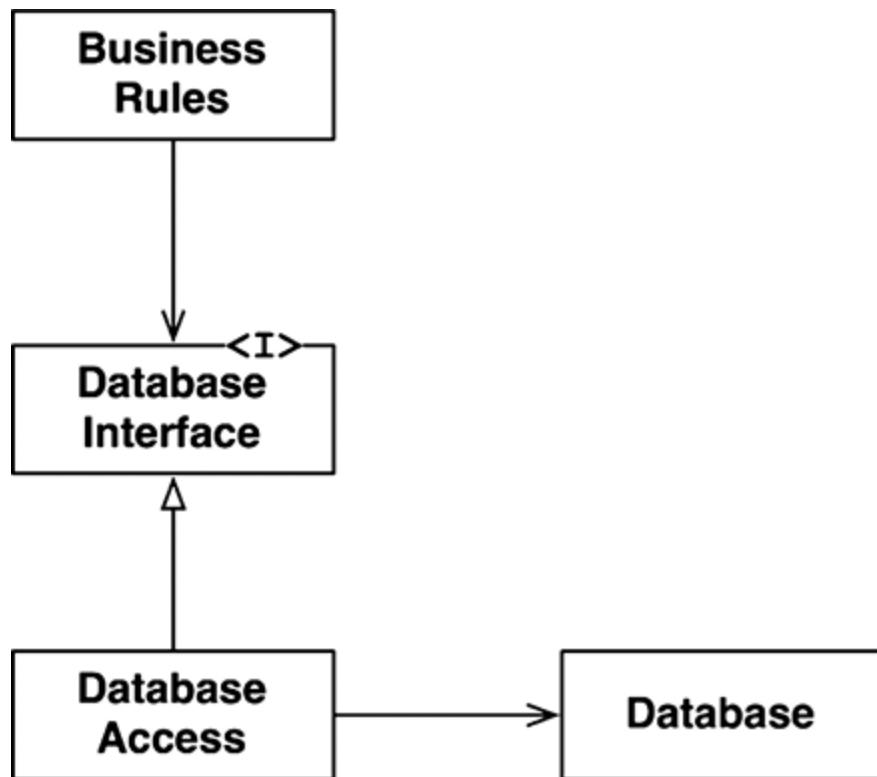


Рис. 17.1. База данных за интерфейсом

Где здесь граница? Граница пересекает отношение наследования чуть ниже DatabaseInterface (рис. 17.2).

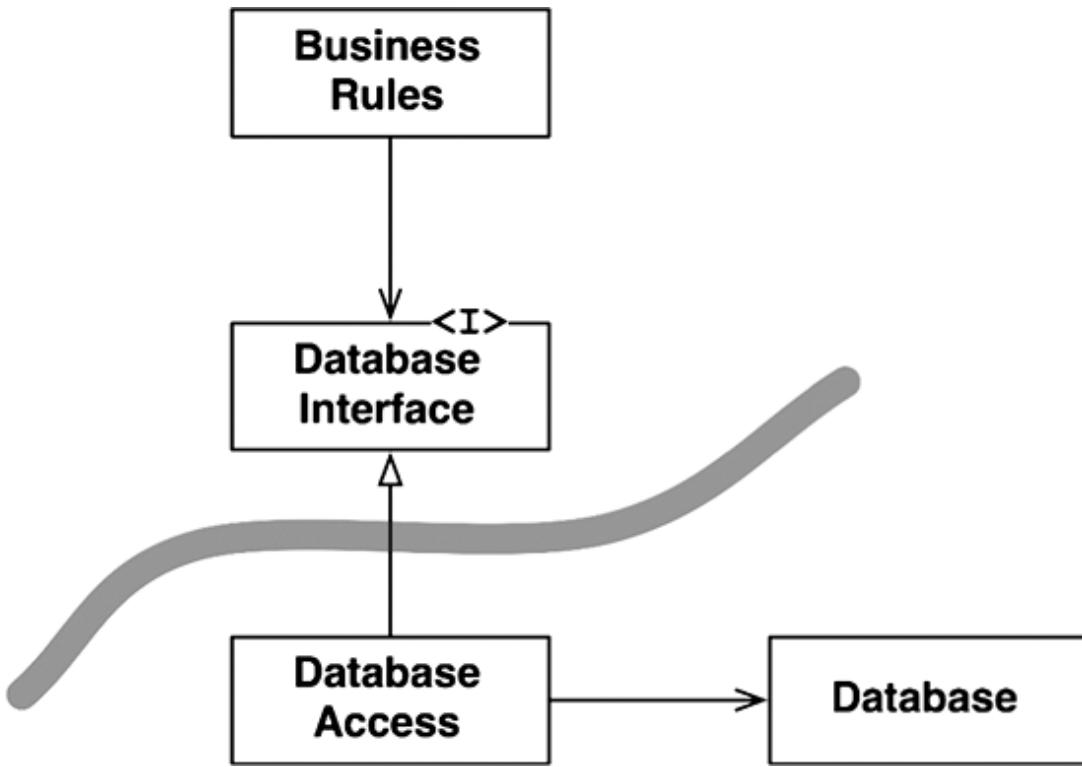


Рис. 17.2. Линия границы

Обратите внимание на стрелки, исходящие из класса DatabaseAccess. Они обе выходят из класса DatabaseAccess, а это значит, что никакой другой класс не знает о существовании DatabaseAccess.

Теперь отступим на шаг назад и рассмотрим компонент с несколькими бизнес-правилами и компонент с базой данных и всеми необходимыми классами доступа (рис. 17.3).

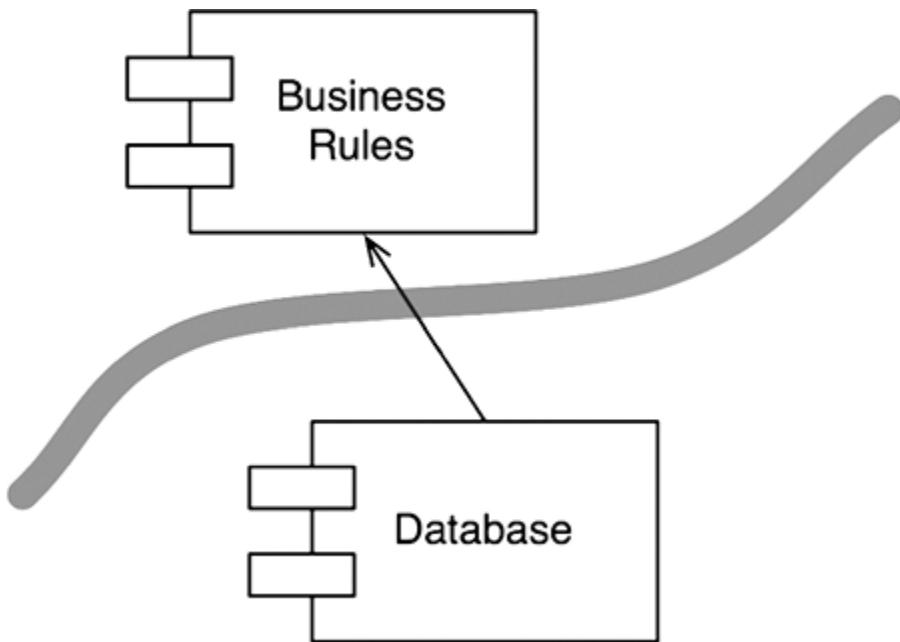


Рис. 17.3. Компоненты с бизнес-правилами и базой данных

Обратите внимание на направление стрелки. Компонент `Database` знает о существовании компонента `BusinessRules`. Компонент `BusinessRules` не знает о существовании компонента `Database`. Это говорит о том, что интерфейсы `DatabaseInterface` находятся в компоненте `BusinessRules`, а классы `DatabaseAccess` – в компоненте `Database`.

Направление этой стрелки важно. Оно показывает, что компонент `Database` не имеет значения для `BusinessRules`, но `Database` не может существовать без `BusinessRules`.

Если вам это кажется странным, просто вспомните, что компонент `Database` содержит код, транслирующий вызовы, выполняемые компонентом `BusinessRules`, на язык запросов базы данных. Именно этот транслирующий код знает о существовании `BusinessRules`.

Проведя границу между двумя компонентами и направив стрелку в сторону `BusinessRules`, мы видим, что компонент

`BusinessRules` мог бы использовать базу данных любого типа. Компонент `Database` можно заменить самыми разными реализациями — для `BusinessRules` это совершенно неважно.

Хранение данных можно организовать в базе данных Oracle, MySQL, Couch, Datomic или даже в простых файлах. Для бизнес-правил это совершенно неважно. А это означает, что выбор базы данных можно отложить и сосредоточиться на реализации и тестировании бизнес-правил.

О вводе и выводе

Разработчики и клиенты часто неправильно понимают, что такое система. Они видят графический интерфейс и думают, что он и есть система. Они определяют систему в терминах графического интерфейса и считают, что должны сразу начать работу с графическим интерфейсом. Они не понимают важнейшего принципа: *ввод/вывод не важен*.

В первый момент это может быть трудно понять. Мы часто рассуждаем о поведении системы в терминах ввода/вывода. Возьмем, например, видеоигру. В вашем представлении наверняка доминирует интерфейс: экран, мышь, управляющие клавиши и звуки. Вы забываете, что за этим интерфейсом находится модель — сложный комплекс структур данных и функций, — управляющая им. Что еще важнее, эта модель не нуждается в интерфейсе. Она благополучно будет решать свои задачи, моделируя все игровые события даже без отображения игры на экране. Интерфейс не важен для модели — для бизнес-правил.

И снова мы видим, что компоненты GUI и `BusinessRules` разделены границей (рис. 17.4). И снова мы видим, что менее важный компонент зависит от более важного компонента.

Стрелки показывают, какой компонент знает о существовании другого и, соответственно, какой компонент зависит от другого. Компонент GUI зависит от BusinessRules.

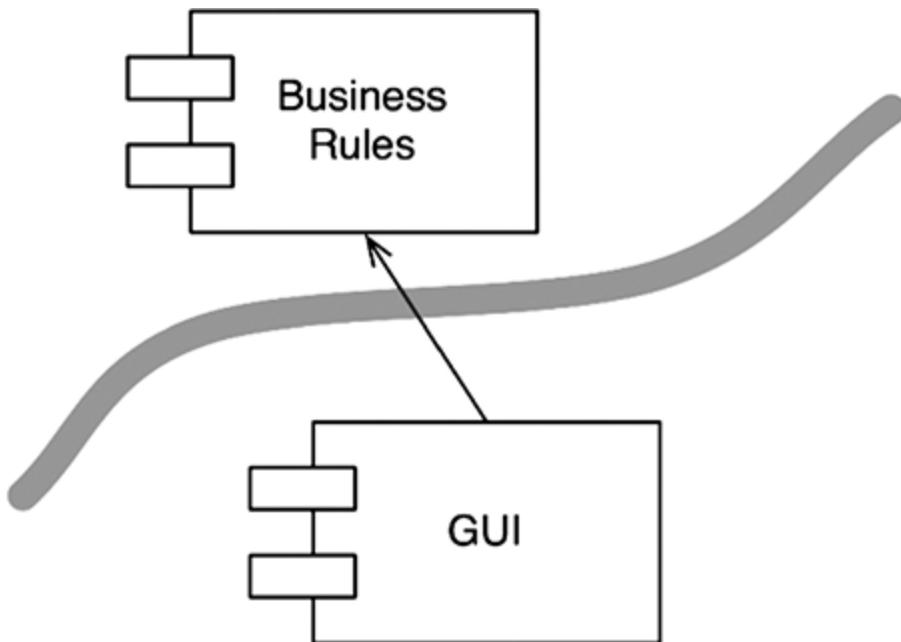


Рис. 17.4. Граница между компонентами GUI и BusinessRules

Проведя границу и нарисовав стрелку, мы теперь видим, что GUI можно заменить интерфейсом какого-то другого вида — для BusinessRules это не важно.

Архитектура с плагинами

Вместе эти два решения о базе данных и графическом интерфейсе образуют шаблон для добавления других компонентов. Это тот же шаблон, что используется в системах, допускающих подключение сторонних сменных модулей — плагинов.

Фактически история технологий разработки программного обеспечения — это история создания плагинов для получения масштабируемой и управляемой архитектуры. Основные

бизнес-правила хранятся отдельно и не зависят от компонентов, которые являются необязательными или могут быть реализованы в множестве разных форм (рис. 17.5).

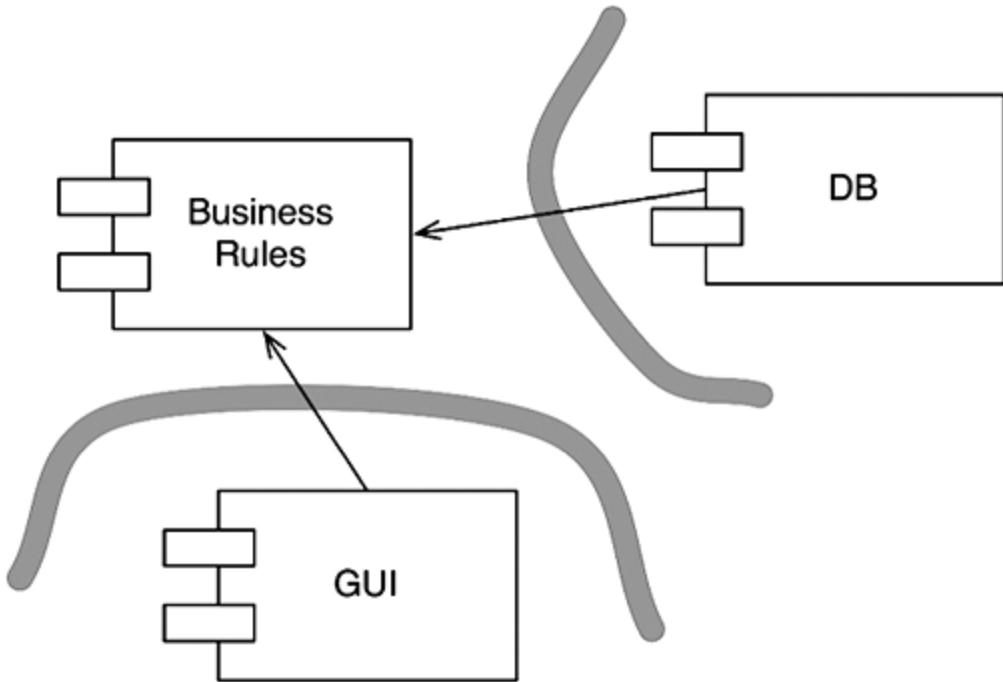


Рис. 17.5. Подключение модулей к бизнес-правилам

Так как в таком дизайне пользовательский интерфейс считается плагином, мы можем позволить подключать множество разных пользовательских интерфейсов. Это могут быть веб-интерфейсы, интерфейсы клиент/сервер, интерфейсы служб, консольные интерфейсы или основанные на других способах взаимодействия с пользователем.

То же верно в отношении базы данных. Решив считать ее плагином, мы можем заменить ее любой базой данных SQL или NOSQL, простыми файлами или любыми другими технологиями хранения данных, которые мы сочтем необходимыми в будущем.

Такие замены осуществляются не всегда просто. Если первоначально система опиралась на веб-интерфейс, создание

плагина для интерфейса клиент/сервер может оказаться сложной задачей. Вполне вероятно, что придется переделать какие-то взаимодействия между бизнес-правилами и новым пользовательским интерфейсом. И все же, допустив существование архитектуры со сменными модулями (плагинами), мы сделали подобную замену практически возможной.

Аргумент в пользу плагинов

Рассмотрим отношения между ReSharper и Visual Studio. Эти компоненты производятся совершенно разными коллективами разработчиков в совершенно разных компаниях. И действительно, компания *JetBrains*, создатель ReSharper, находится в России. Компания *Microsoft*, конечно, находится в Редмонде, штат Вашингтон, США. Трудно представить себе более разные команды разработчиков.

Какая команда может помешать другой? Какая команда защищена от влияния другой? Структура зависимости четко отвечает на оба вопроса (рис. 17.6). Исходный код ReSharper зависит от исходного кода Visual Studio. То есть команда ReSharper никак не сможет помешать команде Visual Studio. Но команда Visual Studio может полностью заблокировать команду ReSharper, если пожелает.

Это крайне асимметричное отношение, и именно его желательно воспроизвести в своих системах. Некоторые модули должны иметь абсолютную

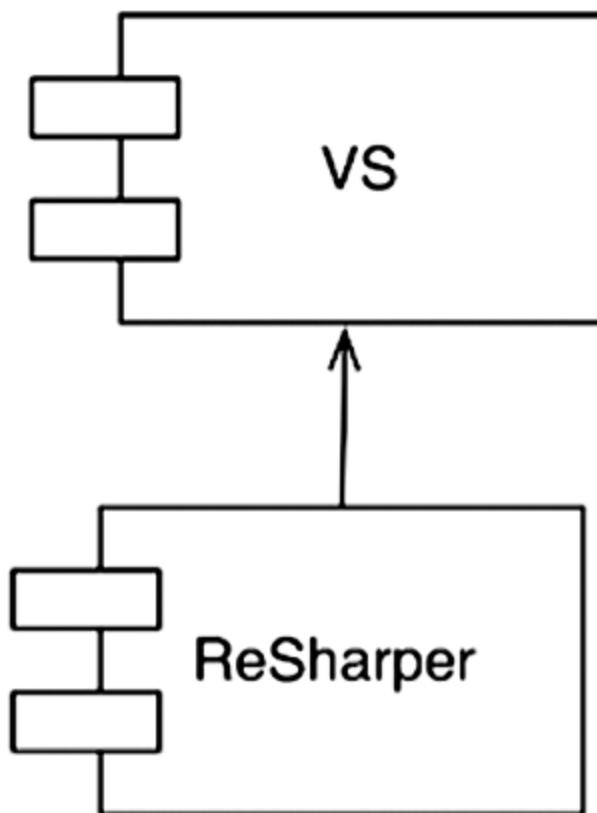


Рис. 17.6. ReSharper зависит от Visual Studio

защиту от влияния других. Например, работа бизнес-правил не должна нарушаться из-за изменения формата веб-страницы или схемы базы данных. Изменения в одной части системы не должны нарушать работу не связанных с ней других частей системы. Наши системы не должны быть настолько хрупкими.

Архитектура плагинов в наших системах создает защитные барьеры, препятствующие распространению изменений. Если графический интерфейс подключается к бизнес-правилам, изменения в графическом интерфейсе не смогут повлиять на бизнес-правила.

Границы проводятся там, где есть ось изменения. Компоненты по разным сторонам границы изменяются с разными скоростями и по разным причинам.

Графические интерфейсы изменяются в иное время и с иной скоростью, чем бизнес-правила, поэтому их должна

разделять граница. Бизнес-правила изменяются в иное время и по иным причинам, чем фреймворки внедрения зависимостей, поэтому их должна разделять граница.

Это снова простой принцип единственной ответственности, подсказывающий, где провести границы.

Заключение

Прежде чем провести линии границ в архитектуре программного обеспечения, систему нужно разделить на компоненты. Некоторые из этих компонентов реализуют основные бизнес-правила; другие являются плагинами, содержащими функции, которые не имеют прямой связи с бизнес-правилами. Затем можно организовать код в компонентах так, чтобы стрелки между ними указывали в одном направлении — в сторону бизнес-правил.

В этом без труда можно заметить принципы инверсии зависимостей (Dependency Inversion Principle) и устойчивости абстракций (Stable Abstractions Principle). Стрелки зависимостей направлены от низкоуровневых деталей в сторону высокоуровневых абстракций.

[36](#) Слово «архитектура» взято в кавычки, потому что трехуровневая архитектура в действительности не является архитектурой — это топология. Вот вам яркий пример решения, принятие которого хорошая архитектура стремится отложить.

[37](#) Много лет спустя мы смогли перенести фреймворк Velocity на FitNesse.

18. Анатомия границ



Архитектура системы определяется множеством программных компонентов и границами, разделяющими их, которые могут принимать самые разные формы. В этой главе мы рассмотрим некоторые наиболее типичные из них.

Пересечение границ

Пересечение границы во время выполнения — это не что иное, как вызов функции из другой функции, находящейся по другую сторону границы и передающей некоторые данные. Вся хитрость создания подобающих пересечений границ

заключается в управлении зависимостями на уровне исходного кода.

Почему исходного кода? Потому что, когда изменяется один модуль с исходным кодом, может потребоваться изменить или перекомпилировать другие модули и затем повторно развернуть их. Создание и управление барьерами, защищающими от таких изменений, — вот главная цель проведения границ.

Ужасный монолит

Наиболее простые и типичные архитектурные границы не имеют явного физического представления. Это просто организационное разделение функций и данных, действующих и находящихся в одном процессе, в общем адресном пространстве. В предыдущей главе я назвал это режимом разделения на уровне исходного кода.

С точки зрения развертывания это единый выполняемый файл — так называемый монолит. Этот файл может быть статически скомпонованным проектом на С или С++, множеством файлов классов Java, объединенных в выполняемый jar-файл, множеством двоичных ресурсов .NET, объединенных в один выполняемый файл .EXE, и т.д.

Невидимость границ на этапе развертывания монолита не означает, что они отсутствуют или не играют значительной роли. Даже когда элементы программы статически компонуются в один выполняемый файл, возможность независимой разработки разных компонентов очень ценна для окончательной сборки.

Такие архитектуры почти всегда зависят от некоторой разновидности динамического полиморфизма³⁸, используемой для управления внутренними зависимостями.

Это одна из причин, почему объектно-ориентированная парадигма приобрела такую важность в последние десятилетия. В отсутствие объектно-ориентированной или эквивалентной формы полиморфизма архитекторы вынуждены возвращаться к порочной практике использования указателей на функции, чтобы добиться требуемого разделения. Большинство архитекторов считают, что использование указателей для ссылки на функции слишком рискованно, и они вынужденно отказываются от любых видов разделения на компоненты.

Простейшим пересечением границы является вызов низкоуровневым клиентом функции в высокоуровневой службе. Обе зависимости — времени выполнения и времени компиляции — указывают в одном направлении, в сторону высокоуровневого компонента.

На рис. 18.1 изображен поток управления, пересекающий границу слева направо. Компонент `Client` вызывает функцию `f()`, находящуюся в компоненте `Service`. Выполняя вызов, он передает экземпляр данных `Data`. Метка `<DS>` просто сообщает, что это структура данных (Data Structure). Структура `Data` может передаваться как аргумент функции или каким-то иным, более сложным способом. Обратите внимание, что определение структуры `Data` находится на *вызываемой* стороне.

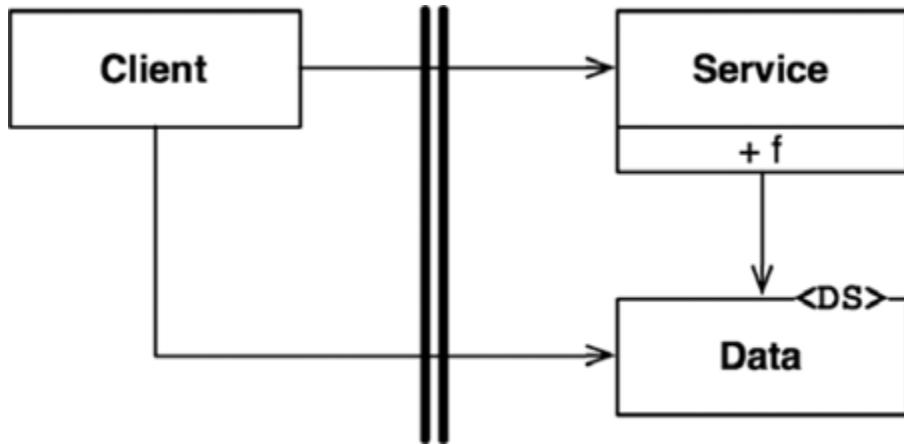


Рис. 18.1. Поток управления пересекает границу в направлении от нижнего уровня к верхнему

Когда требуется организовать вызов низкоуровневой службы из высокоуровневого клиента, для обращения зависимости потока управления используется динамический полиморфизм. Зависимость времени выполнения в этом случае имеет направление, противоположное зависимости времени компиляции.

На рис. 18.2 изображен поток управления, пересекающий границу слева направо, как и прежде. Высокоуровневый компонент `Client` вызывает функцию `f()`, находящуюся в низкоуровневом компоненте `ServiceImpl`, посредством интерфейса `Service`. Но обратите внимание, что все зависимости пересекают границу в направлении справа налево и указывают в сторону высокогоуровневого компонента. Отметьте также, что теперь определение структуры данных находится на вызывающей стороне.

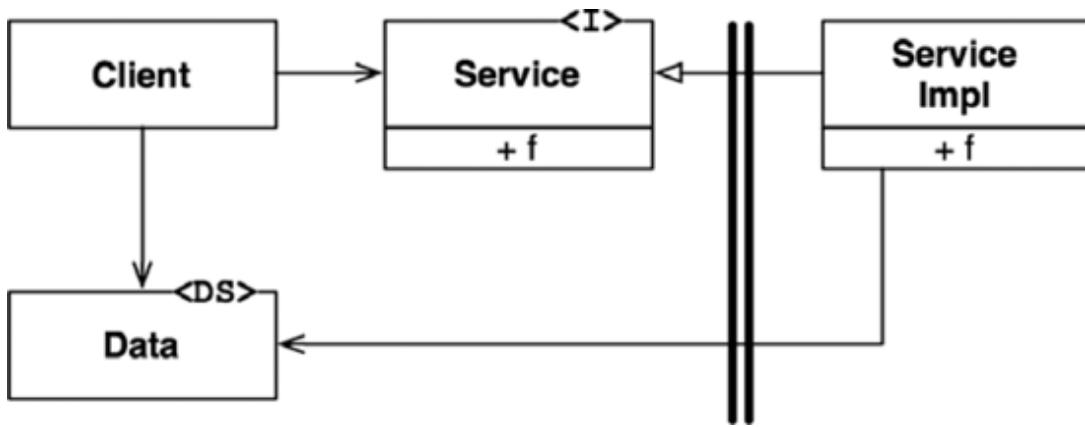


Рис. 18.2. Пересечение границы в направлении, противоположном потоку управления

Даже в монолитном, статически скомпонованном выполняемом файле такой вид организационного разделения значительно помогает в разработке, тестировании и развертывании проекта. Команды разработчиков могут трудиться независимо друг от друга и не наступая друг другу на пятки. Высокоуровневые компоненты остаются независимыми от низкоуровневых деталей.

Взаимодействия между компонентами в монолите протекают быстро и эффективно. Обычно они сводятся к простым вызовам функций. Как следствие, взаимодействия через границы, проведенные на уровне исходного кода, могут быть очень обширными.

Поскольку для развертывания монолита обычно требуется выполнить компиляцию и статическую компоновку, компоненты в таких системах часто поставляются в исходном коде.

Компоненты развертывания

Простейшим примером физического представления архитектурной границы может служить динамическая библиотека DLL в .Net, jar-файл в Java, gem-файл в Ruby или

разделяемая библиотека (.so) в UNIX. Развёртывание в этом случае не связано с компиляцией — компоненты поставляются в двоичном виде или в иной форме, пригодной к непосредственному развертыванию. Это режим разделения на уровне развертывания. Акт развертывания заключается в простой сборке единиц развертывания в удобную форму, например WAR-файл, или даже в обычном копировании файлов в каталог.

Кроме этого единственного исключения, компоненты уровня развертывания остаются теми же, что и в монолите. Все функции обычно действуют в одном процессе и в общем адресном пространстве. Стратегии разделения компонентов и управления их зависимостями не меняются³⁹.

Так же как в монолите, взаимодействия между границами развертываемых компонентов часто осуществляются посредством вызовов функций и, соответственно, обходятся очень дешево. Динамическая компоновка или загрузка во время выполнения могут однократно увеличивать потребление вычислительных ресурсов, но сами взаимодействия через границы все еще могут быть очень обширными.

Потоки выполнения

Монолиты и компоненты развертывания могут выполняться в многопоточном режиме. Потоки выполнения не являются архитектурными границами или единицами развертывания, это способ организации планирования и выполнения задач. Они могут вообще не выходить за рамки компонента или охватывать сразу несколько компонентов.

Локальные процессы

Локальные процессы представляют более надежные физические архитектурные границы. Обычно локальный процесс запускается из командной строки или с помощью эквивалентного системного вызова. Локальные процессы выполняются на одном процессоре или группе процессоров (в многопроцессорной системе), но в разных адресных пространствах. Механизмы защиты памяти обычно не позволяют таким процессам совместно использовать одну и ту же область памяти, хотя для взаимодействий нередко используются сегменты разделяемой памяти.

Чаще всего локальные процессы взаимодействуют друг с другом посредством сокетов или других средств связи, поддерживаемых операционной системой, таких как почтовые ящики или очереди сообщений.

Каждый локальный процесс может быть статически скомпонованным монолитом или динамически компонуемой группой компонентов развертывания. В первом случае несколько монолитных процессов могут содержать одни и те же компоненты, скомпилированные и скомпонованные в них. Во втором — они могут совместно использовать динамически составляемые компоненты развертывания.

Локальный процесс можно считать своеобразным суперкомпонентом: процесс состоит из низкоуровневых компонентов и управляет их зависимостями с помощью динамического полиморфизма.

Стратегия разделения между локальными процессами остается той же, что для монолитов и двоичных компонентов. Зависимости в исходном коде направлены в одну сторону через границу и всегда в сторону высокоуровневого компонента.

Для локальных процессов это означает, что исходный код высокоуровневых процессов не должен содержать имен, физических адресов или ключей в реестре, соответствующих

низкоуровневым процессам. Не забывайте, что главная архитектурная цель — сделать низкоуровневые процессы плагинами (сменными модулями) для высокоуровневых процессов.

Взаимодействия через границы локальных процессов связаны с обращением к системным вызовам, маршалингом и декодированием данных, а также переключением контекстов и обходятся умеренно дорого. Количество таких взаимодействий следует тщательно ограничивать.

Службы

Самыми надежными являются границы служб. Часто служба — это процесс, который запускается из командной строки или с помощью эквивалентного системного вызова. Службы не зависят от физического местоположения. Две взаимодействующие службы могут или не могут действовать на одном процессоре или группе процессоров в многопроцессорной системе. Службы предполагают, что все взаимодействия осуществляются по сети.

Взаимодействия через границы служб осуществляются очень медленно в сравнении с вызовами функций. Время между запросом и ответом может составлять от десятков миллисекунд до нескольких секунд. Взаимодействия следует ограничивать по мере возможностей. Взаимодействия на этом уровне должны учитывать возможность больших задержек.

В остальном к службам применяются те же правила, что и к локальным процессам. Низкоуровневые службы должны «подключаться» к высокоуровневым службам. Исходный код высокоуровневых служб не должен содержать никакой конкретной информации (например, URI) о низкоуровневых службах.

Заключение

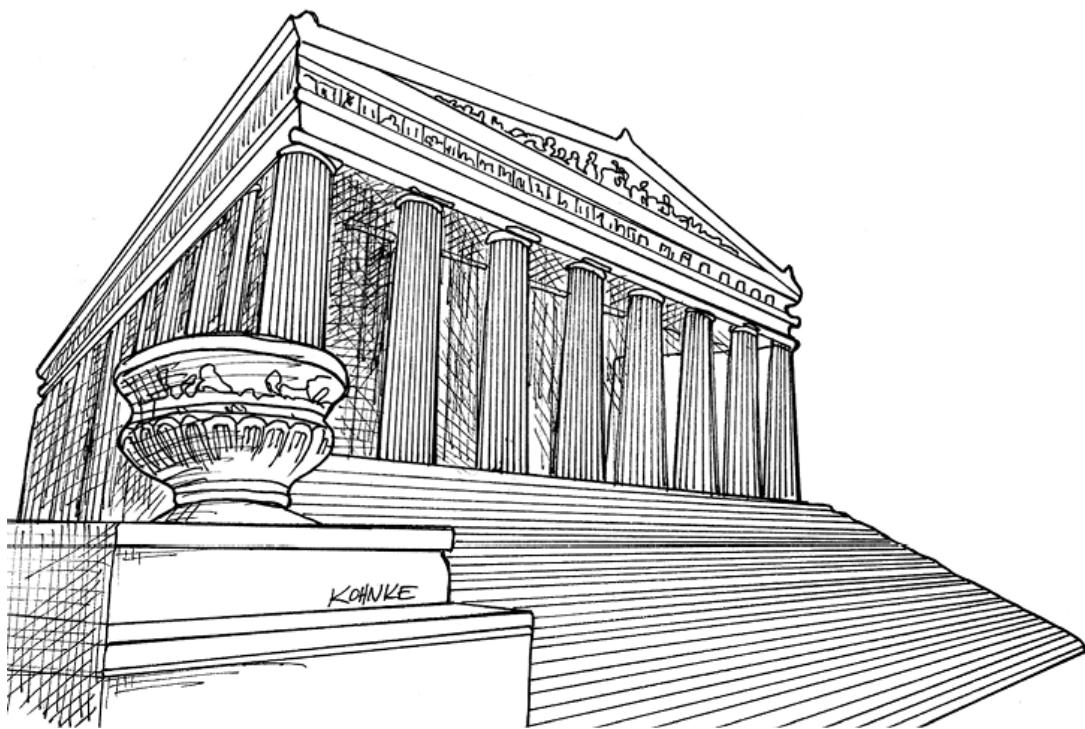
Большинство систем, кроме монолитных, используют несколько стратегий разграничения. Система, состоящая из служб, может также иметь несколько локальных процессов. В действительности служба часто является всего лишь фасадом для нескольких взаимодействующих локальных процессов. Служба или локальный процесс чаще имеют форму монолита, составленного из исходных кодов компонентов, или группы динамически подключаемых компонентов развертывания.

Это означает, что границы в системе часто будут представлены смесью локальных границ (недорогих в пересечении) и границ, страдающих задержками.

[38](#) Статический полиморфизм (например, обобщенные типы или шаблонные классы) иногда оказывается ценным инструментом управления зависимостями в монолитных системах, особенно в таких языках, как C++. Однако разделение, выполненное с применением обобщенных типов, не способно защитить вас от необходимости выполнять повторные компиляцию и развертывание, как это может динамический полиморфизм.

[39](#) Хотя в этом случае статический полиморфизм даже нельзя рассматривать как вариант.

19. Политика и уровень



Программные системы — это заявления о направлении действий. Фактически любая компьютерная программа является таким заявлением. Компьютерная программа — это подробное описание политики преобразования входных данных в выходные.

В большинстве нетривиальных систем общую политику можно разбить на множество более мелких заявлений. Некоторые из этих заявлений могут описывать действие конкретных бизнес-правил. Другие могут определять оформление отчетов. А третьи — описывать порядок проверки входных данных.

Отчасти искусство создания программных архитектур заключается в отделении этих политик друг от друга и их перегруппировке с учетом способов их изменения. Политики,

изменяющиеся по одинаковым причинам и в одно время, находятся на одном уровне и принадлежат одному компоненту. Политики, изменяющиеся по разным причинам или в разное время, находятся на разных уровнях и должны помещаться в разные компоненты.

Искусство создания программных архитектур нередко связано с организацией перегруппированных компонентов в ориентированный ациклический граф. Узлами такого графа являются компоненты, содержащие политики одного уровня. А ориентированными ребрами — зависимости между компонентами. Они соединяют компоненты, находящиеся на разных уровнях.

К ним относятся зависимости на уровне исходного кода или времени компиляции. В Java они выражаются инструкциями `import`. В C# — инструкциями `using`. В Ruby — инструкциями `require`. Все эти зависимости необходимы компилятору для выполнения своей работы.

В хорошей архитектуре направление этих зависимостей обусловлено уровнем компонентов, которые они соединяют. В любом случае низкоуровневые компоненты проектируются так, чтобы они зависели от высокоуровневых компонентов.

Уровень

Термин «уровень» имеет строгое определение: «удаленность от ввода и вывода». Чем дальше политика от ввода и вывода, тем выше ее уровень. Политики, управляющие вводом и выводом, являются самыми низкоуровневыми в системе.

Диаграмма потоков данных на рис. 19.1 соответствует простой программе шифрования, которая читает символы из устройства ввода, преобразует их с использованием таблицы и записывает преобразованные символы в устройство вывода.

Направления потоков данных показаны на диаграмме извилистыми сплошными стрелками. Правильно спроектированные зависимости в исходном коде показаны пунктирными стрелками.

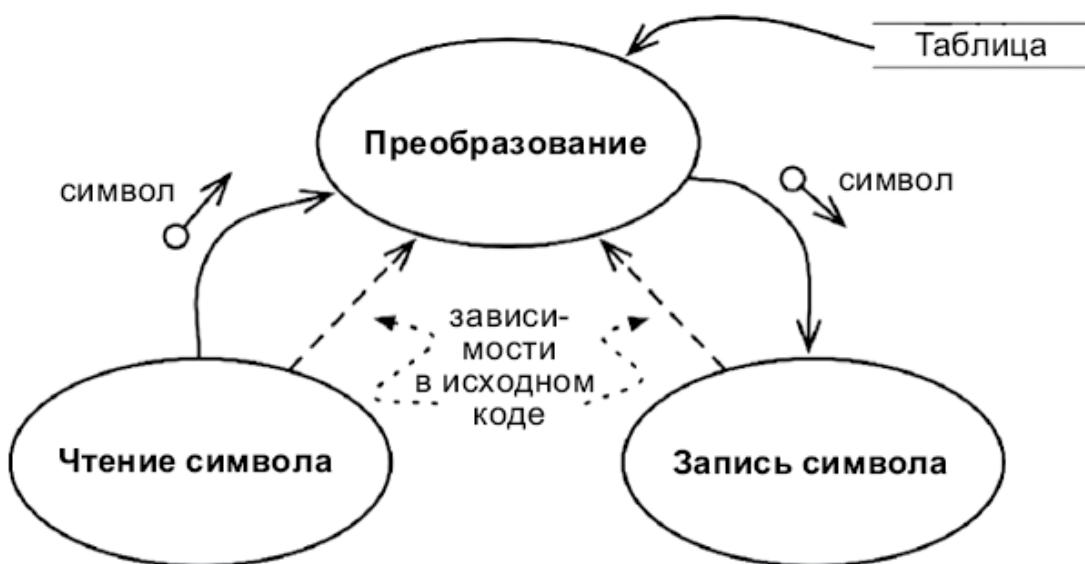


Рис. 19.1. Простая программа шифрования

Компонент, выполняющий преобразование, — это самый высокоуровневый компонент в данной системе, потому что он дальше других находится от ввода и вывода⁴⁰.

Обратите внимание, что потоки данных и зависимости в исходном коде не всегда указывают в одном направлении. Это тоже один из элементов искусства создания программных архитектур. Мы должны отделить зависимости в исходном коде от потоков данных и связать с уровнем.

Мы легко могли бы создать неправильную архитектуру, реализовав программу так:

```
function encrypt() {  
    while(true)  
        writeChar(translate(readChar()));
```

}

Это неправильная архитектура, потому что высокоуровневая функция `encrypt` зависит от низкоуровневых функций `readChar` и `writeChar`.

Более удачная архитектура для этой системы изображена на рис. 19.2. Обратите внимание на пунктирную границу, окружающую класс `Encrypt`, и интерфейсы `CharWriter` и `CharReader`. Все зависимости, пересекающие границу, указывают внутрь. Этот модуль является элементом высшего уровня в системе.

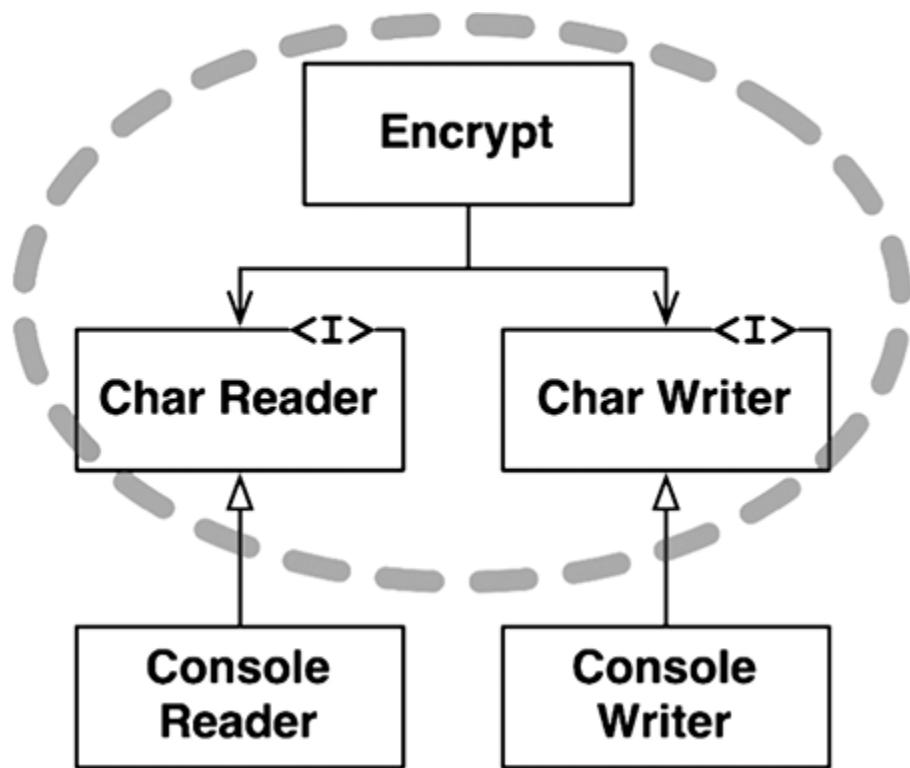


Рис. 19.2. Диаграмма классов, демонстрирующая более удачную архитектуру для системы

На диаграмме также изображены классы `ConsoleReader` и `ConsoleWriter`. Они находятся на более низком уровне, потому что расположены ближе к вводу и выводу.

Обратите внимание, как эта структура отделяет высокоуровневую политику шифрования от низкоуровневых политик ввода/вывода. Это позволяет использовать политику шифрования в широком диапазоне контекстов. Когда в политиках ввода и вывода происходят изменения, они никак не затрагивают политику шифрования.

Помните, что политики группируются в компоненты по способам изменения. Политики, изменяющиеся по одним причинам и в одно время, объединяются в соответствии с принципами единственной ответственности (SRP) и согласованного изменения (CCP). Чем дальше политика от ввода и вывода, тем выше ее уровень и тем реже она изменяется и по более важным причинам. Чем ближе политика к вводу и выводу, тем ниже ее уровень и тем чаще она изменяется и по более неотложным, но менее важным причинам.

Например, даже в тривиальной программе шифрования вероятность изменения устройств ввода/вывода намного выше, чем вероятность изменения алгоритма шифрования. Изменение алгоритма шифрования наверняка будет обусловлено более серьезной причиной, чем изменение устройств ввода/вывода.

Отделение политик друг от друга и организация зависимостей в исходном коде так, что все они направлены в сторону политик более высокого уровня, уменьшает влияние изменений. Тривиальные, но срочные изменения на более низких уровнях системы не влияют или слабо влияют на более высокие уровни.

На эту проблему можно взглянуть с другой стороны, если вспомнить, что низкоуровневые компоненты должны быть плагинами для высокоуровневых компонентов. Этот взгляд демонстрирует диаграмма компонентов на рис. 19.3.

Компонент Encryption ничего не знает о компоненте IODevices; а компонент IODevices зависит от компонента Encryption.



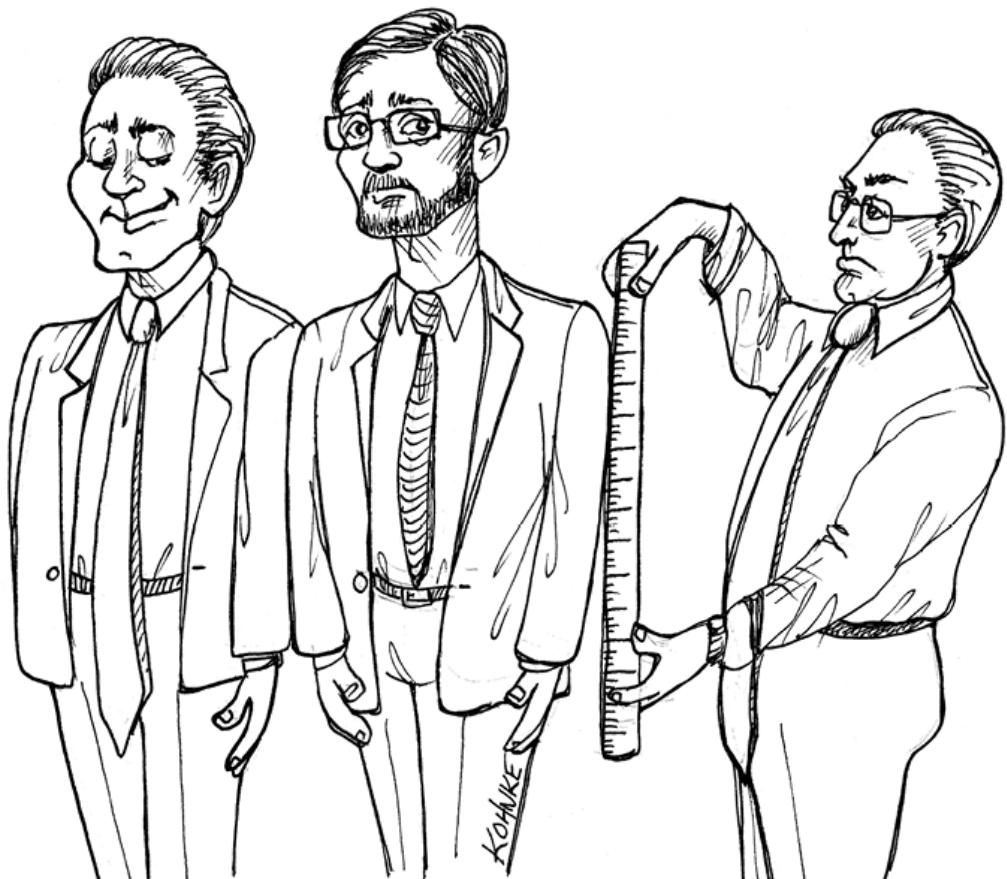
Рис. 19.3. Низкоуровневые компоненты должны быть плагинами для высокоуровневых компонентов

Заключение

На этом этапе в обсуждение политик были вовлечены принципы единственной ответственности (SRP), открытости/закрытости (OCP), согласованного изменения (CCP), инверсии зависимостей (DIP), устойчивых зависимостей (SDP) и устойчивости абстракций (SAP). А теперь вернитесь назад и посмотрите, сможете ли вы определить, где каждый из принципов используется и почему.

[40](#) Меилир Пейдж-Джонс (Meilir Page-Jones) назвал этот компонент «центральным преобразованием» в своей книге *The Practical Guide to Structured Systems Design*, 2nd ed., Yourdon Press, 1988.

20. Бизнес-правила



Прежде чем пытаться делить приложение на бизнес-правила и плагины, необходимо понять, какие бизнес-правила существуют. Как оказывается, их несколько видов.

Строго говоря, бизнес-правила — это правила или процедуры, делающие или экономящие деньги. Еще строже говоря, бизнес-правила — это правила, делающие или экономящие деньги независимо от наличия или отсутствия их реализации на компьютере. Они делают или экономят деньги, даже когда выполняются вручную.

Банк взимает $N\%$ за кредит — это бизнес-правило, которое приносит банку деньги. И неважно, имеется ли компьютерная

программа, вычисляющая процент, или служащий вычисляет его на счетах.

Мы будем называть такие правила *критическими бизнес-правилами*, потому что они имеют решающее значение для бизнеса и будут существовать даже в отсутствие системы, автоматизирующей их.

Критические бизнес-правила обычно требуют каких-то данных для работы. Например, в случае с кредитом нужно иметь сумму остатка, процентную ставку и график платежей.

Мы будем называть такие данные *критическими бизнес-данными*. Эти данные существуют даже в отсутствие системы автоматизации.

Критические правила и критические данные неразрывно связаны друг с другом, поэтому являются отличными кандидатами на объединение в объект. Мы будем называть такие объекты *сущностями*⁴¹.

Сущности

Сущность — это объект в компьютерной системе, воплощающий небольшой набор критических бизнес-правил, оперирующих критическими бизнес-данными. Объект-сущность или содержит критические бизнес-правила в себе, или имеет простой доступ к ним. Интерфейс сущности состоит из функций, реализующих критические бизнес-правила и оперирующих этими данными.

Например, на рис. 20.1 показано, как могла бы выглядеть сущность *Loan* (представляющая банковский кредит) в виде класса на диаграмме UML. Она включает три фрагмента взаимосвязанных критических бизнес-данных и реализует

интерфейс с тремя взаимосвязанными критическими бизнес-правилами.

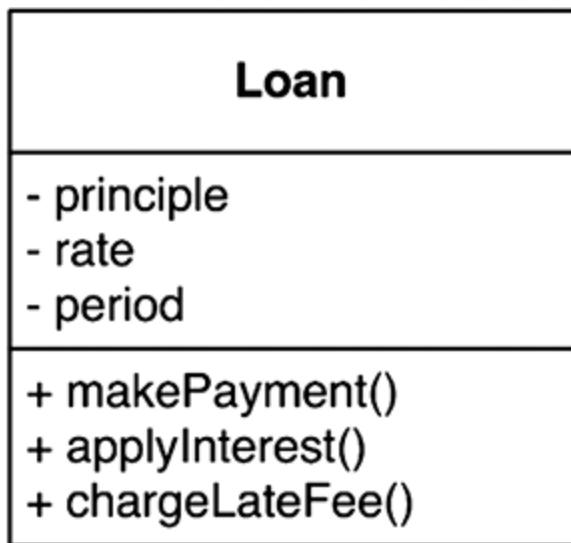


Рис. 20.1. Сущность Loan в виде класса на диаграмме UML

Создавая такой класс, мы объединяем программную реализацию идеи, имеющей решающее значение для бизнеса, и отделяем ее от остальных задач в создаваемой нами системе автоматизации. Этот класс играет роль представителя бизнеса. Он не зависит от выбора базы данных, пользовательского интерфейса или сторонних фреймворков. Он может служить целям бизнеса в любой системе, независимо от того, какой пользовательский интерфейс она имеет, как хранит данные или как организованы компьютеры в этой системе. Сущность — это бизнес в чистом виде и больше ничего.

Кто-то из вас, возможно, забеспокоился, когда я назвал сущность классом. Не волнуйтесь. Чтобы создать сущность, не требуется использовать объектно-ориентированный язык. Необходимо лишь связать воедино критические бизнес-данные с критическими бизнес-правилами и выделить их в отдельный программный модуль.

Варианты использования

Не все бизнес-правила так же чисты, как сущности. Некоторые бизнес-правила делают или экономят деньги, определяя и ограничивая деятельность *автоматизированной* системы. Эти правила не могут выполняться вручную, потому что имеют смысл только как часть автоматизированной системы.

Например, представьте приложение, используемое служащими банка для оформления нового кредита. Банк может решить, что сотрудники, оформляющие кредиты, не должны предлагать график погашения кредита, пока не соберут и не проверят контактную информацию и не убедятся, что кандидат имеет кредитный балл 500 или выше. По этой причине банк может потребовать, чтобы система не отображала форму с графиком платежей, пока не будет заполнена и проверена форма с контактной информацией и не придет подтверждение, что кредитный балл клиента выше требуемого порога.

Это *вариант использования*⁴². Вариант использования описывает способ использования автоматизированной системы. Он определяет, что должен ввести пользователь, что должно быть выведено в ответ и какие действия должны быть выполнены для получения выводимой информации. В отличие от критических бизнес-правил внутри сущностей, вариант использования описывает бизнес-правила, характерные для конкретного приложения.

На рис. 20.2 изображен пример варианта использования. Обратите внимание, что в последней строке упоминается Клиент. Это ссылка на сущность Клиент, содержащую критические бизнес-правила, регулирующие отношения между банком и клиентами.

Порядок сбора информации для оформления кредита

Ввод: имя, адрес, дата рождения, водительские права, номер карты социального страхования и пр.
Выход: Некоторая информация для обратной связи + кредитный балл.

Порядок действий:

1. Принять и проверить имя.
2. Проверить адрес, дату рождения, водительские права, номер карты социального страхования и пр.
3. Узнать кредитный балл.
4. Если балл < 500, отклонить заявку на кредит
5. Иначе создать Клиента и активировать график погашения

Рис. 20.2. Пример варианта использования

Варианты использования определяют, как и когда вызываются критические бизнес-правила в сущности. Варианты использования управляют действиями сущности.

Отмечу также, что варианты использования не описывают пользовательский интерфейс, они лишь неформально определяют входные и выходные данные, поступающие и возвращаемые через интерфейс. По вариантам использования нельзя определить, является ли данная система веб-приложением, толстым клиентом, утилитой командной строки или чистой службой.

Это очень важно. Варианты использования не описывают, как выглядит система для пользователя. Они описывают только конкретные правила работы приложения, определяющие порядок взаимодействий между пользователями и

сущностями. Для вариантов использования абсолютно неважно, как система осуществляет ввод/вывод данных.

Вариант использования — это объект. Он имеет одну или несколько функций, реализующих конкретные прикладные бизнес-правила. Он также имеет элементы данных, включая входные данные, выходные данные и ссылки на соответствующие сущности, с которыми он взаимодействует.

Сущности не знают ничего о вариантах использования, контролирующих их. Это еще один пример ориентации зависимостей в соответствии с принципом инверсии зависимостей (Dependency Inversion Principle). Высокоуровневые элементы, такие как сущности, ничего не знают о низкоуровневых элементах, таких как варианты использования. Напротив, низкоуровневые варианты использования знают все о высокогоревневых сущностях.

Почему сущности относятся к высокому уровню, а варианты использования к низкому? Потому что варианты использования характерны для единственного приложения и, соответственно, ближе к вводу и выводу системы. Сущности — это обобщения, которые можно использовать в множестве разных приложений, соответственно, они дальше от ввода и вывода системы. Варианты использования зависят от сущностей; сущности не зависят от вариантов использования.

Модели запросов и ответов

Варианты использования принимают входные данные и возвращают результат. Однако в правильно сформированном объекте варианта использования ничто не должно говорить о способах передачи данных пользователю или другим компонентам. И тем более код в классе, реализующем вариант использования, ничего не должен знать об HTML или SQL!

Класс варианта использования принимает простые структуры данных на входе и возвращает простые структуры данных на выходе. Эти структуры данных ни от чего не зависят. Они не связаны со стандартными интерфейсами фреймворков, такими как `HttpRequest` и `HttpResponse`. Они ничего не знают о Веб и не используют никакие атрибуты пользовательского интерфейса.

Такое отсутствие зависимостей имеет решающее значение. Если модели запросов и ответов зависят от чего-то, варианты использования, зависящие от них, оказываются косвенно связанными с зависимостями, которые вносят эти модели.

У вас может возникнуть соблазн включить в эти структуры ссылки на объекты сущностей. На первый взгляд кажется, что в этом есть определенный смысл, потому что сущности и модели запросов/ответов совместно используют так много данных. Боритесь с этим искушением! Эти два объекта имеют слишком разные цели. Со временем они будут меняться по разным причинам, поэтому создание связи между ними нарушит принципы согласованного изменения (CCP) и единственной ответственности (SRP). В результате у вас появится множество кочующих данных и масса условных инструкций в коде.

Заключение

Бизнес-правила являются причиной существования программной системы. Они составляют основу функционирования. Они порождают код, который делает или экономит деньги. Они — наши семейные реликвии.

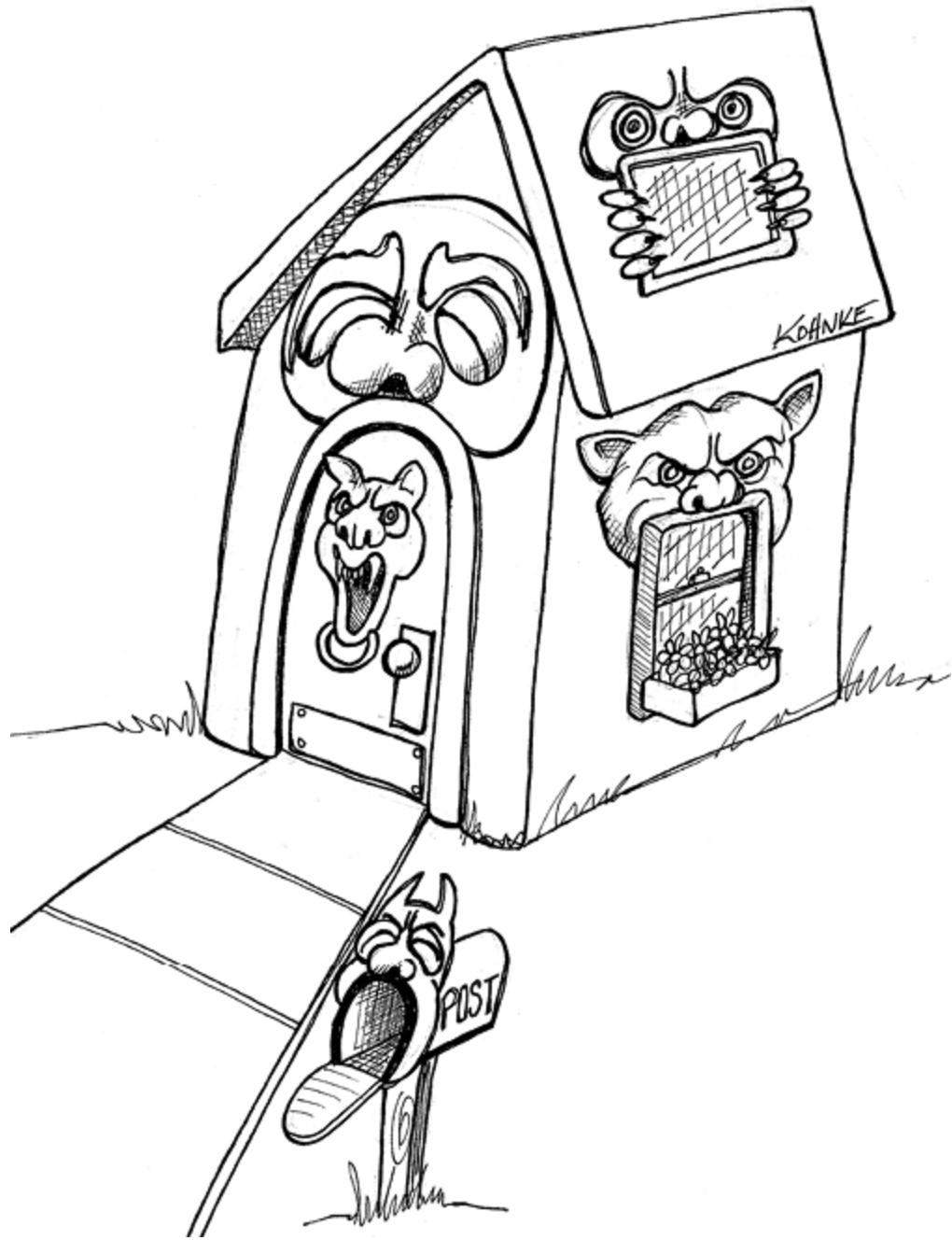
Бизнес-правила должны оставаться в неприкасаемости, незапятнанными низкоуровневыми аспектами, такими как пользовательский интерфейс или база данных. В идеале код, представляющий бизнес-правила, должен быть сердцем

системы, а другие задачи — просто подключаться к ним. Реализация бизнес-правил должна быть самым независимым кодом в системе, готовым к многоократному использованию.

[41](#) Это название предложил Ивар Якобсон (Ivar Jacobson, *Object Oriented Software Engineering*, Addison-Wesley, 1992).

[42](#) Это название тоже было предложено Иваром Якобсоном в той же книге.

21. Кричащая архитектура



Представьте, что перед вами чертежи здания. Они подготовлены архектором и представляют собой планы здания. О чем они вам говорят?

Если это чертежи коттеджа, вы наверняка увидите на них парадный вход, вестибюль, ведущий в гостиную и, может быть, в столовую. Рядом со столовой, скорее всего, будет расположена кухня. Рядом с кухней, вероятно, будет малая столовая и где-то поблизости — общая комната. При рассмотрении этих планов у вас не возникнет сомнений, что вы видите коттедж для одной семьи. Архитектура как бы кричит: «Это коттедж».

А теперь представьте, что вы рассматриваете архитектуру библиотеки. Вы наверняка увидите центральный вход, места для размещения библиотекарей, читальные залы, небольшие конференц-залы и галереи для хранения книг. Архитектура кричит: «Это библиотека».

А что кричит архитектура вашего приложения? Увидев высокоуровневую структуру каталогов и пакетов с исходным кодом, услышите ли вы, как она кричит: «Это медицинская система», или «Это система учета», или «Это система управления складским хозяйством»? Или вы услышите: «Rails», или «Spring/Hibernate», или «ASP»?

Тема архитектуры

Отвлекитесь и прочтайте фундаментальный труд Ивара Якобсона об архитектуре программного обеспечения: книгу *Object Oriented Software Engineering*⁴³. Обратите внимание на подзаголовок книги: *A Use Case Driven Approach*⁴⁴. В этой книге Якобсон подчеркивает, что архитектура программного обеспечения — это структура, поддерживающая варианты использования системы. В частности как план библиотеки или коттеджа кричит о назначении здания, архитектура программного обеспечения должна кричать о вариантах использования приложения.

Архитектуры не связаны (и не должны быть связаны) с фреймворками. Архитектура не должна определяться фреймворками. Фреймворки — это инструменты, а вовсе не аспекты, определяющие черты архитектуры. Если ваша архитектура опирается на фреймворки, она не сможет опираться на варианты использования.

Цель архитектуры

Хорошие архитектуры опираются на варианты использования и помогают архитекторам описывать структуры, поддерживающие эти варианты использования, не связывая себя фреймворками, инструментами и окружениями. Взгляните на план коттеджа еще раз. Одной из первоочередных забот архитектора является удобная планировка дома, а не выбор материала для стен. Действительно, архитектор прилагает максимум усилий, чтобы дать домовладельцу возможность самому выбрать материал (кирпич, камень или дерево), потом, когда планы, соответствующие вариантам использования, уже будут начертены.

Хорошая архитектура позволяет отложить решение о выборе фреймворков, баз данных, веб-серверов и других инструментов и элементов окружения. *Фреймворки относятся к возможностям, которые должны оставаться открытыми.* Хорошая архитектура позволяет отложить выбор Rails, Spring, Hibernate, Tomcat или MySQL на отдаленный срок. Хорошая архитектура позволяет с легкостью менять подобные решения. Хорошая архитектура подчеркивает варианты использования и отделяет их от второстепенных проблем.

А что насчет Веб?

Веб — это архитектура? Зависит ли архитектура системы от того факта, что свои услуги она предоставляет через Веб? Конечно нет! Всемирная паутина (или Веб) — это механизм доставки, устройство ввода/вывода. И архитектура вашего приложения должна интерпретировать его именно так, а не иначе. Факт предоставления услуг через Веб — это деталь, и она не должна определять структуру системы. В действительности решение о предоставлении услуг через Веб относится к разряду решений, которые должны откладываться. Архитектура системы должна быть максимально нейтральной к механизмам доставки услуг. У вас должна иметься возможность реализовать систему в форме консольного приложения, веб-приложения, толстого клиента или даже веб-службы без чрезмерного усложнения или изменения основной архитектуры.

Фреймворки — это инструменты, а не образ жизни

Фреймворки могут быть очень мощными и удобными. Авторы фреймворков часто глубоко убеждены в своих творениях. Примеры, которые они пишут, рассказывают об использовании их фреймворков с точки зрения истинно верующих. Другие авторы, пишущие книги о фреймворках, тоже часто стоят на позициях истинной веры. Они показывают способы использования фреймворков, часто с позиции всеобъемлющего, всепроникающего и повсеместного применения.

Вы не должны вставать на эту позицию.

Рассматривайте фреймворки с холодком. Смотрите на них скептически. Да, они могут помочь, но какой ценой? Спросите себя, как бы вы их использовали и как защитились бы от них.

Подумайте, как сохранить приоритет вариантов использования для архитектуры. Разработайте стратегию, не позволяющую фреймворку влиять на архитектуру.

Тестируемые архитектуры

Если архитектура вашей системы основана исключительно на вариантах использования и вам удалось удержать фреймворки на расстоянии, у вас должна иметься возможность протестировать все эти варианты использования без привлечения фреймворков. Вы не должны испытывать потребности в веб-сервере для выполнения тестов. Вы не должны испытывать потребности в подключении к базе данных для выполнения тестов. Ваши сущности должны быть самыми обычными объектами, не зависящими от фреймворков, баз данных или чего-то другого. Ваши объекты вариантов использования должны координировать действия сущностей. Наконец, должна иметься возможность протестировать их вместе без привлечения любых фреймворков.

Заключение

Ваша архитектура должна рассказывать о системе, а не о фреймворках, использованных в системе. Если вы работаете над системой для медицинского учреждения, тогда первый же взгляд на репозиторий с исходным кодом должен вызывать у новых программистов, подключающихся к проекту, мысль: «Да, это медицинская система». Новые программисты должны иметь возможность изучить все возможные варианты использования, даже не зная, как будут доставляться услуги системы. Они могут подойти к вам и сказать:

«Мы видим что-то похожее на модели, но где контроллеры и представления?»

А вы должны ответить:

«Это такая мелочь, которая нас пока не заботит. Мы решим этот вопрос позже».

[43](#) Проектирование объектно-ориентированного программного обеспечения. — *Примеч. пер.*

[44](#) Подход на основе вариантов использования. — *Примеч. пер.*

22. Чистая архитектура



За последние несколько десятилетий мы видели целый ряд идей об организации архитектур. В том числе:

- Гексагональная архитектура (Hexagonal Architecture, также известна как архитектура портов и адаптеров), разработанная Алистером Кокберном и описанная Стивом Фриманом и Натом Прайсом в их замечательной книге *Growing Object Oriented Software with Tests*.
- DCI⁴⁵, предложенная Джеймсом Коплиеном и Трюгве Реенскаугом.
- BCE⁴⁶, предложенная Иваром Якобсоном в книге *Object Oriented Software Engineering: A Use-Case Driven Approach*.

Несмотря на различия в деталях, все эти архитектуры очень похожи. Они все преследуют одну цель — разделение задач. Они все достигают этой цели путем деления программного обеспечения на уровни. Каждая имеет хотя бы один уровень для бизнес-правил и еще один для пользовательского и системного интерфейсов.

Каждая из этих архитектур способствует созданию систем, обладающих следующими характеристиками:

- *Независимость от фреймворков.* Архитектура не зависит от наличия какой-либо библиотеки. Это позволяет рассматривать фреймворки как инструменты, вместо того чтобы стараться втиснуть систему в их рамки.
- *Простота тестирования.* Бизнес-правила можно тестировать без пользовательского интерфейса, базы данных, веб-сервера и любых других внешних элементов.
- *Независимость от пользовательского интерфейса.* Пользовательский интерфейс можно легко изменять, не затрагивая остальную систему. Например, веб-интерфейс можно заменить консольным интерфейсом, не изменяя бизнес-правил.
- *Независимость от базы данных.* Вы можете поменять Oracle или SQL Server на Mongo, BigTable, CouchDB или что-то еще. Бизнес-правила не привязаны к базе данных.
- *Независимость от любых внешних агентов.* Ваши бизнес-правила ничего не знают об интерфейсах, ведущих во внешний мир.

Диаграмма на рис. 22.1 — это попытка объединить все эти архитектуры в одну практически осуществимую идею.



Рис. 22.1. Чистая архитектура

Правило зависимостей

Концентрические круги на рис. 22.1 представляют разные уровни программного обеспечения. Чем ближе к центру, тем выше уровень. Внешние круги — это механизмы. Внутренние — политики.

Главным правилом, приводящим эту архитектуру в действие, является *правило зависимостей* (Dependency Rule):

Зависимости в исходном коде должны быть направлены внутрь, в сторону высокоуровневых политик.

Ничто во внутреннем круге ничего не знает о внешних кругах. Например, имена, объявленные во внешних кругах, не должны упоминаться в коде, находящемся во внутренних кругах. Сюда относятся функции, классы, переменные и любые другие именованные элементы программы.

Точно так же форматы данных, объявленные во внешних кругах, не должны использоваться во внутренних, особенно если эти форматы генерируются фреймворком во внешнем круге. Ничто во внешнем круге не должно влиять на внутренние круги.

Сущности

Сущности заключают в себе критические бизнес-правила уровня предприятия. Сущность может быть объектом с методами или набором структур данных и функций. Сама организация не важна, если сущности доступны для использования разными приложениями на предприятии.

Если вы пишете только одно приложение и не для предприятия, эти сущности становятся бизнес-объектами приложения. Они инкапсулируют наиболее общие и высокоуровневые правила. Их изменение маловероятно под влиянием внешних изменений. Например, трудно представить, что эти объекты изменятся из-за изменения структуры навигации или безопасности страницы. Никакие изменения в работе любого конкретного приложения не должны влиять на уровень сущностей.

Варианты использования

Программное обеспечение на уровне вариантов использования содержит бизнес-правила, *характерные для приложения*. Оно инкапсулирует и реализует все варианты использования

системы. Варианты использования организуют поток данных в сущности и из них и требуют от этих сущностей использовать их критические бизнес-правила для достижения своих целей.

Изменения внутри этого уровня не должны влиять на сущности. Аналогично изменения во внешних уровнях, например в базе данных, пользовательском интерфейсе или в любом из общих фреймворков, не должны влиять на этот уровень. Уровень вариантов использования изолирован от таких проблем.

Но изменения в работе приложения безусловно повлияют на варианты использования и, соответственно, на программное обеспечение, находящееся на этом уровне. Изменение деталей вариантов использования определенно затронет некоторый код на этом уровне.

Адаптеры интерфейсов

Программное обеспечение на уровне адаптеров интерфейсов — это набор адаптеров, преобразующих данные из формата, наиболее удобного для вариантов использования и сущностей, в формат, наиболее удобный для некоторых внешних агентов, таких как база данных или веб-интерфейс. Именно на этом уровне целиком находится архитектура MVC графического пользователяского интерфейса. Презентаторы, представления и контроллеры — все принадлежат уровню адаптеров интерфейсов. Модели — часто лишь структуры данных, которые передаются из контроллеров в варианты использования и затем обратно из вариантов использования в презентаторы и представления.

Аналогично на этом уровне преобразуются данные из формата, наиболее удобного для вариантов использования и сущностей, в формат, наиболее удобный для инфраструктуры

хранения данных (например, базы данных). Никакой код, находящийся в других внутренних кругах, не должен ничего знать о базе данных. Если данные хранятся в базе данных SQL, тогда весь код на языке SQL должен находиться именно на этом уровне, точнее, в элементах этого уровня, связанных с базой данных.

Также в этом уровне находятся любые другие адаптеры, необходимые для преобразования данных из внешнего формата, например полученных от внешней службы, во внутренний, используемый вариантами использования и сущностями.

Фреймворки и драйверы

Самый внешний уровень модели на рис. 22.1 обычно состоит из фреймворков и инструментов, таких как база данных и веб-фреймворк. Как правило, для этого уровня требуется писать не очень много кода, и обычно этот код играет роль связующего звена со следующим внутренним кругом.

На уровне фреймворков и драйверов сосредоточены все детали. Веб-интерфейс — это деталь. База данных — это деталь. Все это мы храним во внешнем круге, где они не смогут причинить большого вреда.

Только четыре круга?

Круги на рис. 22.1 лишь схематически изображают основную идею: иногда вам может понадобиться больше четырех кругов. Фактически нет никакого правила, утверждающего, что кругов должно быть именно четыре. Но всегда действует правило зависимостей. Зависимости в исходном коде всегда должны быть направлены внутрь. По мере движения внутрь уровень абстракции и политик увеличивается. Самый внешний круг

включает низкоуровневые конкретные детали. По мере приближения к центру программное обеспечение становится все более абстрактным и инкапсулирует все более высокоуровневые политики. Самый внутренний круг является самым обобщенным и находится на самом высоком уровне.

Пересечение границ

Справа внизу на рис. 22.1 приводится пример пересечения границ кругов. На нем изображены контроллеры и презентаторы, взаимодействующие с вариантами использования на следующем уровне. Обратите внимание на поток управления: он начинается в контроллере, проходит через вариант использования и завершается в презентаторе. Отметьте также, как направлены зависимости в исходном коде: каждая указывает внутрь, на варианты использования.

Обычно мы разрешаем это кажущееся противоречие с использованием принципа инверсии зависимостей (Dependency Inversion Principle). В таких языках, как Java, например, мы можем определять интерфейсы и использовать механизм наследования, чтобы направление зависимостей в исходном коде было противоположно направлению потока управления на границах справа.

Допустим, что вариант использования должен вызвать презентатора. Такой вызов нельзя выполнить непосредственно, потому что иначе нарушится правило зависимостей: никакие имена, объявленные во внешних кругах, не должны упоминаться во внутренних. Поэтому вариант использования должен вызвать интерфейс (на рис. 22.1 подписан как «Порт вывода варианта использования»), объявленный во внутреннем круге, а презентатор во внешнем круге должен реализовать его.

Тот же прием используется для всех пересечений границ в архитектуре. Мы используем преимущество динамического полиморфизма, чтобы обратить зависимости в исходном коде в направлении, противоположном потоку управления, и тем самым соблюсти правило зависимостей при любом направлении потока управления.

Какие данные пересекают границы

Обычно через границы данные передаются в виде простых структур. При желании можно использовать простейшие структуры или объекты передачи данных (Data Transfer Objects; DTO). Данные можно также передавать в вызовы функций через аргументы. Или упаковывать их в ассоциативные массивы или объекты. Важно, чтобы через границы передавались простые, изолированные структуры данных. Не нужно хитрить и передавать объекты сущностей или записи из базы данных. Структуры данных не должны нарушать правило зависимостей.

Например, многие фреймворки для работы с базами данных возвращают ответы на запросы в удобном формате. Их можно назвать «представлением записей». Такие представления записей не должны передаваться через границы внутрь. Это нарушает правило зависимостей, потому что заставляет внутренний круг знать что-то о внешнем круге.

Итак, при передаче через границу данные всегда должны принимать форму, наиболее удобную для внутреннего круга.

Типичный сценарий

Диаграмма на рис. 22.2 показывает типичный сценарий работы веб-системы на Java, использующей базу данных. Веб-сервер

принимает исходные данные от пользователя и передает их контроллеру Controller в левом верхнем

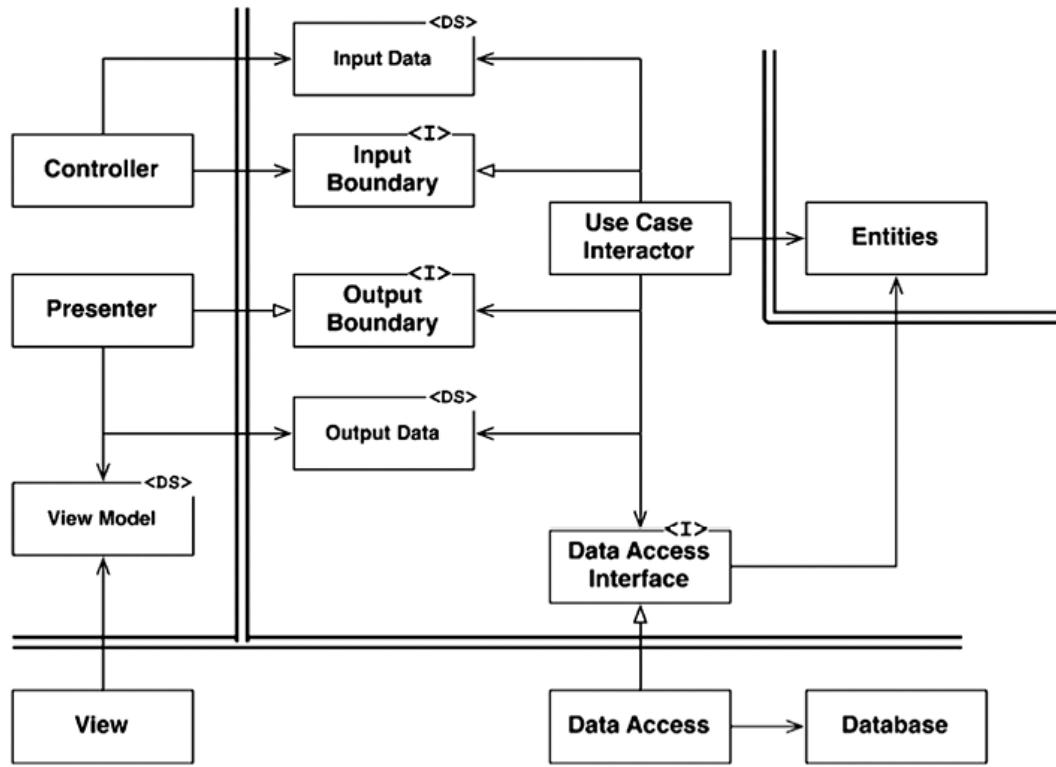


Рис. 22.2. Типичный сценарий работы веб-системы на Java, использующей базу данных

углу. Контроллер упаковывает данные в простой Java-объект `InputData` и передает его через интерфейс `InputBoundary` классу `UseCaseInteractor`. Класс `UseCaseInteractor` интерпретирует данные и использует их для управления действиями сущностей `Entities`. Он также переносит данные из базы данных `Database` в память сущностей `Entities` через интерфейс `DataAccessInterface`. По завершении `UseCaseInteractor` забирает данные из сущностей `Entities` и конструирует из них другой простой Java-объект `OutputData`. Затем объект `OutputData` передается через интерфейс `OutputBoundary` презентатору `Presenter`.

Презентатор Presenter переупаковывает данные из объекта OutputData в объект ViewModel, еще один простой Java-объект, содержащий в основном строки и флаги, используемые представлением View для отображения данных. Объект OutputData мог бы содержать объекты типа Date, но Presenter преобразует их в процессе формирования ViewModel в строковые значения, пригодные для отображения. То же произойдет с объектами типа Currency и любыми другими бизнес-данными. Имена кнопок Button и пунктов меню MenuItem помещаются в ViewModel вместе с флагами, сообщающими представлению View, следует ли отобразить эти кнопки и пункты меню в неактивном состоянии.

В результате представлению View практически ничего не приходится делать, кроме как перенести данные из ViewModel в HTML-страницу.

Обратите внимание на направления зависимостей. Все они пересекают линии границ только в направлении внутрь, следуя правилу зависимостей.

Заключение

Следование этим простым правилам не требует больших усилий и поможет сохранить душевный покой в будущем. Разделив программное обеспечение на уровни и соблюдая правило зависимостей, вы создадите систему, которую легко протестировать — со всеми вытекающими из этого преимуществами. Когда какой-либо из внешних элементов системы, например база данных или веб-фреймворк, устареет, вы сможете без всякой суэты заменить его.

[45](#) Data-Context-Interaction — данные, контекст, взаимодействие. — Примеч. пер.

[46](#) Boundary-Control-Entity – граница, управление, сущность. – *Примеч. пер.*

23. Презентаторы и скромные объекты



В главе 22 мы познакомились с понятием презентатора (presenter). Презентаторы являются разновидностью шаблона проектирования «Скромный объект» (Humble Object), помогающего выявлять и защищать архитектурные границы.

Фактически чистая архитектура в предыдущей главе полна реализациями шаблона «*Скромный объект*».

Шаблон «Скромный объект»

Шаблон проектирования «Скромный объект»⁴⁷ первоначально предлагался для использования в модульном тестировании как способ отделения поведения, легко поддающегося тестированию, от поведения, с трудом поддающегося тестированию. Идея очень проста: разделить поведение на два модуля или класса. Один из модулей, который называется «скромным», содержит все, что с трудом поддается тестированию, в виде, упрощенном до предела. Второй — все, что было выброшено из «скромного» модуля.

Например, графические пользовательские интерфейсы сложны для модульного тестирования, потому что трудно писать тесты, которые могут видеть изображение на экране и проверять присутствие соответствующих элементов. Однако в действительности большая часть поведения пользовательского интерфейса легко тестируется. Используя шаблон «Скромный объект», можно разделить два вида поведения на два разных класса, которые называют Презентатором (Presenter) и Представлением (View).

Презентаторы и представления

Представление (View) — это «скромный» объект, сложный для тестирования. Код в этом объекте упрощается до предела. Он просто переносит данные в графический интерфейс, никак не обрабатывая их.

Презентатор (Presenter) — это легко тестируемый объект. Его задача — получить данные от приложения и преобразовать

их так, чтобы Представление (View) могло просто переместить их на экран. Например, если приложению потребуется отобразить дату в некотором поле, оно должно передать Презентатору объект Date. Презентатор затем должен преобразовать дату в строку и поместить ее в простую структуру данных, которую называют Моделью представления (View Model), где Представление сможет найти ее.

Если приложению потребуется отобразить на экране денежную сумму, оно может передать Презентатору объект Currency. Презентатор должен преобразовать этот объект в строковое представление десятичного числа с соответствующим количеством десятичных знаков и знаком валюты и поместить полученную строку в Модель представления. Если отрицательные суммы должны отображаться красным цветом, тогда в Модели представления должен устанавливаться соответствующий флаг.

Каждая кнопка на экране имеет имя. Это имя является строкой в Модели представления, помещаемой туда Презентатором. Если кнопка должна отображаться как неактивная, Презентатор установит соответствующий логический флаг в Модели представления. Имя каждого пункта меню представлено строкой в Модели представления, помещаемой туда Презентатором. Имена всех радиокнопок, фляжков и текстовых полей и соответствующих логических флагов устанавливаются Презентатором в Модели представления. Таблицы чисел, которые должны отображаться на экране, преобразуются Презентатором в таблицы форматированных строк в Модели представления.

Все, что отображается на экране и чем так или иначе управляет приложение, представлено в Модели представления строкой, или логическим значением, или элементом перечисления. На долю Представления остается только

перенести данные из Модели представления на экран. То есть Представление играет скромную роль.

Тестирование и архитектура

Давно известно, что простота тестирования является характерным признаком хорошей архитектуры. Шаблон «Скромный объект» — хороший пример, потому что раздел между легко и тяжело тестируемыми частями часто совпадает с архитектурными границами. Раздел между Презентаторами и Представлениями — одна из таких границ, но существует много других.

Шлюзы к базам данных

Между средствами управления в вариантах использования и базами данных находятся шлюзы к базам данных⁴⁸. Эти шлюзы являются полиморфными интерфейсами, содержащими методы для каждой операции создания, чтения, изменения и удаления, которые приложению может выполнить в базе данных. Например, если приложению может понадобиться узнать фамилии всех пользователей, работавших вчера, в интерфейсе UserGateway должен иметься метод с именем `getLastNamesOfUsersWhoLoggedInAfter`, принимающий объект Date и возвращающий список фамилий.

Как вы помните, мы полностью отвергаем возможность появления кода на SQL на уровне вариантов использования; для этого используются интерфейсы шлюзов, имеющие определенные методы. Эти шлюзы реализуются классами на уровне базы данных. Такие реализации являются «скромными объектами». Они просто используют код SQL или любой другой интерфейс доступа к базе данных для извлечения данных,

необходимых каждому из методов. Механизмы управления в вариантах использования, напротив, не являются «скромными», потому что заключают бизнес-правила, характерные для приложения. Даже при том, что они не являются «скромными», средства управления легко поддаются тестированию, потому что шлюзы можно заменить соответствующими заглушками.

Преобразователи данных

Вернемся к теме баз данных. На каком уровне, по вашему мнению, должны находиться фреймворки ORM, такие как Hibernate?

Во-первых, давайте кое-что проясним: нет такой штуки, как инструмент объектно-реляционного преобразования (Object Relational Mapper; ORM), потому что объекты не являются структурами данных, по крайней мере с точки зрения пользователя. Пользователи объекта не видят данных, потому что все они хранятся в приватных полях. Пользователи видят только общедоступные методы объекта. То есть с точки зрения пользователя объект — это набор операций.

Структура данных, напротив, — набор общедоступных данных, не обладающих подразумеваемым поведением. Название «преобразователи данных» лучше подходит для инструментов ORM, потому что они загружают данные из реляционных таблиц в структуры.

Где должны находиться такие системы ORM? Конечно, на уровне базы данных. В действительности инструменты ORM представляют еще одну разновидность границы «Скромный объект» между интерфейсами шлюза и базой данных.

Службы

А что можно сказать о службах? Если приложение взаимодействует со службами или реализует свой набор служб, можно ли в этом случае применить шаблон «*Скромный объект*» для создания границы, отделяющей службу?

Конечно! Приложение должно загружать данные в простые структуры и передавать эти структуры через границу модулям, которые преобразуют данные и посылают их внешним службам. С другой стороны, в точке ввода слушатель службы должен принимать данные через интерфейс службы, преобразовывать их в простые структуры данных, пригодные для использования в приложении, и затем передавать их через границу службы.

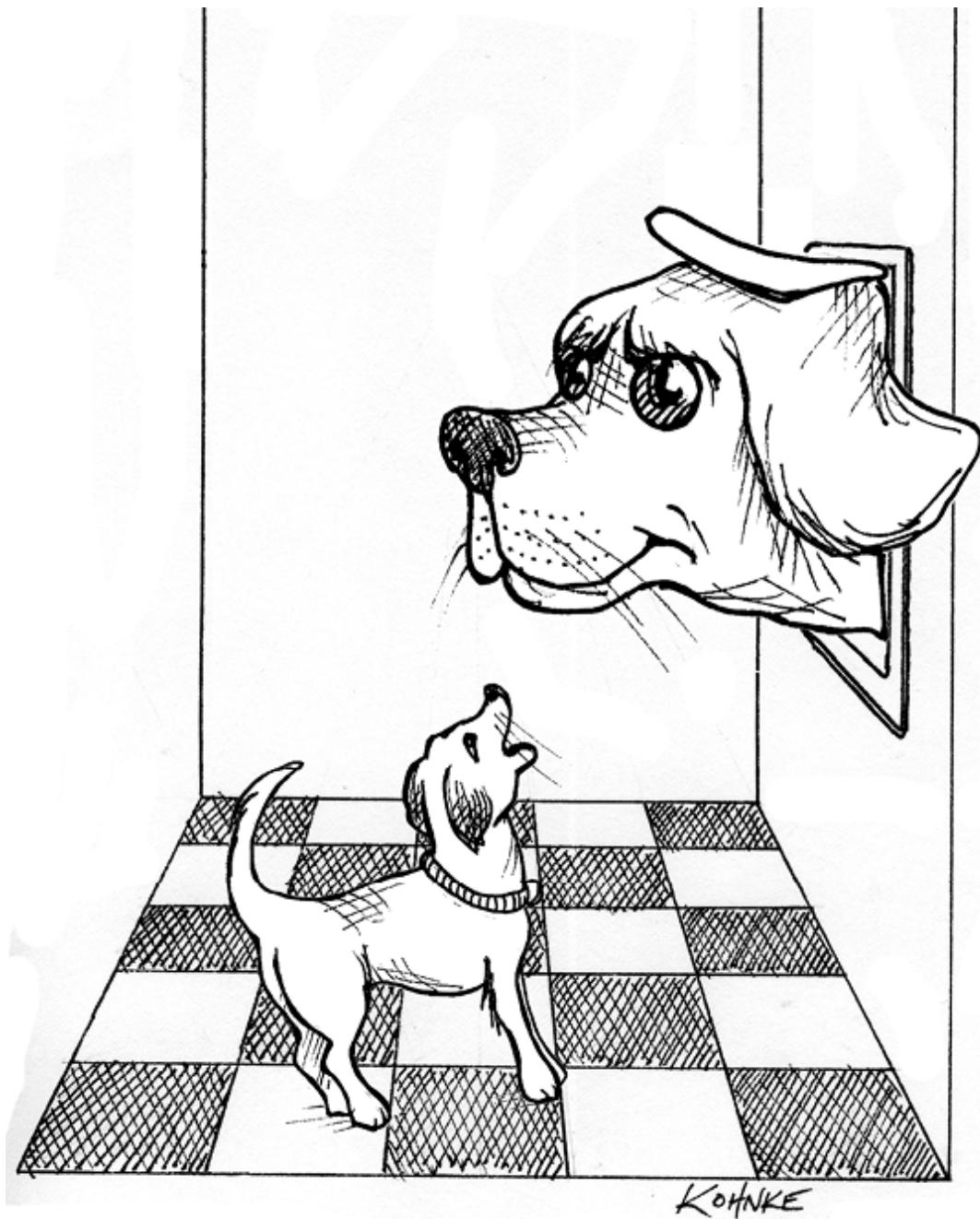
Заключение

Практически на каждой архитектурной границе можно найти возможность применить шаблон «*Скромный объект*». Взаимодействия через границу почти всегда осуществляются с применением некой простой структуры данных, и граница часто делит что-то на сложное и простое для тестирования. Применение этого шаблона для организации архитектурных границ значительно улучшает возможность тестирования системы в целом.

[47](#) *xUnit Patterns*, Meszaros, Addison-Wesley, 2007, p. 695. (Джерард Месарош. Шаблоны тестирования xUnit: рефакторинг кода тестов. М.: Вильямс, 2017. — Примеч. пер.).

[48](#) *Patterns of Enterprise Application Architecture*, Martin Fowler, et. al., Addison-Wesley, 2003, p. 466. (Мартин Фаулер. Шаблоны корпоративных приложений. М.: Вильямс, 2017. — Примеч. пер.).

24. Неполные границы



Полноценные архитектурные границы обходятся дорого. Они требуют определения двусторонних пограничных интерфейсов, структур для входных и выходных данных и

управления зависимостями для выделения двух сторон в компоненты, компилируемые и развертываемые независимо. Это требует значительных усилий для создания и сопровождения.

Во многих ситуациях хороший архитектор мог бы посчитать затраты на создание такой границы слишком высокими, но хотел бы сохранить место для такой границы на будущее.

Подобное упреждающее проектирование часто расценивается многими последователями гибкой разработки как нарушение принципа YAGNI: «You Aren't Going to Need It» («Вам это не понадобится»). Однако некоторые архитекторы смотрят на эту проблему и думают: «А мне может это понадобиться». В этом случае они могут реализовать неполную границу.

Пропустить последний шаг

Один из способов сконструировать неполную границу — проделать все, что необходимо для создания независимо компилируемых и развертываемых компонентов, и затем просто оставить их в одном компоненте. В этом компоненте будут присутствовать парные интерфейсы, структуры входных и выходных данных и все остальное, но все это будет компилироваться и развертываться как один компонент.

Очевидно, что для реализации такой неполной границы потребуется тот же объем кода и подготовительного проектирования, что и для полной границы. Но в этом случае не потребуется администрировать несколько компонентов. Не потребуется следить за номерами версий и нести дополнительное бремя управления версиями. Это отличие не нужно недооценивать.

Эта стратегия использовалась в начале развития FitNesse. Компонент веб-сервера FitNesse проектировался с возможностью отделения от компонентов вики и тестирования. Мы думали, что впоследствии у нас может появиться желание создать другие веб-приложения, использующие этот веб-компонент. В то же время мы не хотели вынуждать пользователей загружать два компонента. Как вы помните, одна из наших целей выражалась фразой «Загрузи и вперед». Мы специально хотели, чтобы пользователям нужно было загрузить и выполнить только один jar-файл, не заботясь о поиске других jar-файлов с совместимыми версиями и т.д.

История FitNesse также указывает на одну из опасностей такого подхода. Со временем, когда стало ясно, что отдельный веб-компонент никогда не понадобится, грань между веб-компонентом и компонентом вики стала размываться. Начали появляться зависимости, пересекающие линию в неправильном направлении. Разделить их сейчас было бы очень сложно.

Одномерные границы

Для оформления полноценной архитектурной границы требуется создать парные пограничные интерфейсы для управления изоляцией в обоих направлениях. Поддержание разделения в обоих направлениях обходится дорого не только на начальном этапе, но и на этапе сопровождения.

На рис. 24.1 показана более простая схема, помогающая зарезервировать место для последующего превращения в полноценную границу. Это пример традиционного шаблона «Стратегия». Клиенты пользуются интерфейсом ServiceBoundary, который реализуют классы ServiceImpl.

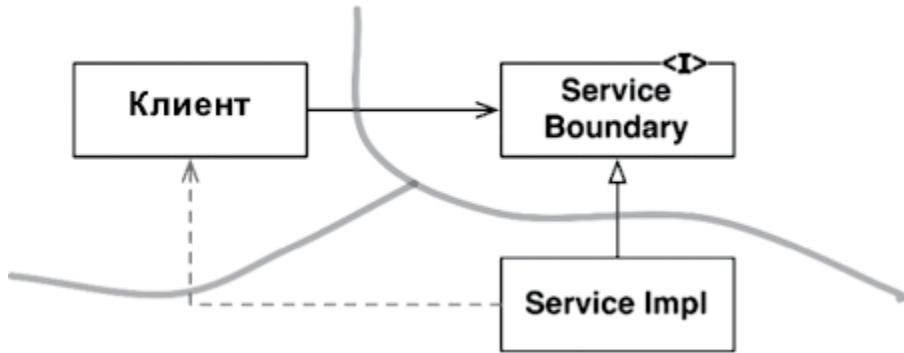


Рис. 24.1. Шаблон «Стратегия»

Должно быть ясно, что это создает основу для будущей архитектурной границы. Здесь имеет место инверсия зависимости, необходимая для отделения клиента от класса `ServiceImpl`. Также должно быть ясно, что разделение может очень быстро стираться, о чем свидетельствует пунктирная стрелка на диаграмме. В отсутствие парных интерфейсов ничто не мешает появлению таких обратных зависимостей, кроме старательности и дисциплинированности разработчиков и архитекторов.

Фасады

Еще более простой подход к организации границ дает шаблон «Фасад», изображенный на рис. 24.2. В этом случае отсутствует даже инверсия зависимостей. Граница определяется простым классом `Facade` с методами, представляющими службы и реализующими обращения к службам, к которым клиенты, как предполагается, не должны иметь прямого доступа.

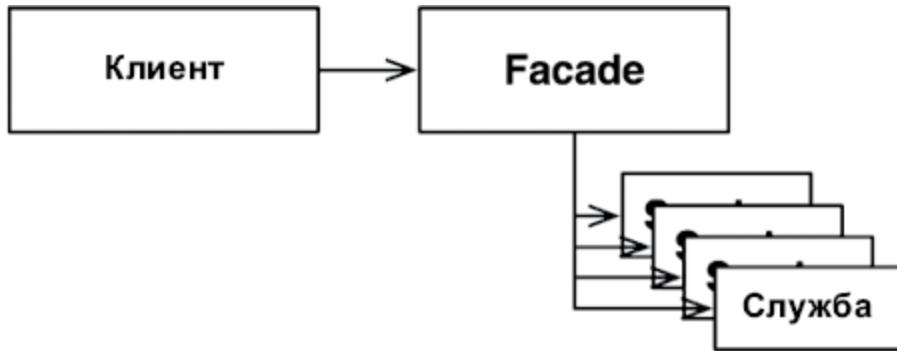


Рис. 24.2. Шаблон «Фасад»

Обратите внимание, однако, что клиент имеет транзитивную (переходную) зависимость от всех этих классов служб. В языках со статической системой типов изменение исходного кода в одном из классов служб вызывает необходимость повторной компиляции клиента. Также представьте, насколько просто в этой схеме создать обратные связи.

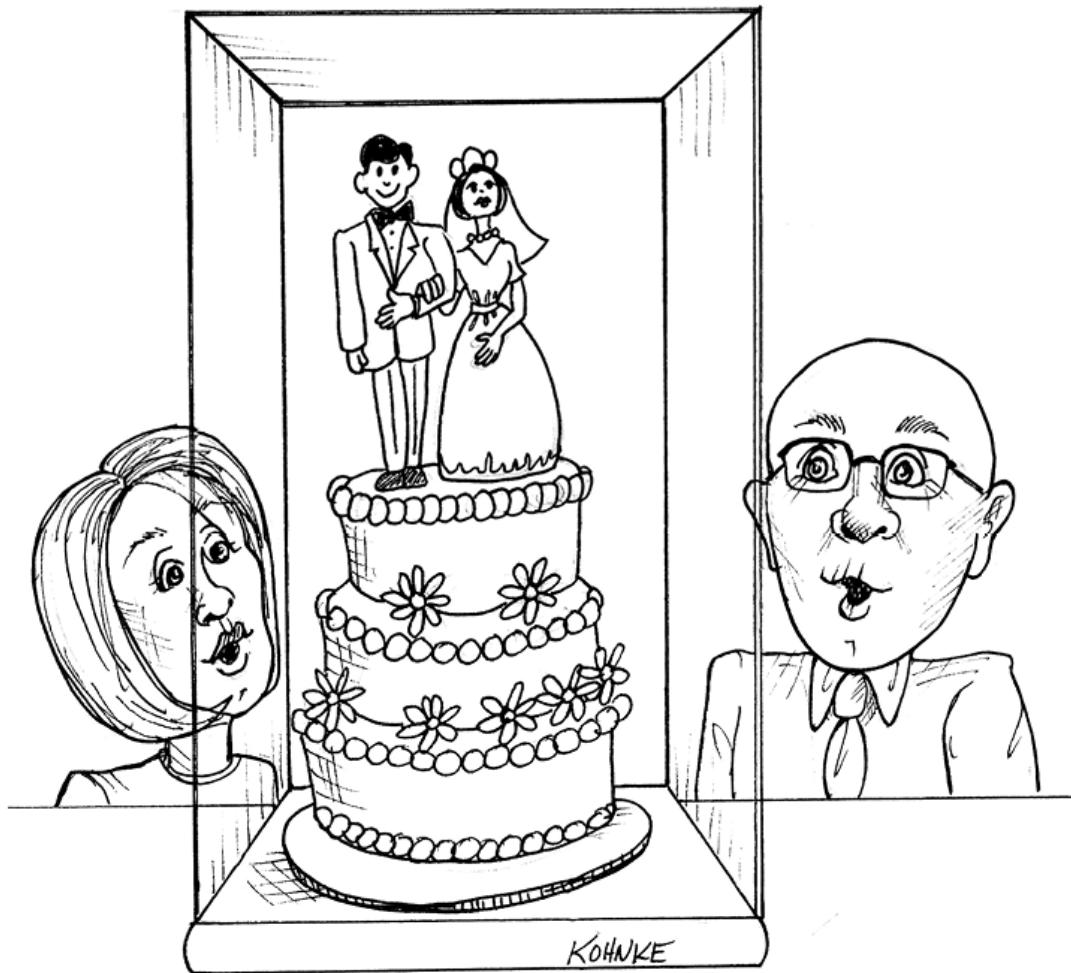
Заключение

Мы увидели три простых способа реализации неполных архитектурных границ. Конечно, таких способов намного больше. Эти три стратегии служат лишь примерами.

Каждый из представленных подходов имеет свои достоинства и недостатки. Каждый подходит на роль заменителя полноценной архитектурной границы в определенных контекстах. И каждый может со временем деградировать, если граница никогда не будет материализована.

Одна из задач архитектора — решить, где провести архитектурную границу и как ее реализовать, частично или полностью.

25. Уровни и границы



В любой системе легко выделить три компонента: пользовательский интерфейс, бизнес-правила и базу данных. Для простых систем этого более чем достаточно. Но для большинства систем число компонентов должно быть больше.

Рассмотрим, например, простую компьютерную игру. В ней легко выделить три компонента. Пользовательский интерфейс обрабатывает все сообщения от пользователя и передает их правилам игры. Правила сохраняют состояние игры в

некоторой хранимой структуре данных. Но действительно ли это все, что нужно?

Охота на Вампуса

Давайте немного конкретизируем. Возьмем в качестве примера почтенную приключенческую игру «Охота на Вампуса»⁴⁹, придуманную в 1972 году. В этой текстовой игре используются очень простые команды, такие как GO EAST (идти быстро) и SHOOT WEST (выстрелить в западном направлении). Игрок вводит команду, а компьютер в ответ сообщает, что персонаж видит, обоняет, слышит и чувствует. Игрок охотится за Вампусом в лабиринте пещер и должен избегать ловушек, ям и других опасностей. Желающие без труда найдут правила игры в Интернете.

Допустим, мы решили сохранить текстовый интерфейс, но отделить его от правил игры, чтобы наша версия могла обрабатывать команды на разных языках и распространяться в разных странах. Игровые правила взаимодействуют с компонентом пользовательского интерфейса посредством прикладного интерфейса, не зависящего от языка, а пользовательский интерфейс транслирует команды API на соответствующий естественный язык.

При правильной организации зависимостей, как показано на рис. 25.1, можно создать произвольное количество компонентов с пользовательским интерфейсом, использующих те же игровые правила. Правила игры ничего не знают об используемом естественном языке общения с пользователем.

Допустим также, что состояние игры сохраняется в некотором хранилище — это может быть флешка, облачное хранилище или просто ОЗУ. В любом из этих случаев игровые правила не должны знать деталей. Поэтому снова мы создаем

прикладной интерфейс, который игровые правила смогут использовать для взаимодействия с компонентом хранилища.

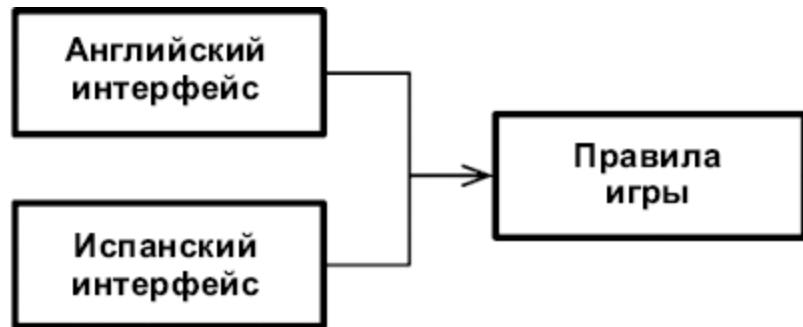


Рис. 25.1. Одни и те же игровые правила могут использоваться любым числом компонентов пользовательского интерфейса

Игровые правила не должны ничего знать о разных видах хранилищ, поэтому зависимости должны быть направлены в соответствии с правилом зависимостей, как показано на рис. 25.2.



Рис. 25.2. Следование правилу зависимостей

Чистая архитектура?

Очевидно, что в этом контексте мы легко смогли бы применить приемы создания чистой архитектуры⁵⁰ со всеми вариантами использования, границами, сущностями и соответствующими структурами данных. Но действительно ли мы нашли все важнейшие архитектурные границы?

Например, язык не является единственной осью изменения для пользовательского интерфейса. Также может измениться механизм ввода текста. Например, мы можем использовать обычное окно командной оболочки, текстовые сообщения или приложение чата. Возможности в этом плане бесчисленны.

Это означает, что существует потенциальная архитектурная граница, определяемая этой осью изменения. Возможно, нам следует сконструировать API, пересекающий границу и отделяющий язык от механизма ввода; эта идея показана на рис. 25.3.

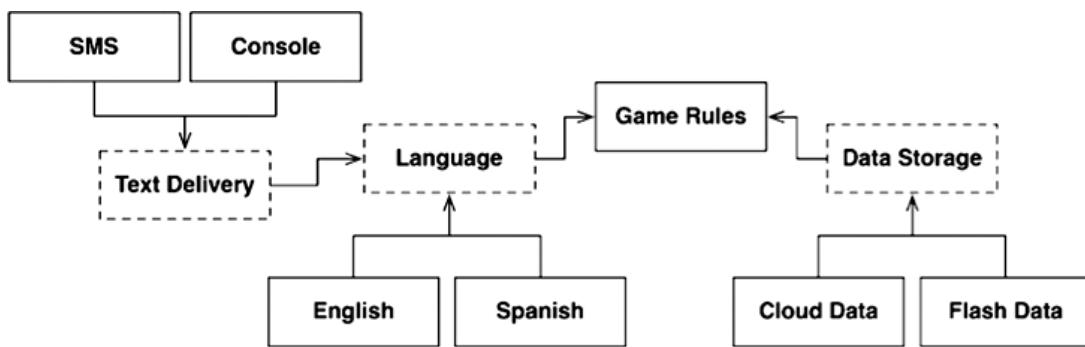


Рис. 25.3. Следующая версия диаграммы

Диаграмма на рис. 25.3 стала сложнее, но не содержит никаких сюрпризов. Пунктиром обведены абстрактные компоненты, определяющие API, реализуемый компонентами, стоящими выше или ниже их. Например, Language API реализуют компоненты English и Spanish.

GameRules взаимодействует с компонентом Language через API, который определяет GameRules и реализует Language. Также компонент Language взаимодействует с компонентом TextDelivery посредством API, который определяется в Language и реализуется в TextDelivery. Как видите, API определяется и принадлежит компоненту-пользователю, но не реализующему его.

Если заглянуть в GameRules, можно увидеть полиморфные пограничные интерфейсы, используемые внутри GameRules и реализованные в компоненте Language. Имеются также полиморфные пограничные интерфейсы, используемые компонентом Language и реализованные в GameRules.

Заглянув в Language, мы увидели бы то же самое: полиморфные пограничные интерфейсы, используемые кодом в Language и реализованные в TextDelivery, и полиморфные пограничные интерфейсы, используемые кодом в TextDelivery и реализованные в Language.

В каждом случае API определяется пограничными интерфейсами, принадлежащими компоненту, находящемуся уровнем выше.

Варианты, такие как English, SMS и CloudData, предоставляются полиморфными интерфейсами, определяемыми в API абстрактных компонентов и реализуемыми конкретными компонентами, которые обслуживают их. Например, предполагается, что полиморфные интерфейсы, объявленные в Language, будут реализованы в English и Spanish.

Эту диаграмму можно упростить, устранив все варианты и сосредоточившись исключительно на API компонентов, как показано на рис. 25.4.

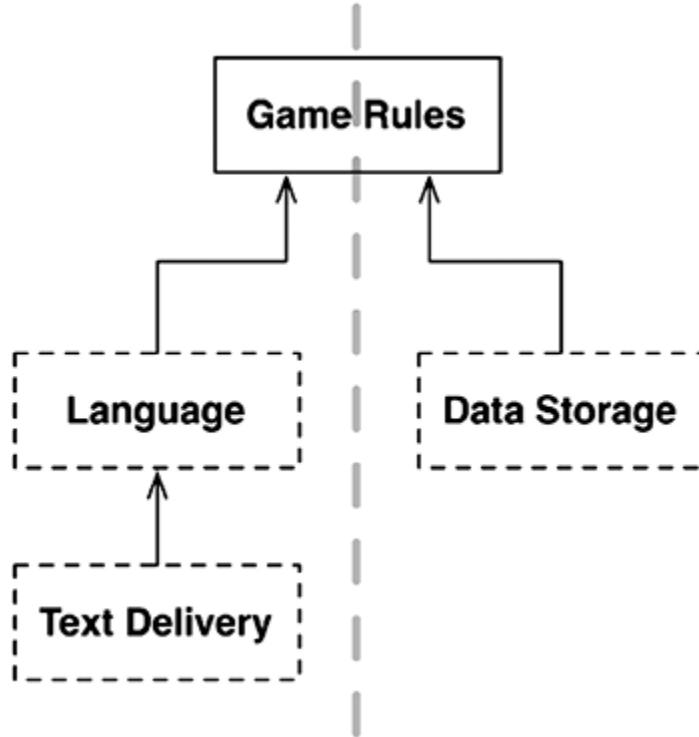


Рис. 25.4. Упрощенная диаграмма

Обратите внимание, что диаграмма на рис. 25.4 ориентирована так, чтобы все стрелки указывали вверх. В результате компонент GameRules оказался вверху. Такая ориентация имеет определенный смысл, потому что GameRules содержит политики высшего уровня.

Рассмотрим направление движения информации. Ввод от пользователя передается через компонент TextDelivery снизу слева. Она поднимается вверх до компонента Language, где транслируется в команды, понятные GameRules. GameRules обрабатывает ввод пользователя и посыпает соответствующие данные вниз, в компонент DataStorage справа внизу.

Затем GameRules посыпает ответ обратно в компонент Language, который переводит его на соответствующий язык и возвращает пользователю через компонент TextDelivery.

Такая организация эффективно делит поток данных на два потока^{[51](#)}. Поток слева соответствует взаимодействию с пользователем, а поток справа — с хранилищем данных. Оба потока встречаются на вершине^{[52](#)}, в компоненте GameRules — конечном обработчике данных, через который проходят оба потока.

Пересечение потоков

Всегда ли существует только два потока данных, как в данном примере? Нет, не всегда. Представьте, что мы захотели реализовать сетевой вариант игры «Охота на Вампуса», в которой участвует несколько игроков. В этом случае нам потребуется сетевой компонент, как показано на рис. 25.5. В данном случае поток данных делится на три потока, управляемых компонентом GameRules.

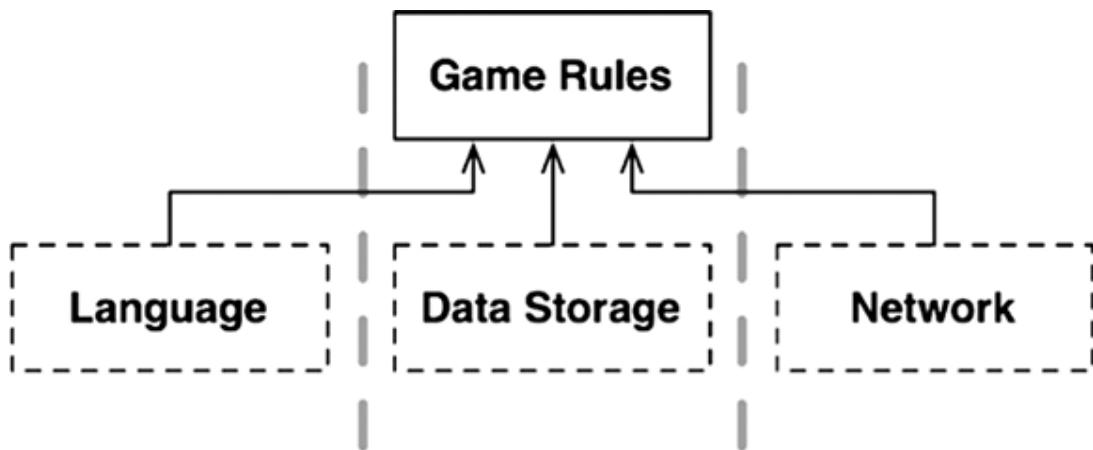


Рис. 25.5. Добавление сетевого компонента

То есть с ростом сложности системы структура компонентов может разбиваться на несколько потоков.

Разбиение потоков

Сейчас вы наверняка подумали, что все потоки в конечном счете встречаются на вершине диаграммы, в единственном компоненте. Ах, если бы все было так просто! Увы, действительность намного сложнее.

Рассмотрим компонент `GameRules` для игры «Охота на Вампуса». Часть игровых правил связана с механикой карты. Они знают, как соединены пещеры и какие объекты находятся в каждой пещере. Они знают, как переместить игрока из пещеры в пещеру и как генерировать события для игрока.

Но есть еще ряд политик на еще более высоком уровне — политики, которые управляют здоровьем персонажа и действием определенных событий. Эти политики могут вызывать ухудшение здоровья у персонажа или улучшать его, давая персонажу еду и питье. Низкоуровневые политики, отвечающие за механику перемещений, могут определять события для этой высокоуровневой политики, такие как `FoundFood` или `FellInPit`. А высокоуровневая политика могла бы управлять состоянием персонажа (как показано на рис. 25.6). В конечном итоге эта политика могла бы определять окончательный итог — победу или проигрыш в игре.

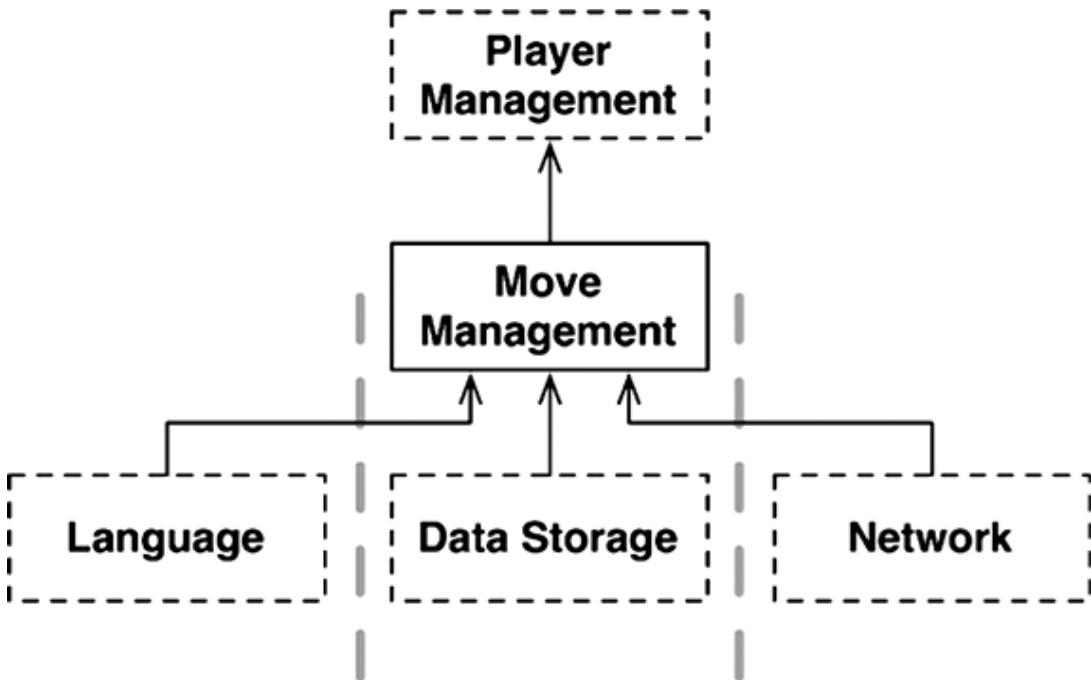


Рис. 25.6. Высокоуровневая политика управляет состоянием персонажа

Является ли это архитектурной границей? Нужно ли нам определить API, отделяющий MoveManagement от PlayerManagement? А давайте сделаем ситуацию еще интереснее и добавим микрослужбы.

Допустим, что мы получили массивную многопользовательскую версию игры «Охота на Вампира». Компонент MoveManagement действует локально, на компьютере игрока, а PlayerManagement действует на сервере. PlayerManagement предлагает API микрослужбы для всех подключенных компонентов MoveManagement.

Диаграмма на рис. 25.7 представляет несколько упрощенное отражение этого сценария. Элементы Network в действительности немного сложнее, чем показано на диаграмме, но сама идея должна быть понятна. В данном случае между MoveManagement и PlayerManagement пролегает полноценная архитектурная граница.

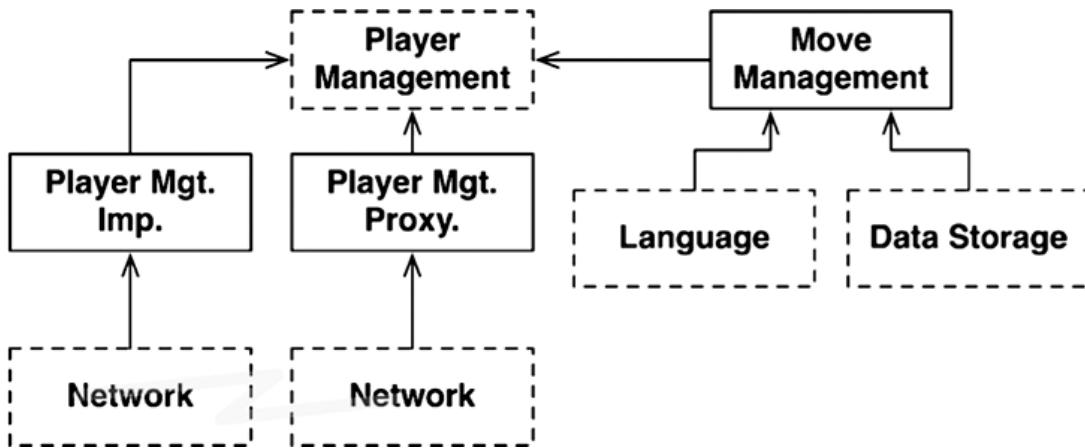


Рис. 25.7. Добавление API микрослужб

Заключение

Что из всего этого следует? Почему я взял эту до абсурда простую программу, которую можно уместить в 200 строк кода на языке оболочки Kornshell, и развел ее до огромных размеров со всеми этими сумасшедшими архитектурными границами?

Этот пример призван был показать, что архитектурные границы существуют повсюду. Мы как архитекторы должны проявлять осторожность и проводить их, только когда они действительно нужны. Мы также должны помнить, что полная реализация границ обходится дорого.

В то же время мы должны помнить, что игнорирование границ может вызвать сложности в будущем — даже при наличии всеобъемлющего набора тестов и жесткой дисциплины рефакторинга.

Итак, что мы должны делать как архитекторы? Ответ едва ли удовлетворит вас. С одной стороны, некоторые очень умные люди много лет говорили нам, что мы не должны испытывать потребности в абстракциях. Это философия YAGNI: «You Aren't Going to Need It» («Вам это не понадобится»). В этом есть

определенная мудрость, поскольку излишнее усложнение конструкции часто намного хуже ее упрощения. С другой стороны, когда обнаруживается, что в том или ином месте действительно необходимо провести архитектурную границу, стоимость и риск ее добавления могут оказаться очень высокими.

Вот так-то, Архитектор Программного Обеспечения, вы должны предвидеть будущее. Вы должны предугадывать с пониманием дела. Вы должны взвесить все за и против, определить, где пролегают архитектурные границы и какие из них должны быть реализованы полностью, какие частично, а какие можно вообще игнорировать.

Но это не единовременное решение. Невозможно раз и навсегда решить на ранних этапах проектирования, какие границы реализовать, а какие игнорировать. Вы должны наблюдать за развитием системы, отмечать места, где может потребоваться провести новую границу, и затем внимательно следить за появлением первых трений, возникающих из-за отсутствия границ.

В этот момент нужно взвесить затраты на реализацию границ и цену их игнорирования и принять решение. Ваша цель — создать границу прямо в точке перегиба, когда реализовать ее окажется дешевле, чем продолжать игнорировать.

Для этого вы должны наблюдать очень внимательно.

[49](https://ru.wikipedia.org/wiki/Hunt_the_Wumpus) https://ru.wikipedia.org/wiki/Hunt_the_Wumpus. — Примеч. пер.

[50](#) Так же должно быть ясно, что мы не будем применять приемы создания чистой архитектуры для таких тривиальных программ, как эта игра. В конце концов, программу целиком можно уместить в 200 строк кода и даже меньше. В этом случае мы используем простую программу как прокси для более крупной системы со значимыми архитектурными границами.

[51](#) Если вас взволновало несоответствие направлений стрелок на рис. 25.4, напомню, что они соответствуют зависимостям в исходном коде, но не движению

потоков данных.

[52](#) В далеком прошлом мы назвали бы компонент на вершине центральным преобразователем (Central Transform). Подробности см. в книге Meilir Page-Jones, *Practical Guide to Structured Systems Design*», 2nd ed., 1988.

26. Главный компонент



В каждой системе имеется хотя бы один компонент, который создает другие компоненты, наблюдает за ними и

координирует их действия. Я называю такой компонент главным (`Main`).

Конечная деталь

Компонент `Main` — это конечная деталь, политика самого низкого уровня. Он является точкой входа в систему. От него ничего не зависит, кроме работоспособности системы. Его задача — создать все Фабрики, Стратегии и другие глобальные средства и затем передать управление высокоуровневым абстракциям в системе.

Именно в компоненте `Main` должны внедряться все зависимости с использованием инфраструктуры внедрения зависимостей. После этого компонент `Main` должен распространить эти зависимости, обычно без использования инфраструктуры.

Компонент `Main` можно считать самым грязным из всех грязных компонентов.

Рассмотрим следующий компонент `Main` из последней версии игры «Охота на Вампуса». Обратите внимание, как он загружает все строки, о которых не должен знать основной код.

```
public class Main implements HtwMessageReceiver
{
    private static HuntTheWumpus game;
    private static int hitPoints = 10;
    private static final List<String> caverns =
        new ArrayList<>();
    private static final String[] environments = new String[]{
        "bright",
        "humid",
```

```
"dry",
"creepy",
"ugly",
"foggy",
"hot",
"cold",
"drafty",
"dreadful"
};
```

```
private static final String[] shapes = new String[] {
"round",
"square",
"oval",
"irregular",
"long",
"craggy",
"rough",
"tall",
"narrow"
};
```

```
private static final String[] cavernTypes = new String[] {
"cavern",
"room",
"chamber",
"catacomb",
"crevasse",
```

```
"cell",
"tunnel",
"passageway",
"hall",
"expanse"
};
```

```
private static final String[] adornments = new String[] {
    "smelling of sulfur",
    "with engravings on the walls",
    "with a bumpy floor",
    "",
    "littered with garbage",
    "spattered with guano",
    "with piles of Wumpus droppings",
    "with bones scattered around",
    "with a corpse on the floor",
    "that seems to vibrate",
    "that feels stuffy",
    "that fills you with dread"
};
```

Далее следует функция `main`. Обратите внимание, как она использует `HtwFactory` для создания игры. Она передает имя класса, `htw.game.HuntTheWumpusFacade`, потому что этот класс даже грязнее, чем `Main`. Это предотвращает изменения в данном классе из-за повторной компиляции/развертывания `Main`.

```
public static void main(String[] args) throws
IOException {
    game = =
HtwFactory.makeGame("htw.game.HuntTheWumpusFacade",
    new Main());
createMap();
BufferedReader br =
    new BufferedReader(new InputStreamReader(System.in));
game.makeRestCommand().execute();
while (true) {
    System.out.println(game.getPlayerCavern());
    System.out.println("Health: " + hitPoints + " arrows: " +
        game.getQuiver());
    HuntTheWumpus.Command c = game.makeRestCommand();
    System.out.println(">");
    String command = br.readLine();
    if (command.equalsIgnoreCase("e"))
        c = game.makeMoveCommand(EAST);
    else if (command.equalsIgnoreCase("w"))
        c = game.makeMoveCommand(WEST);
    else if (command.equalsIgnoreCase("n"))
        c = game.makeMoveCommand(NORTH);
    else if (command.equalsIgnoreCase("s"))
        c = game.makeMoveCommand(SOUTH);
    else if (command.equalsIgnoreCase("r"))
        c = game.makeRestCommand();
    else if (command.equalsIgnoreCase("sw"))
        c = game.makeShootCommand(WEST);
    else if (command.equalsIgnoreCase("se"))
```

```

        c = game.makeShootCommand(EAST);
    else if (command.equalsIgnoreCase("sn"))
        c = game.makeShootCommand(NORTH);
    else if (command.equalsIgnoreCase("ss"))
        c = game.makeShootCommand(SOUTH);
    else if (command.equalsIgnoreCase("q"))
        return;

    c.execute();
}
}

```

Отметьте также, что функция `main` создает поток ввода и содержит главный цикл игры, в котором происходит интерпретация простых команд, но их обработка поручается другим, высокоуровневым компонентам.

Наконец, посмотрите, как `main` создает карту подземелий.

```

private static void createMap() {
    int nCaverns = (int) (Math.random() * 30.0 + 10.0);
    while (nCaverns-- > 0)
        caverns.add(makeName());
    for (String cavern : caverns) {
        maybeConnectCavern(cavern, NORTH);
        maybeConnectCavern(cavern, SOUTH);
        maybeConnectCavern(cavern, EAST);
        maybeConnectCavern(cavern, WEST);
    }
}

```

```
String playerCavern = anyCavern();
```

```
    game.setPlayerCavern(playerCavern);
    game.setWumpusCavern(anyOther(playerCavern));
    game.addBatCavern(anyOther(playerCavern));
    game.addBatCavern(anyOther(playerCavern));
    game.addBatCavern(anyOther(playerCavern));

    game.addPitCavern(anyOther(playerCavern));
    game.addPitCavern(anyOther(playerCavern));
    game.addPitCavern(anyOther(playerCavern));

    game.setQuiver(5);
}

// здесь следует еще много кода...
}
```

Компонент `Main` — самый грязный низкоуровневый модуль, находящийся во внешнем круге чистой архитектуры. Он загружает все, что потребуется системе более высокого уровня, а затем передает ей управление.

Заключение

Представьте, что `Main` является плагином для приложения — плагином, который настраивает начальное окружение, загружает все внешние ресурсы и затем передает управление политике верхнего уровня. Так как это плагин, может иметься множество компонентов `Main`, по одному для каждой конфигурации приложения.

Например, у нас может иметься плагин `Main` для *разработки*, еще один для *тестирования* и третий для *эксплуатации*. Можно также создать по плагину `Main` для каждой страны, или каждой юрисдикции, или каждого клиента.

Когда вы начинаете думать о компоненте `Main` как о плагине, находящемся за архитектурной границей, проблемы настройки решаются намного проще.

27. Службы: большие и малые



Сервис-ориентированные «архитектуры» и микросервисные «архитектуры» приобрели особую популярность в последнее время. Эта популярность обусловлена, в том числе, следующими причинами:

- Службы выглядят полностью независимыми друг от друга. Но, как мы увидим далее, это верно лишь отчасти.
- Службы, похоже, можно разрабатывать и развертывать независимо. И снова, как мы увидим далее, это верно лишь отчасти.

Сервисная архитектура?

Прежде всего уточним, что утверждение, будто использование служб по своей природе является архитектурой, в принципе неверно. Архитектура системы определяется границами, отделяющими высоконивневые политики от низконивневых

деталей, и следованием правилу зависимостей. Службы, просто делящие приложение на функциональные элементы, по сути, являются лишь функциями, вызовы которых обходятся довольно дорого и не обязательно имеют важное архитектурное значение.

То есть не все службы *должны* быть архитектурно значимыми. Часто бывает выгодно создавать службы, распределяющие функциональные возможности по разным процессам и платформам, независимо от того, подчиняются ли они правилу зависимостей. Службы сами по себе не определяют архитектуру.

Наглядной аналогией является организация функций. Архитектура монолитной или компонентной системы определяется некоторыми вызовами функций, пересекающими архитектурные границы и следующими правилу зависимостей. Однако многие другие функции в системе просто отделяют одно поведение от другого и не являются архитектурно значимыми.

То же верно в отношении служб. Службы, в конечном счете, — это всего лишь вызовы функций через границы процессов и/или платформ. Некоторые из этих служб действительно являются архитектурно значимыми, а другие — нет. Основной интерес для нас в этой главе представляют первые.

Преимущества служб?

Знак вопроса в заголовке указывает, что в этом разделе мы поставим под сомнение популярное мнение о сервис-ориентированной архитектуре. Давайте рассмотрим предполагаемые преимущества по одному.

Заблуждение о независимости

Одно из самых больших предполагаемых преимуществ деления системы на службы заключается в полной независимости их друг от друга. В конце концов, каждая служба выполняется в отдельном процессе или даже на другой машине; поэтому службы не имеют доступа к переменным друг друга. Более того, для каждой службы должен быть четко определен ее интерфейс.

Все это верно до определенной степени, но не до конца. Да, службы независимы на уровне отдельных переменных. Но они все еще могут быть связаны вместе общими ресурсами на одной машине или в сети. Более того, они тесно связаны общими данными.

Например, чтобы добавить новое поле в запись, которая передается между службами, придется изменить все службы, использующие это новое поле. Службы должны также согласовать интерпретацию данных в этом поле. То есть службы тесно связаны с записью и, соответственно, косвенно связаны друг с другом.

Утверждение о необходимости четко определять интерфейс тоже верно, но оно в той же мере относится к функциям. Интерфейсы служб не являются более формальными, строгими и четкими, чем интерфейсы функций. Как видите, это преимущество довольно иллюзорно.

Заблуждение о возможности независимой разработки и развертывания

Другое предполагаемое преимущество — возможность передачи служб во владение отдельным командам. Каждая такая команда может отвечать за разработку, сопровождение и эксплуатацию службы, как это предполагает стратегия

интеграции разработки и эксплуатации (DevOps). Такая независимость разработки и развертывания, как предполагается, является масштабируемой. Считается, что крупные корпоративные системы могут состоять из десятков, сотен и даже тысяч служб, разрабатываемых и развертываемых независимо. Разработку, сопровождение и эксплуатацию системы можно разделить между аналогичным количеством независимых команд.

Это утверждение верно, но лишь отчасти. Во-первых, как показывает история, крупные корпоративные системы могут состоять из монолитных и компонентных систем, а также из систем, состоящих из служб. То есть службы не единственный способ построения масштабируемых систем.

Во-вторых, как было показано в разделе «Заблуждение о независимости», службы не всегда могут разрабатываться, развертываться и эксплуатироваться независимо. Их разработку приходится координировать в той же мере, в какой они связаны данными или поведением.

Проблема с животными

Чтобы рассмотреть эти два заблуждения на примере, вернемся снова к нашей системе агрегатора такси. Как вы помните, эта система знает о нескольких компаниях, предоставляющих услуги такси в данном городе, и позволяет клиентам заказывать поездки. Представим, что клиенты выбирают такси по ряду критериев, таких как время ожидания, стоимость, комфорт и опытность водителя.

Мы решили для большей масштабируемости реализовать ее в виде множества маленьких микрослужб. Мы разделили большой коллектив разработчиков на множество маленьких

команд и каждой поручили разработку, сопровождение и эксплуатацию соответствующего⁵³ количества микрослужб.

Схема на рис. 27.1 демонстрирует, как наши предполагаемые архитекторы распределили ответственность приложения между микрослужбами. Служба TaxiUI взаимодействует непосредственно с клиентом, заказывающим такси с помощью своего мобильного устройства. Служба TaxiFinder исследует параметры разных поставщиков услуг TaxiSupplier и выбирает возможных

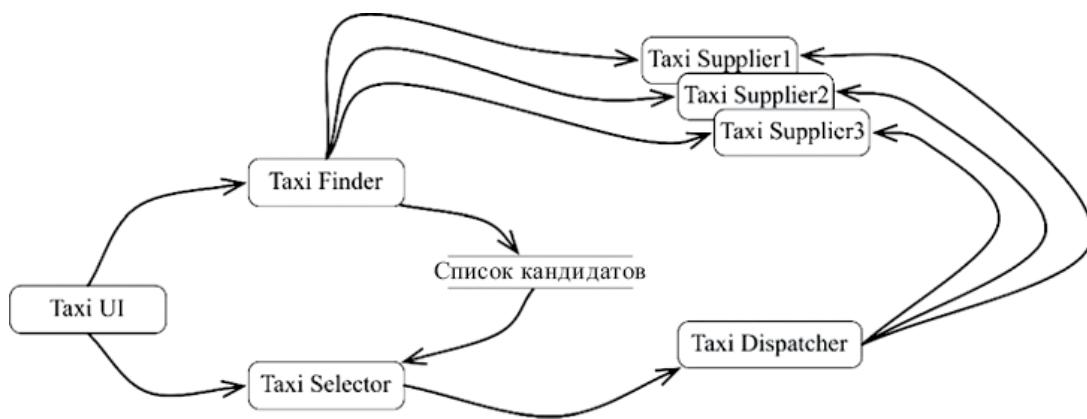


Рис. 27.1. Организация системы-агрегатора услуг такси в виде комплекса служб

кандидатов для обслуживания клиента. Она собирает их во временной записи, прикрепленной к данному пользователю. Служба **TaxiSelector** получает критерии, обозначенные пользователем, касающиеся стоимости, времени, комфорта и т.д., и выбирает наиболее подходящий вариант из списка кандидатов. Затем она передает выбранный вариант службе **TaxiDispatcher**, которая оформляет заказ.

Теперь представим, что эта система работает уже больше года. Наши разработчики успешно добавляют новые возможности, сопровождают и эксплуатируют все эти службы.

В один прекрасный солнечный день сотрудники отдела маркетинга организовали встречу с разработчиками. На этой

встрече они объявили о своем решении организовать услугу по доставке котят. Как предполагается, клиенты смогут заказать доставку котят себе домой или на работу.

Компания создаст несколько пунктов сбора котят по всему городу, и когда поступит заказ, ближайшее такси заберет котенка в одном из пунктов сбора и доставит по указанному адресу.

Один из поставщиков услуг согласился участвовать в этой программе. Другие, скорее всего, последуют его примеру, а трети могут отклонить предложение.

У некоторых водителей, как вы понимаете, может быть аллергия на кошек, поэтому их нельзя выбирать для выполнения таких заказов. Кроме того, у некоторых клиентов тоже может быть аллергия на кошек, поэтому для обслуживания тех, кто заявил об аллергии на кошек, не должны выбираться машины, осуществлявшие доставку котят в последние три дня.

А теперь взгляните на диаграмму служб и скажите, сколько из них придется изменить, чтобы реализовать описанную услугу? *Все!* Совершенно понятно, что разработка и развертывание услуги доставки котят должны быть тщательно скоординированы.

Иными словами, все службы тесно связаны между собой и не могут разрабатываться, развертываться и сопровождаться независимо.

Эта проблема свойственна всем сквозным задачам. С этой проблемой приходится сталкиваться любой системе, независимо от того, организована она в виде комплекса служб или нет. Системы, разделенные на службы по функциональному признаку, как показано на рис. 27.1, очень уязвимы для новых особенностей, пересекающих все эти функциональные уровни.

Спасение в объектах

Как бы мы решили эту проблему в архитектуре, состоящей из компонентов? Неуклонное следование принципам проектирования SOLID вынудило бы нас создать набор классов, которые можно было бы полиморфно расширять под потребности новых возможностей.

Схема на рис. 27.2 демонстрирует эту стратегию. Классы на этой схеме примерно соответствуют службам на рис. 27.1. Но обратите внимание на границы. Отметьте также, что все зависимости оформлены в соответствии с правилом зависимостей.

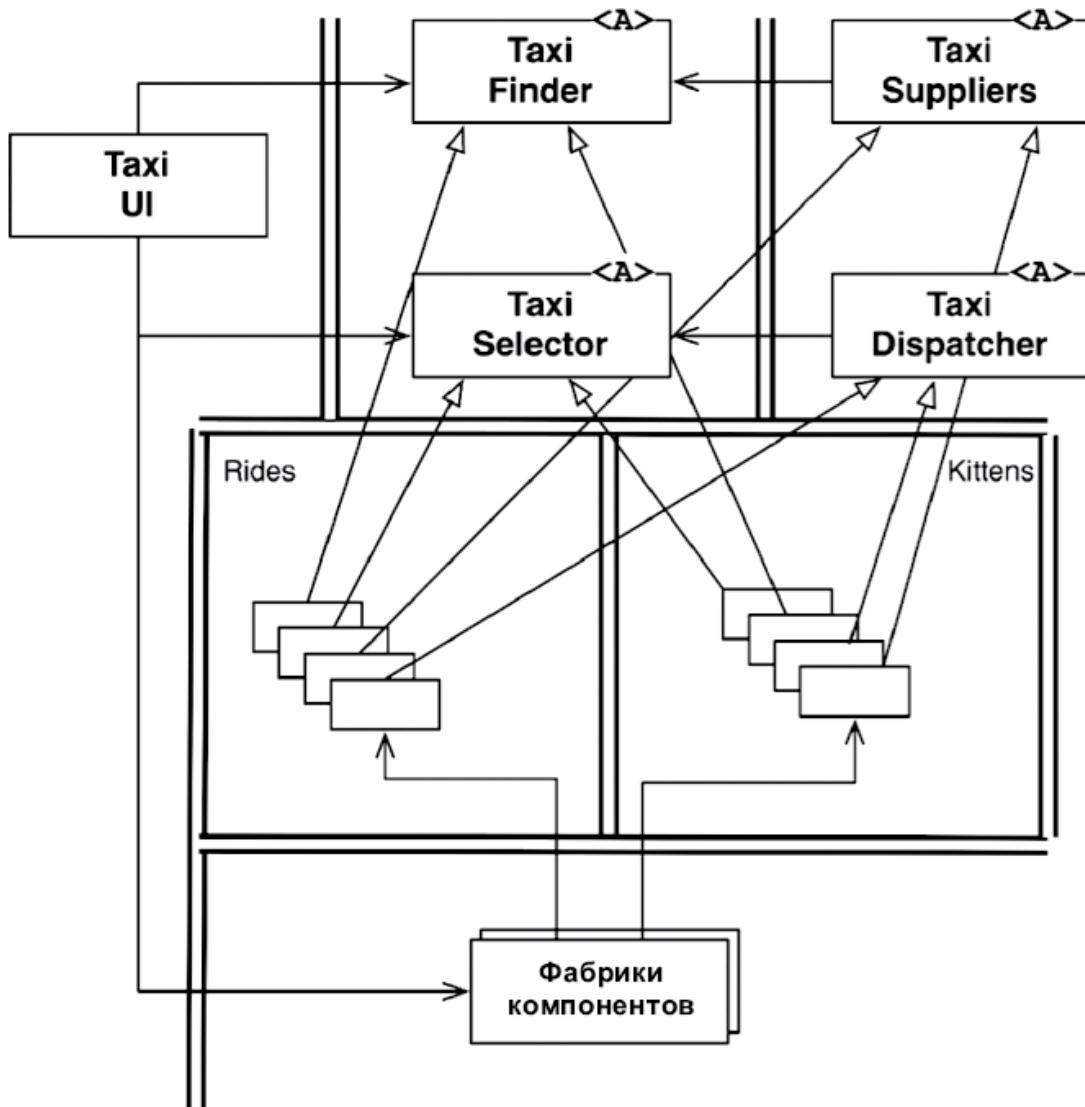


Рис. 27.2. Использование объектно-ориентированного подхода для преодоления проблемы сквозных задач

Большая часть логики оригинальных служб сосредоточена в базовых классах объектной модели. Однако часть логики, связанная с поездками, выделена в компонент **Rides**. Реализация новой услуги по доставке котят помещена в отдельный компонент **Kittens**. Эти два компонента переопределяют абстрактные базовые классы из оригинальных компонентов с применением шаблонов, таких как «Шаблонный метод» или «Стратегия».

Отметим еще раз, что два новых компонента, `Rides` и `Kittens`, подчиняются правилу зависимостей. Обратите также внимание, что классы, реализующие эти возможности, создаются фабриками под управлением пользовательского интерфейса.

Ясно, что в этой схеме после реализации `Kittens` придется изменить `TaxiUI`. Но все остальное останется в прежнем виде. В систему добавится лишь новый файл `jar`, `Gem` или `DLL`, который будет динамически загружаться во время выполнения системы.

То есть компонент `Kittens` существует отдельно от других и может разрабатываться и развертываться независимо.

Службы на основе компонентов

Возникает резонный вопрос: можно ли реализовать нечто подобное для служб? И ответ, конечно, да! Службы не обязательно должны быть маленькими монолитами. Службы также могут проектироваться в соответствии с принципами SOLID и данной структурой компонентов, чтобы иметь возможность добавлять в них новые компоненты, не изменяя существующие.

Представьте службу на Java как набор абстрактных классов в одном или нескольких `jar`-файлах. Каждую новую возможность или расширение существующей возможности можно реализовать как отдельный `jar`-файл, содержащий классы, наследующие абстрактные классы из уже имеющихся `jar`-файлов. Для развертывания новой возможности в этом случае больше не потребуется повторно развертывать службы, достаточно лишь добавить новые `jar`-файлы в пути загрузки этих служб. Иными словами, добавление новой возможности

выполняется в соответствии с принципом открытости/закрытости (Open-Closed Principle; OCP).

Схема на рис. 27.3 демонстрирует такую организацию служб. На ней представлены все те же службы, только на этот раз каждая состоит из компонентов, что позволяет добавлять новые возможности, реализованные в виде новых производных классов. Эти производные классы находятся в собственных компонентах.

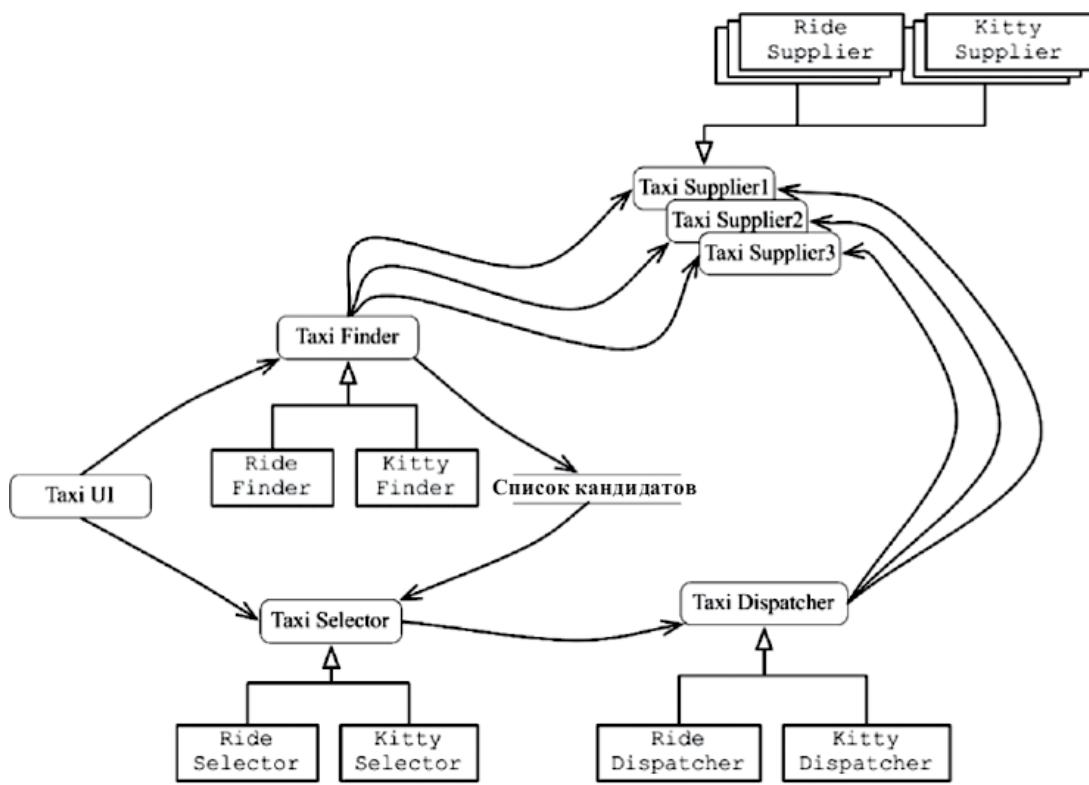


Рис. 27.3. Каждая служба состоит из компонентов, что позволяет добавлять новые возможности, реализованные в виде новых производных классов

Сквозные задачи

Теперь мы знаем, что архитектурные границы не всегда совпадают с границами между службами. Часто эти границы проходят через службы, деля их на компоненты.

Для преодоления проблем, связанных со сквозными задачами, с которыми сталкиваются все достаточно крупные системы, службы должны иметь компонентные архитектуры, следующие правилу зависимостей, как показано на рис. 27.4. Эти службы не определяют архитектурные границы в системе; границы определяются компонентами в службах.

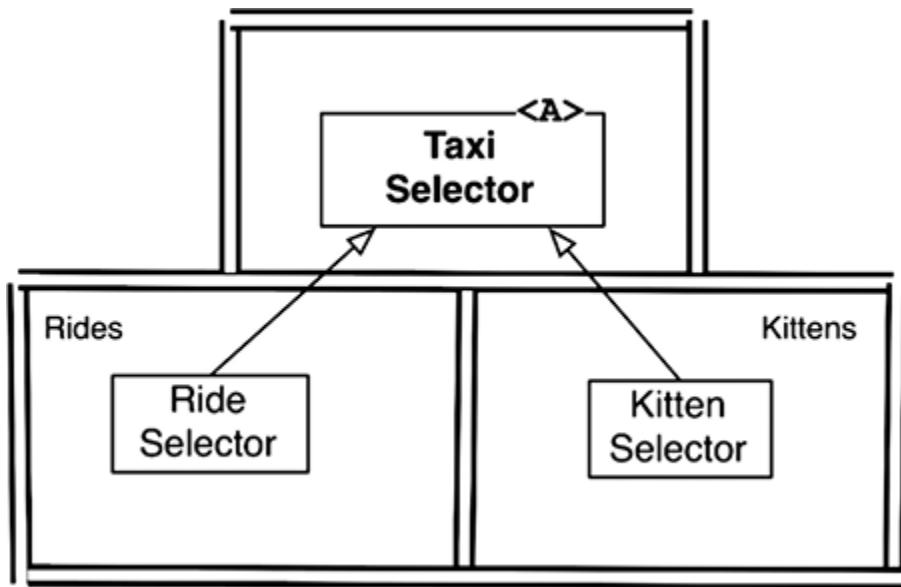


Рис. 27.4. Службы должны иметь компонентные архитектуры, следующие правилу зависимостей

Заключение

Несмотря на все преимущества масштабируемости и удобства разработки системы, службы не являются архитектурно значимыми элементами. Архитектура системы определяется границами, проводимыми внутри этой системы, и зависимостями, пересекающими эти границы. Архитектура не определяется физическими механизмами, посредством которых элементы взаимодействуют и выполняются.

Служба может быть единственным компонентом, полностью окруженным архитектурной границей. Но точно так

же служба может состоять из нескольких компонентов, разделенных архитектурными границами. В редких⁵⁴ случаях клиенты и службы могут быть настолько связаны, что не могут иметь архитектурной значимости.

[53](#) Примерно по числу программистов в команде.

[54](#) Хотелось бы надеяться, что в очень редких, но опыт показывает, что это не так.

28. Границы тестов



Да, все правильно: *тесты являются частью системы* и занимают свое место в архитектуре, как любые другие части системы. В одних случаях это вполне нормальное явление. В других оно может быть уникальным.

Тесты как компоненты системы

С тестами связано много неясностей. Являются ли они частью системы? Должны ли они отделяться от системы? Какие виды тестов бывают? Являются ли модульные и интеграционные

тесты разными тестами? Какое место во всем этом занимают приемочные тесты, функциональные тесты, тесты для фреймворка Cucumber, тесты TDD, тесты BDD, тесты для компонентов и т.д.?

Обсуждение данной конкретной темы не является целью этой книги, и, к счастью, этого не требуется. С архитектурной точки зрения все тесты одинаковы. Будь то маленькие тесты TDD, или большие тесты FitNesse, Cucumber, SpecFlow, или JBehave — все они архитектурно эквивалентны.

Тесты по самой своей природе следуют правилу зависимостей; они очень детальны и конкретны; и они всегда зависят от тестируемого кода. Фактически тесты можно считать самым внешним кругом архитектуры. Ничто в системе не зависит от тестов, но тесты всегда зависят от внутренних компонентов системы.

Тесты также можно развертывать независимо. Фактически в большинстве случаев они развертываются в тестовых системах, но не развертываются в производственных системах. То есть даже в системах, где независимое развертывание не требуется, тесты все равно развертываются независимо.

Тесты являются наиболее изолированными компонентами системы. Они не нужны системе для нормального функционирования. Пользователи не зависят от них. Их роль — поддержать разработку, но не работу. И все же они являются не менее важным системным компонентом, чем любые другие. Фактически они представляют модель, которой должны следовать все остальные компоненты.

Проектирование для простоты тестирования

Крайняя степень изоляции тестов в сочетании с тем фактом, что они обычно не развертываются, часто заставляет

разработчиков думать, что тесты выходят за рамки дизайна системы. Это абсолютно неверная точка зрения. Тесты, недостаточно хорошо интегрированные в дизайн системы, обычно оказываются хрупкими и делают систему жесткой и неудобной для изменения.

Проблема, конечно, заключается в тесной зависимости. Тесты, тесно связанные с системой, должны изменяться вместе с системой. Даже самые безобидные изменения в системном компоненте могут нарушить нормальную работу многих связанных тестов или потребовать их изменения.

Эта ситуация может приобретать особую остроту. Изменения в общих системных компонентах могут нарушить работу сотен и даже тысяч тестов. Эта ситуация известна как «проблема хрупких тестов».

Нетрудно понять, как это может произойти. Представьте, например, набор тестов, использующих графический интерфейс для проверки бизнес-правил. Такие тесты могут начинать работу на странице авторизации и затем последовательно переходить от страницы к странице, проверяя определенные бизнес-правила. Любое изменение в странице авторизации или в структуре навигации может нарушить работу большого количества тестов.

Хрупкие тесты часто оказывают отрицательное влияние, делая систему жесткой. Когда разработчики замечают, что простые изменения в системы вызывают массовые отказы тестов, они могут противиться таким изменениям. Например, представьте диалог между командой разработчиков и сотрудниками отдела маркетинга, просящими внести простое изменение в структуру навигации, которое нарушит работу 1000 тестов.

Решение заключается в том, чтобы закладывать возможность тестирования в проект. Первое правило

проектирования программного обеспечения — идет ли речь о тестируемости или о чем-то еще — всегда одно: *не зависеть ни от чего, что может часто меняться*. Пользовательские интерфейсы переменчивы. Наборы тестов, осуществляющие проверки посредством пользовательского интерфейса, должны быть хрупкими. Поэтому система и тесты должны проектироваться так, чтобы работу бизнес-правил можно было проверить без пользовательского интерфейса.

Программный интерфейс для тестирования

Для достижения этой цели следует создать специальный программный интерфейс для тестов, чтобы дать им возможность проверить все бизнес-правила. Этот API должен защищать тесты от проблем с ограничениями безопасности, не использовать дорогостоящие ресурсы (таких, как базы данных) и заставлять систему входить в особые тестируемые состояния. Такой API должен быть надмножеством *интеракторов* и *адаптеров интерфейсов*, которые используются пользовательским интерфейсом.

Цель API тестирования — отделить тесты от приложения. Под отделением подразумевается не только отделение тестов от пользовательского интерфейса: цель — отделить структуру тестов от структуры приложения.

Структурная зависимость

Структурная зависимость — одна из самых сильных и наиболее коварных форм зависимости тестов. Представьте набор тестов, в котором имеются тестовые классы для всех прикладных классов и тестовые методы для всех прикладных методов. Такой набор очень тесно связан со структурой приложения.

Изменение в одном из прикладных методов или классов может повлечь необходимость изменить большое количество тестов. Следовательно, тесты слишком хрупкие и могут сделать прикладной код слишком жестким.

Роль API тестирования — скрыть структуру приложения от тестов. Это позволит развивать прикладной код, не влияя на тесты. Это также позволит развивать тесты, не влияя на прикладной код.

Такая возможность независимого развития абсолютно необходима, потому что с течением времени тесты становятся все более конкретными, а прикладной код, напротив, — все более абстрактным и обобщенным. Тесная структурная зависимость препятствует такому развитию — или, по меньшей мере, затрудняет его — и мешает прикладному коду становиться все более обобщенным и гибким.

Безопасность

Открытость API тестирования может представлять опасность, если развернуть его в производственной системе. Если это действительно так, API тестирования и небезопасные части его реализации должны находиться в отдельном компоненте, устанавливаемом независимо.

Заключение

Тесты не находятся вне системы; они — часть системы, и к их проектированию следует подходить с неменьшим вниманием, чтобы получить от них выгоды в виде стабильности и защищенности от регрессий. Тесты, которые не проектируются как часть системы, получаются хрупкими и сложными в сопровождении. Такие тесты часто заканчивают свое

существование в комнате персонала, осуществляющего сопровождение, потому что их слишком тяжело поддерживать.

29. Чистая встраиваемая архитектура

Автор: Джеймс Греннинг (James Grenning)



Некоторое время тому назад я прочитал статью *The Growing Importance of Sustaining Software for the DoD⁵⁵* («Растущее значение устойчивого программного обеспечения для министерства обороны») в блоге Дуга Шмидта. В ней Дуг сделал следующее заявление:

Несмотря на то что программное обеспечение не изнашивается, встроенные микропрограммы и оборудование устаревают, что требует модификации программного обеспечения.

Этот момент кое-что прояснил для меня. Дуг упомянул два термина, которые я мог или не мог бы счесть очевидными. Программное обеспечение — это то, что имеет долгий срок

службы, но встроенные микропрограммы (firmware) устаревают в процессе развития аппаратного обеспечения. Если у вас есть опыт разработки встраиваемых систем, вы должны знать, что оборудование постоянно совершенствуется. В то же время добавляются новые функции в «программное обеспечение», и его сложность постоянно растет.

Я хотел бы добавить к заявлению Дуга:

Несмотря на то что программное обеспечение не изнашивается, оно может быть разрушено неуправляемыми зависимостями от микропрограмм и оборудования.

Нередко встроенному программному обеспечению может быть отказано в долгой жизни из-за заражения зависимостями от аппаратного обеспечения.

Мне нравится, как Дуг охарактеризовал микропрограммы, но давайте посмотрим, какие еще определения можно дать. Я, например, нашел следующие варианты:

- «Микропрограммы хранятся в энергонезависимых запоминающих устройствах, таких как ПЗУ, ППЗУ или флеш-память» (https://ru.wikipedia.org/wiki/Встроенное_программное_обеспечение).
- «Микропрограмма — это программа, или набор инструкций, заключенная в аппаратном устройстве» (<https://techterms.com/definition/firmware>).
- «Микропрограмма — это программное обеспечение, встроенное в устройство» (<https://www.lifewire.com/what-is-firmware-2625881>).
- Микропрограмма — это «программное обеспечение (программа или данные), записанное в постоянное

запоминающее устройство (ПЗУ)» (<http://www.webopedia.com/TERM/F/firmware.html>).

Заявление Дуга помогло мне заметить ошибочность всех этих общепринятых определений микропрограмм или, по крайней мере, их неактуальность. Название «микропрограмма» не подразумевает, что код хранится в ПЗУ. Принадлежность к категории микропрограмм не зависит от места хранения; в большей степени она зависит от сложности изменения в процессе совершенствования оборудования. Оборудование действительно совершенствуется (приостановитесь и взгляните на свой телефон), поэтому мы должны структурировать свой встраиваемый код с учетом этой данности.

Я ничего не имею против микропрограмм или разработчиков микропрограмм (я сам занимался созданием микропрограмм). Но мы действительно должны меньше писать микропрограммы и больше — программное обеспечение. На самом деле я разочарован тем, что разработчики микропрограмм пишут их как микропрограммы!

Инженеры, не занимающиеся разработкой встраиваемого программного обеспечения, тоже пишут микропрограммы! Вы тоже фактически пишете микропрограммы, когда внедряете SQL в свой код или когда ставите его в зависимость от платформы. Разработчики приложений для Android пишут микропрограммы, когда не отделяют бизнес-логику от Android API.

Я участвовал в разработке многих проектов, где грань между прикладным кодом (программным обеспечением) и кодом, взаимодействующим с оборудованием (микропрограммой), была размыта до полного исчезновения. Например, в конце 1990-х годов мне посчастливилось

участвовать в перепроектировании подсистемы коммуникации с целью перехода от технологии мультиплексирования с разделением по времени (Time-Division Multiplexing; TDM) к технологии передачи голоса по протоколу IP (Voice Over IP; VOIP). Технология VOIP широко используется в наши дни, а технология TDM считалась современной в 1950 – 1960-х годах и широко применялась в 1980 – 1990-х годах.

Всякий раз, когда у нас появлялся вопрос к инженеру-системотехнику о том, как вызов должен реагировать в той или иной ситуации, он исчезал и спустя какое-то время появлялся с очень подробным ответом. «Откуда ты это узнал?» — спрашивали мы. «Из кода продукта», — отвечал он. Запутанный и устаревший код служил справочником по новому продукту! Существующая реализация не имела разделения между TDM и бизнес-логикой, выполняющей вызовы. Весь продукт целиком зависел от оборудования/технологий, и этот клубок нельзя было распутать. Весь продукт фактически был микропрограммой.

Рассмотрим другой пример: управляющие сообщения поступают в систему через последовательный порт. Неудивительно, что в такой системе имеется обработчик/диспетчер сообщений. Обработчик сообщений знает их форматы, может их анализировать и передавать коду, который сгенерирует ответ. Ничто из перечисленного не вызывает удивления, кроме того, что обработчик/диспетчер сообщений находится в том же файле, что и код, взаимодействующий с микросхемой UART⁵⁶. Обработчик сообщений инфицирован деталями, имеющими отношение к микросхеме UART. Он мог бы быть программным обеспечением с потенциально большим сроком службы, но вместо этого он стал микропрограммой. Обработчику сообщений отказано в праве быть программным обеспечением — и это неправильно!

Я давно знал и понимал важность отделения программного обеспечения от оборудования, но слова Дуга прояснили, как использовать термины *программное обеспечение* и *микропрограмма* в отношении друг к другу.

Это ясное сообщение для инженеров и программистов: прекратите писать так много микропрограмм и дайте своему коду шанс служить долго. Конечно, потребовать этого не получится. Но давайте посмотрим, как сохранить архитектуру встраиваемого программного кода в чистоте, чтобы дать программному обеспечению шанс служить долго.

Тест на профпригодность

Почему так много программного обеспечения превращается в микропрограммы? Похоже, что основная причина заключается в стремлении получить действующий встраиваемый код и практически не уделяется внимания его структурированию для увеличения срока службы. Кент Бек описывает три шага в создании программного обеспечения (далее в кавычках приводятся слова Кента, а курсивом выделены мои комментарии):

1. «Сначала заставьте его работать». *Вы останетесь не у дел, если он не работает.*
2. «Затем перепишите его правильно». *Реорганизуйте код, чтобы вы и другие смогли понимать и развивать его, когда потребуется что-то изменить или понять.*
3. «Затем заставьте его работать быстро». *Реорганизуйте код, чтобы добиться «необходимой» производительности.*

Большая часть встраиваемых систем, которые мне приходилось видеть, похоже, писалась с единственной мыслью в голове: «Задавайте его работать», — и иногда с навязчивой идеей: «Задавайте его работать быстро», — воплощаемой введением микрооптимизаций при каждом удобном случае. В своей книге *The Mythical Man-Month*⁵⁷ Фред Брукс предлагает «планировать отказ от первой версии». Кент и Фред советуют практически одно и то же: узнайте, как это работает, и найдите лучшее решение.

Встраиваемое программное обеспечение ничем не отличается в отношении этих проблем. Многие невстраиваемые приложения доводятся только до стадии «работает», и мало что делается, чтобы код получился правильным и служил долго.

Получение работающего приложения — это то, что я называю тестом на профпригодность для программиста. Программист, разрабатывающий программное обеспечение, встраиваемое или нет, который заботится только о том, чтобы получить работающее приложение, наносит вред своим продуктам и работодателю. Программирование — это нечто большее, чем умение писать работающие приложения.

В качестве примера взгляните на следующие функции, находящиеся в одном файле маленькой встраиваемой системы, написанные в ходе прохождения теста на профпригодность:

```
ISR(TIMER1_vect) { ... }  
ISR(INT2_vect) { ... }  
void btn_Handler(void) { ... }  
float calc_RPM(void) { ... }  
static char Read_RawData(void) { ... }  
void Do_Average(void) { ... }
```

```
void Get_Next_Measurement(void) { ... }  
void Zero_Sensor_1(void) { ... }  
void Zero_Sensor_2(void) { ... }  
void Dev_Control(char Activation) { ... }  
char Load_FLASH_Setup(void) { ... }  
void Save_FLASH_Setup(void) { ... }  
void Store_DataSet(void) { ... }  
float bytes2float(char bytes[4]) { ... }  
void Recall_DataSet(void) { ... }  
void Sensor_init(void) { ... }  
void uC_Sleep(void) { ... }
```

В таком порядке функции были объявлены в файле с исходным кодом. А теперь разделим их и сгруппируем по решаемым задачам:

- функции, реализующие предметную логику:

- float calc_RPM(void) { ... }
- void Do_Average(void) { ... }
- void Get_Next_Measurement(void) { ... }
- void Zero_Sensor_1(void) { ... }
- void Zero_Sensor_2(void) { ... }

- функции, обслуживающие аппаратную платформу:

- ISR(TIMER1_vect) { ... }*

- `ISR(INT2_vect) { ... }`
- `void uC_Sleep(void) { ... }`
- функции, реагирующие на нажатия кнопок:
 - `void btn_Handler(void) { ... }`
 - `void Dev_Control(char Activation) { ... }`
- функция, читающая данные из аппаратного аналогово-цифрового преобразователя:
 - `static char Read_RawData(void) { ... }`
- функции, записывающие значения в долговременное хранилище:
 - `char Load_FLASH_Setup(void) { ... }`
 - `void Save_FLASH_Setup(void) { ... }`
 - `void Store_DataSet(void) { ... }`
 - `float bytes2float(char bytes[4]) { ... }`
 - `void Recall_DataSet(void) { ... }`
- функция, которая не делает того, что подразумевает ее имя:
 - `void Sensor_init(void) { ... }`

Заглянув в другие файлы этого приложения, я нашел множество препятствий, мешающих пониманию кода. Также я обнаружил, что организация файлов подразумевает единственный способ тестирования этого кода — непосредственно внутри целевого устройства. Практически каждый бит этого кода знает, что относится к специализированной микропроцессорной архитектуре, используя «расширенные» конструкции языка C⁵⁸, привязывающие код к конкретному набору инструментов и микропроцессору. У этого кода нет ни малейшего шанса служить долго, если будет решено перенести продукт на другую аппаратную платформу.

Приложение работает: инженер прошел тест на профпригодность. Но нельзя сказать, что оно имеет чистую встраиваемую архитектуру.

Привязка к оборудованию — узкое место

Существует масса особых проблем, с которыми приходится сталкиваться разработчикам встраиваемых систем и не знакомых разработчикам обычного программного обеспечения. Например, ограниченный объем памяти, ограничения по времени выполнения операций, ограниченные возможности ввода/вывода, нетрадиционные пользовательские интерфейсы, а также наличие датчиков и контактов, соединяющих с внешним миром. В большинстве случаев аппаратное обеспечение развивается параллельно с программным обеспечением и микропрограммами. У вас, как инженера, разрабатывающего код для такого рода систем, может не быть места для запуска кода. Но это еще не самое худшее — полученное оборудование может иметь собственные

недостатки, что замедляет разработку программного обеспечения больше, чем обычно.

Да, встраиваемое программное обеспечение имеет свои особенности, и инженеры встраиваемых систем — особые люди. Но разработка встраиваемых систем не настолько особенная, чтобы принципы, описываемые в этой книге, нельзя было применить к встраиваемым системам.

Одна из особых проблем встраиваемых систем — *тесная зависимость от оборудования*. Когда встраиваемый код структурируется без применения принципов и приемов чистой архитектуры, часто приходится сталкиваться со сценарием, когда код можно протестировать только на целевом оборудовании. Если это оборудование — единственное место, где возможно тестирование, такая тесная связь начинает замедлять вас.

Чистая встраиваемая архитектура — архитектура, поддерживающая тестирование

Давайте посмотрим, как применяются некоторые архитектурные принципы к встраиваемому программному обеспечению и микропрограммам и как они помогают избавиться от тесной привязки к оборудованию.

Уровни

Деление на уровни можно произвести разными способами. Начнем с трехуровневой организации, изображенной на рис. 29.1. Внизу находится уровень оборудования. Как предупреждает Дуг, вследствие совершенствования технологий и согласно закону Мура оборудование будет изменяться. Одни компоненты устаревают, и на смену им приходят новые компоненты, потребляющие меньше электроэнергии, или

имеющие более высокую производительность, или стоящие дешевле. Независимо от причин я, как инженер встраиваемых систем, не хотел бы делать больше, чем необходимо, когда неизбежное изменение оборудования наконец произойдет.

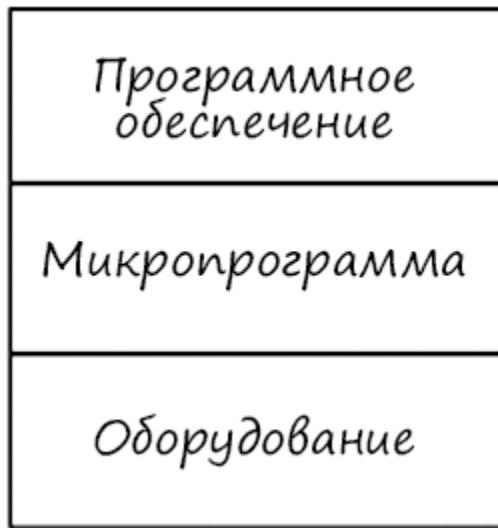


Рис. 29.1. Три уровня

Раздел между оборудованием и остальной частью системы — объективная реальность, по крайней мере после определения оборудования (рис. 29.2). Именно здесь часто возникают проблемы при попытке пройти тест на профпригодность. Ничто не мешает знаниям об оборудовании инфицировать весь код. Если не проявить осторожность при структурировании кода и не ограничить просачивание сведений об одном модуле в другой, код будет трудно изменить. Я говорю не только о случае, когда изменяется оборудование, но также о ситуации, когда понадобится исправить ошибку или внести изменение по требованию пользователя.



Рис. 29.2. Оборудование должно отделяться от остальной системы

Смешивание программного обеспечения и микропрограмм — это антишаблон. Код, демонстрирующий этот антишаблон, будет сопротивляться изменениям. Кроме того, изменения сопряжены с опасностями, часто ведущими к непредвиденным последствиям. Даже незначительные изменения требуют полного регрессионного тестирования системы. Если вы не создали тесты с внешним оборудованием, готовьтесь проводить тестирование вручную, а затем ожидать новых сообщений об обнаруженных ошибках.

Оборудование — это деталь

Линия между программным обеспечением и микропрограммами обычно видна не так четко, как линия, разделяющая код и оборудование (рис. 29.3).

Одна из задач разработчика встраиваемого программного обеспечения — укрепить эту линию. Границу между программным обеспечением и микро-



Рис. 29.3. Линия между программным обеспечением и микропрограммами обычно более размытая, чем линия между кодом и оборудованием

программой (рис. 29.4) называют слоем аппаратных абстракций (Hardware Abstraction Layer; HAL). Это не новая идея: она была реализована в персональных компьютерах еще в эпоху до Windows.

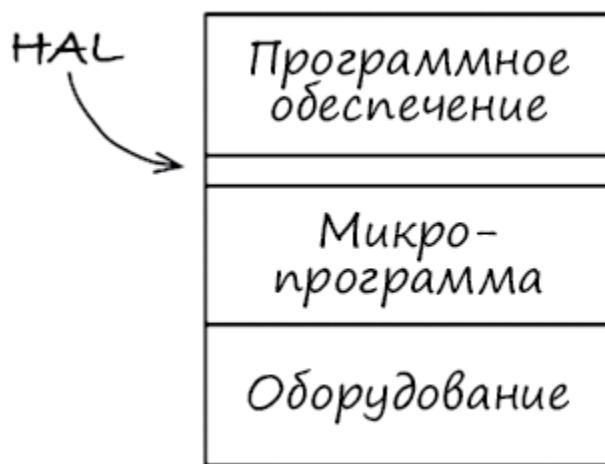


Рис. 29.4. Слой аппаратных абстракций

Слой HAL существует для программного обеспечения над ним, и его API должен приспосабливаться под потребности этого программного обеспечения. Например, микропрограмма может хранить байты и массивы байтов во флеш-памяти, а

приложению требуется хранить и читать пары имя/значение с использованием некоторого механизма хранения. Программное обеспечение не должно заботиться о деталях хранения пар имя/значение во флеш-памяти, на вращающемся диске, в облаке или в основной памяти. Слой аппаратных абстракций (HAL) предоставляет услугу и не раскрывает программному обеспечению, как она работает. Реализация поддержки флеш-памяти — это деталь, которая должна быть скрыта от программного обеспечения.

Еще один пример: управление светодиодом привязано к биту GPIO⁵⁹. Микропрограмма может предоставлять доступ к битам GPIO, а слой HAL может иметь функцию `Led_TurnOn(5)`. Это довольно низкоуровневый слой аппаратной абстракции. Давайте посмотрим, как повысить уровень абстракции с точки зрения программного обеспечения/продукта. Какую информацию сообщает светодиод? Допустим, что включение светодиода сообщает о низком заряде аккумулятора. На некотором уровне микропрограмма (или пакет поддержки платформы) может предоставлять функцию `Led_TurnOn(5)`, а слой HAL — функцию `Indicate_LowBattery()`. Таким способом слой HAL выражает назначение услуги для приложения. Кроме того, уровни могут содержать подуровни. Это больше напоминает повторяющийся фрактальный узор, чем ограниченный набор предопределенных уровней. Назначение ввода/выводов GPIO — это детали, которые должны быть скрыты от программного обеспечения.

Не раскрывайте деталей об оборудовании пользователям HAL

Встраиваемое программное обеспечение с чистой архитектурой допускает возможность тестирования без

целевого оборудования. Удачно спроектированный слой аппаратных абстракций (HAL) предоставляет тот шов, или набор, точек подстановки, которые облегчат тестирование без оборудования.

Процессор – это деталь

Когда сборка встраиваемого программного обеспечения производится с применением специализированного набора инструментов, в комплект часто входят заголовочные файлы с <i>поддержкой дополнительных конструкций </i>⁶⁰. Эти компиляторы часто допускают вольное обращение с языком C, добавляя новые ключевые слова для доступа к функциям процессора. Код выглядит как код на C, но он больше не является кодом на C.

Иногда компиляторы языка C, поставляемые производителем оборудования, поддерживают нечто, напоминающее глобальные переменные, дающие прямой доступ к регистрам процессора, портам ввода/вывода, таймерам, битам интерфейсов ввода/вывода, контроллерам прерываний и другим функциям процессора. Их удобно использовать для доступа ко всему перечисленному, но вы должны понимать, что код, использующий эти вспомогательные средства, больше не является кодом на языке C. Он не будет компилироваться для другого процессора или даже другим компилятором для этого же процессора.

Не хотелось бы думать, что производитель оборудования цинично привязывает ваш продукт к своему компилятору, поэтому будем считать, что он искренне желает помочь. Но теперь вам решать — как использовать эту помощь, чтобы не навредить себе же в будущем. Вам придется ограничить круг

файлов, которым будет известно о нестандартных расширениях языка С.

Взгляните на следующий заголовочный файл, созданный для семейства ACME процессоров цифровой обработки сигналов:

```
#ifndef _ACME_STD_TYPES
#define _ACME_STD_TYPES
#if defined(_ACME_X42)
    typedef unsigned int  Uint_32;
    typedef unsigned short Uint_16;
    typedef unsigned char  Uint_8;

    typedef int          Int_32;
    typedef short         Int_16;
    typedef char          Int_8;

#elif defined(_ACME_A42)
    typedef unsigned long  Uint_32;
    typedef unsigned int   Uint_16;
    typedef unsigned char  Uint_8;

    typedef long          Int_32;
    typedef int            Int_16;
    typedef char           Int_8;

#else
    #error <acmetypes.h> is not supported for this environment
#endif
```

```
#endif
```

Заголовочный файл `acmetypes.h` не следует использовать непосредственно, иначе ваш код окажется тесно связанным с процессорами ACME. Вы думаете, что если используете процессор ACME, то какой вред он может причинить? Все просто, вы не сможете скомпилировать свой код, не подключив этот заголовочный файл. А если подключить его и определить символ `_ACME_X42` или `_ACME_A42`, целые числа будут иметь неправильный размер при тестировании за пределами целевой платформы. Какое-то время это может не вызывать никаких проблем, но однажды у вас может появиться желание перенести свое приложение на другой процессор, и тогда вы обнаружите, что задача сильно усложнилась из-за отказа от переносимости и ограничения на подключение файлов, знающих о процессорах ACME.

Вместо использования `acmetypes.h` следует попробовать пойти более стандартным путем и использовать `stdint.h`. Но как быть, если в состав целевого компилятора не входит файл `stdint.h`? Вы можете сами написать этот заголовочный файл. Файл `stdint.h` может использовать `acmetypes.h`, когда выполняется компиляция для целевой платформы:

```
#ifndef _STDINT_H_
#define _STDINT_H_
#include <acmetypes.h>
typedef UInt_32 uint32_t;
typedef UInt_16 uint16_t;
typedef UInt_8 uint8_t;

typedef Int_32 int32_t;
typedef Int_16 int16_t;
```

```
typedef Int_8 int8_t;
```

```
#endif
```

Использование `stdint.h` во встраиваемом программном обеспечении и микропрограммах поможет вам сохранить код чистым и переносимым. Всякое *программное обеспечение* должно быть независимым от типа процессора, но этот совет годится не для всякой *микропрограммы*. В следующем фрагменте кода используются особые расширения языка C, позволяющие коду обращаться к периферийным устройствам в микроконтроллере. Продукт может быть оснащен этим микроконтроллером, поэтому вы можете использовать интегрированные в него периферийные устройства. Следующая функция выводит в последовательный порт строку с текстом "hi". (Этот пример основан на реальном коде.)

```
void say_hi()
{
    IE = 0b11000000;
    SBUF0 = (0x68);
    while(TI_0 == 0);
    TI_0 = 0;
    SBUF0 = (0x69);
    while(TI_0 == 0);
    TI_0 = 0;
    SBUF0 = (0x0a);
    while(TI_0 == 0);
    TI_0 = 0;
    SBUF0 = (0x0d);
    while(TI_0 == 0);
```

```
    TI_0 = 0;  
    IE = 0b11010000;  
}
```

Эта маленькая функция страдает множеством проблем. Первое, что бросается в глаза, — присутствие последовательности символов `0b11000000`. Такая форма записи двоичных чисел очень удобна, но поддерживается ли она стандартным языком C? К сожалению, нет. Еще несколько проблем проистекают непосредственно из использования нестандартных расширений:

- IE: устанавливает биты разрешения прерываний.
- SBUF0: буфер вывода последовательного порта.
- TI_0: прерывание опустошения буфера передачи последовательного порта. Если операция чтения возвращает 1, это указывает, что буфер пуст.

Переменные с именами, состоящими из букв верхнего регистра, в действительности представляют механизмы доступа к встроенной периферии микроконтроллера. Если программе понадобится управлять прерываниями и выводить символы, вам придется использовать эту периферию. Да, это удобно, но это уже не язык C.

Чистая архитектура будет напрямую использовать эти средства доступа к периферии лишь в нескольких местах и только в пределах микропрограммы. Любой код, знающий о существовании регистров, автоматически превращается в микропрограмму и, соответственно, оказывается тесно связанным с конкретным оборудованием. Тесная связь кода с оборудованием мешает получить действующий код до

получения стабильно работающего оборудования. Она также будет мешать переносить встраиваемое приложение на новый процессор.

Если вы используете микроконтроллер, подобный этому, ваша программа могла бы спрятать эти низкоуровневые функции за *слоем абстракций процессора* (Processor Abstraction Layer; PAL). Часть микропрограммы, находящаяся над слоем PAL, могла бы проверять платформу, на которой выполняется, и таким способом ослабить жесткость кода.

Операционная система – это деталь

Слой аппаратных абстракций (HAL) является насущной необходимостью, но достаточно ли его? Во встраиваемых системах, где отсутствует другое программное окружение, слоя HAL более чем достаточно, чтобы оградить код от избыточной зависимости от операционной среды. Но что, если встраиваемая система использует некоторую операционную систему реального времени (RealTime Operating System; RTOS) или встраиваемую версию Linux или Windows?

Чтобы дать коду шанс служить долго, операционную систему следует рассматривать как деталь и защищаться от зависимостей, связывающих с ней.

Программное обеспечение обращается к операционному окружению посредством операционной системы (ОС). ОС – это слой, отделяющий программное обеспечение от микропрограмм (рис. 29.5). Прямое использование механизмов ОС может стать источником проблем. Например, представьте,

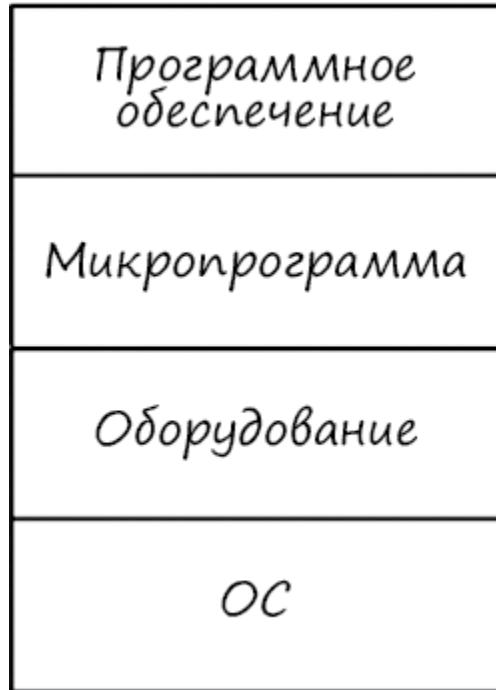


Рис. 29.5. Дополнительный слой операционной системы

что производителя вашей RTOS купила другая компания и из-за этого выросла стоимость системы или упало ее качество. Или ваши потребности изменились, а используемая вами RTOS не обладает необходимыми возможностями. Вам придется изменить много кода. И это будут не просто синтаксические изменения, обусловленные сменой API, скорее всего, вам придется приспособливать семантику кода к различным механизмам и примитивам новой ОС.

Чистая встраиваемая архитектура изолирует программное обеспечение от операционной системы, реализуя слой *абстракции операционной системы* (Operating System Abstraction Layer; OSAL), как показано на рис. 29.6. В некоторых случаях этот слой может иметь очень простую реализацию, выражющуюся в простой подмене имен функций. Но иногда может потребоваться полное обертывание некоторых функций.

Если вам доводилось переносить программное обеспечение с одной RTOS на другую, вы знаете, насколько трудно это

дается. Если ваше программное обеспечение зависит только от слоя OSAL, но не зависит от ОС, вам потребуется только написать новый слой OSAL, совместимый с прежним. Что бы вы предпочли: изменить кучу сложного кода или написать новый код, определяющий интерфейс и поведение? Это даже не спорный вопрос. Я выбираю последнее.

Вас может беспокоить проблема разбухания кода. Однако вам не о чем беспокоиться. Большая часть повторяющегося кода, обусловленного использованием операционной системы, будет сосредоточена в слое абстракции. Такой повторяющийся код не будет вызывать больших накладных расходов.

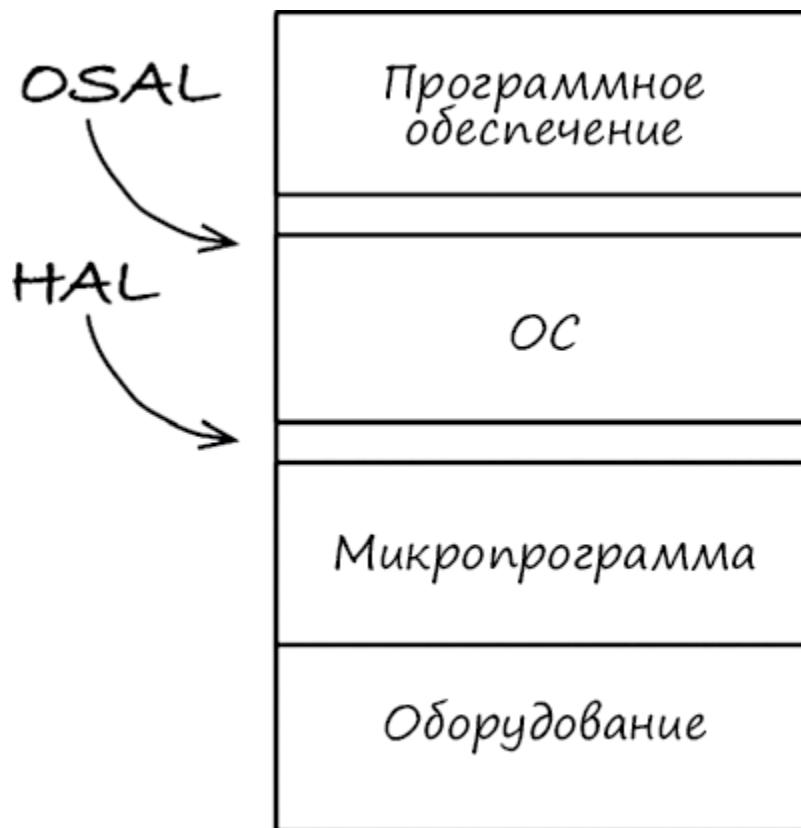


Рис. 29.6. Слой абстракции операционной системы

Определяя слой OSAL, вы также получаете возможность привести свои приложения к некоторой общей структуре. Например, вы могли бы реализовать механизмы передачи

сообщений и отказаться от ручного управления конкурентным выполнением в каждом потоке.

Слой OSAL может помочь создать точки для тестирования, чтобы прикладной программный код можно было тестировать без использования целевого оборудования и ОС. Программное обеспечение с чистой встраиваемой архитектурой поддерживает возможность тестирования вне целевой операционной системы. Удачно спроектированный слой OSAL предоставляет тот шов, или набор, точек подстановки, которые облегчат тестирование вне целевой среды.

Программирование с применением интерфейсов и возможность подстановки

Помимо добавления слоев HAL и, возможно, OSAL, внутри каждого из основных уровней (программное обеспечение, ОС, микропрограмма и оборудование) можно и должно применять принципы, описанные в этой книге. Эти принципы способствуют разделению ответственности, программированию с применением интерфейсов и возможности подстановки.

Идея многоуровневой архитектуры основывается на идеи программирования с применением интерфейсов. Когда один модуль взаимодействует с другим посредством интерфейса, появляется возможность заменить одного поставщика услуг другим. Многим из вас наверняка приходилось писать свои небольшие версии `printf` для развертывания в целевой среде. Если интерфейс вашей функции `printf` совпадает с интерфейсом стандартной версии `printf`, вы можете использовать их взаимозаменямо.

Главное правило — использовать заголовочные файлы как определения интерфейсов. Однако такой подход требует

внимательно следить за тем, что помещается в заголовочный файл. Желательно не помещать в него ничего, кроме объявлений функций, а также констант и имен структур, необходимых функциям.

Не загромождайте интерфейсные заголовочные файлы структурами данных, константами и определениями типов, которые нужны только реализации. Это не просто вопрос беспорядка: подобный беспорядок приводит к нежелательным зависимостям. Ограничьте видимость деталей реализации. Исходите из предположения, что детали реализации будут изменяться. Чем меньше мест, где код знает детали, тем меньше кода вам придется просматривать и изменять.

Чистая встраиваемая архитектура позволяет выполнять послойное тестирование, потому что модули взаимодействуют посредством интерфейсов. Каждый интерфейс обеспечивает шов, или точку подстановки, для тестирования вне целевого окружения.

Принцип DRY и директивы условной компиляции

Одно из использований возможности подстановки, которое часто упускают из виду, связано с особенностями обработки разных целевых платформ или операционных систем программами на С и С++. Для этого часто используются директивы условной компиляции, включающие и выключающие сегменты кода. Я вспоминаю один особенно тяжелый случай, когда в телекоммуникационном приложении директива `#ifdef BOARD_V2` встречалась несколько тысяч раз.

Повторяющийся код нарушает принцип «не повторяйся» (Don't Repeat Yourself; DRY)⁶¹. Если `#ifdef BOARD_V2` встречается один раз, в этом нет никакой проблемы. Но *шесть тысяч раз — это большая проблема*. Условная компиляция,

идентифицирующая тип целевого оборудования, часто используется во встраиваемых системах. Можно ли от нее избавиться?

Можно, если реализовать слой аппаратных абстракций и скрыть определение типа оборудования в этом слое. Если слой HAL предоставляет набор интерфейсов, вместо условной компиляции можно взвалить всю работу на компоновщика (редактора связей) или использовать некоторую форму связывания во время выполнения для подключения программного обеспечения к оборудованию.

Заключение

Программисты, разрабатывающие встраиваемое программное обеспечение, могут многое взять из опыта разработки обычного программного обеспечения. Если вы занимаетесь разработкой встраиваемого программного обеспечения, то найдете в этой книге много мудрых и полезных советов и идей, которые пригодятся и вам.

Не позволяйте всему вашему коду превращаться в микропрограммы — это вредно для долгого и доброго здоровья вашего продукта. Возможность тестирования исключительно в целевом окружении — вредна для долгого и доброго здоровья вашего продукта. Следование принципам чистой встраиваемой архитектуры — полезно для долгого и доброго здоровья вашего продукта.

55 https://insights.sei.cmu.edu/sei_blog/2011/08/the-growing-importance-of-sustaining-software-for-the-dod.html

56 Микросхема, управляющая последовательным портом.

57 Фредерик Брукс. Мифический человеко-месяц, или Как создаются программные системы. СПб.: Символ-Плюс (2007). — Примеч. пер.

[58](#) Некоторые производители микроконтроллеров добавляют свои ключевые слова в язык C, чтобы упростить доступ к регистрам и портам из кода на C. К сожалению, как только код начинает использовать такие ключевые слова, он перестает быть кодом на C.

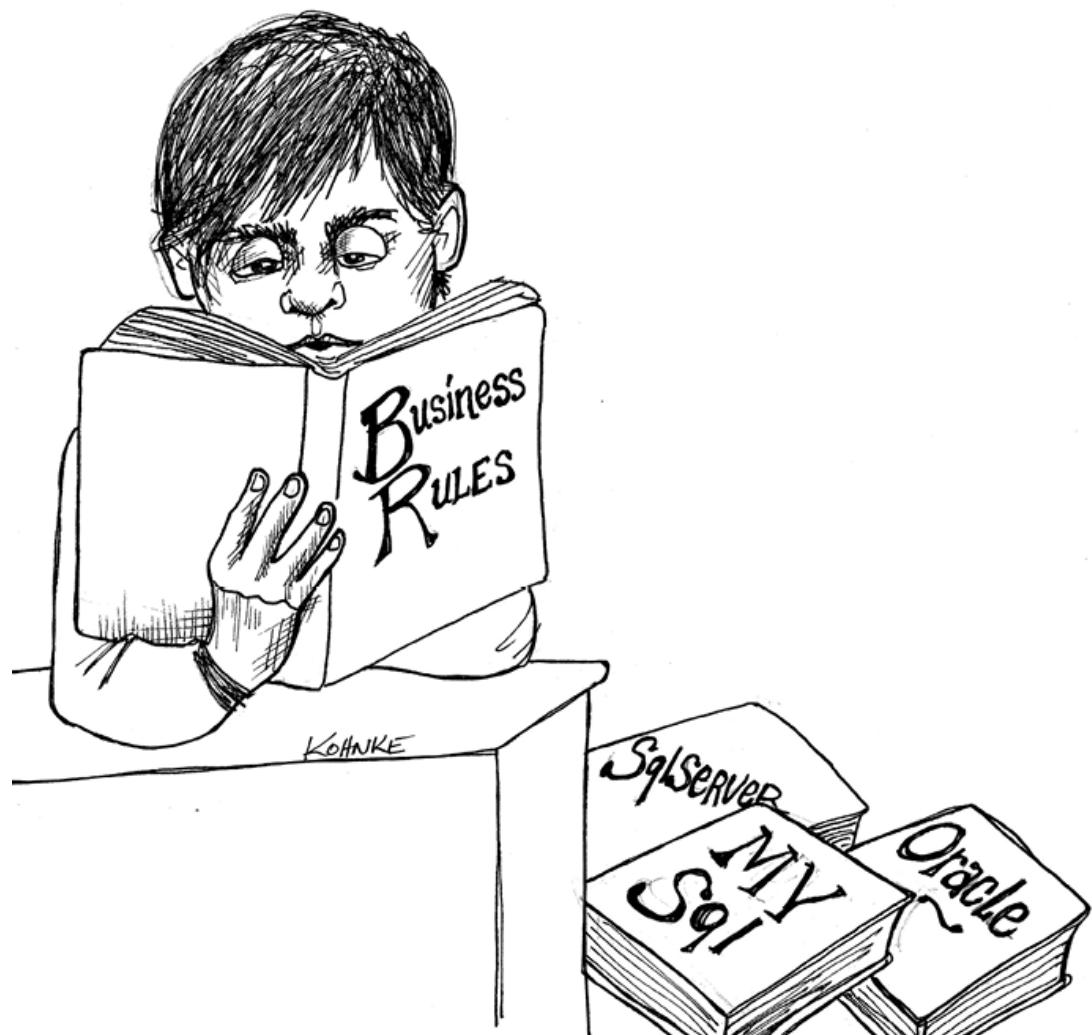
[59](#) General-Purpose Input/Output (интерфейс ввода/вывода общего назначения). — Примеч. пер.

[60](#) В этом предложении преднамеренно использована разметка HTML.

[61](#) Hunt and Thomas, *The Pragmatic Programmer*. (Э. Хант, Д. Томас. Программист-прагматик. Путь от подмастерья к мастеру. М.: Лори, 2016. — Примеч. пер.)

VI. Детали

30. База данных – это деталь



С архитектурной точки зрения база данных не является сущностью — это деталь, которая не должна подниматься до уровня архитектурного элемента. Ее отношение к архитектуре программной системы сопоставимо с отношением дверной ручки к архитектуре здания.

Я понимаю, что эти слова могут показаться провокационными. Но верьте мне, я знаю, о чем говорю. А теперь позвольте мне пояснить: я не имею в виду модель

данных. Структура, которую вы придаете данным в своем приложении, очень важна для архитектуры системы. Но база данных — это не модель данных. База данных — это часть программного обеспечения. База данных — это утилита, обеспечивающая доступ к данным. С архитектурной точки зрения эта утилита не имеет никакого значения, это низкоуровневая деталь — механизм. А хороший архитектор не позволяет низкоуровневым механизмам просачиваться в архитектуру системы.

Реляционные базы данных

Принципы реляционных баз данных были определены Эдгаром Коддом в 1970 году. К середине 1980-х годов реляционная модель разрослась и превратилась в доминирующую форму хранения данных. Такая популярность появилась не на пустом месте: реляционная модель элегантна, дисциплинирована и надежна. Это отличная технология хранения и доступа к данным.

Но какой бы блестящей, полезной и близкой к математике ни выглядела технология, она не перестает быть просто технологией. А значит, она — деталь.

Реляционные таблицы удобны для некоторых видов доступа к данным, но в упорядочивании данных по записям в таблицах нет ничего архитектурно значимого. Варианты использования в приложении не должны знать и заботиться о таких вещах. В действительности о табличной организации данных должны знать только низкоуровневые вспомогательные функции во внешних кругах архитектуры.

Многие фреймворки доступа к данным позволяют передавать записи и таблицы из базы данных в виде объектов через всю систему. Но такой способ действий является

архитектурной ошибкой. Он связывает варианты использования, бизнес-правила, а в некоторых случаях даже пользовательский интерфейс с определенной реляционной структурой данных.

Почему системы баз данных настолько распространены?

Почему среди программных систем и корпоративных программных продуктов доминирующее положение заняли системы баз данных? Чем объясняется превосходство Oracle, MySQL и SQL Server? Если говорить одним словом — диски.

Пять десятилетий вращающийся магнитный диск был основой хранения данных. Несколько поколений программистов не знали иных форм хранения данных. Технология производства дисковых накопителей прошла долгий путь от огромных пакетов массивных пластин по 48 дюймов в диаметре, весивших тысячи килограммов и способных хранить 20 мегабайт, до тонких дисков по 3 дюйма в диаметре, весящих несколько граммов и способных хранить терабайт данных и даже больше. *Это была сумасшедшая гонка.* И на протяжении всей этой гонки программисты страдали от одного фатального свойства дисков: они работают слишком медленно.

Данные хранятся на диске на круговых дорожках. Дорожки делятся на секторы, хранящие определенное круглое число байтов, обычно 4 Кбайт. На каждой пластине может иметься несколько сотен дорожек, а в одном дисковом накопителе — десяток или около того пластин. Чтобы прочитать конкретный байт, устройство должно поместить головку на конкретную дорожку, дождаться, пока диск повернется так, что требуемый сектор окажется под головкой, прочитать все 4 Кбайт из сектора в ОЗУ, и затем вам останется только прочитать

требуемый байт из ОЗУ. Но все это требует времени — миллисекунд времени.

Может показаться, что миллисекунды — это немного, но миллисекунда в миллион раз продолжительнее одного такта большинства процессоров. Если бы данные хранились не на диске, к ним можно было бы получить доступ не за миллисекунды, а за наносекунды.

Чтобы уменьшить задержку, вносимую дисками, нужны индексы, кэши и оптимизированные схемы запросов; а еще нужны какие-то регулярные средства представления данных, чтобы все эти индексы, кэши и схемы запросов знали, с чем они работают. Проще говоря, нужен доступ к данным и система управления. За годы развития эти системы разделились на два разных типа: файловые системы и системы управления реляционными базами данных (СУРБД).

Файловые системы опираются на понятие документа. Они поддерживают естественный и удобный способ хранения целых документов. Они хорошо работают, когда требуется сохранить или извлечь набор документов по имени, но не предлагают сколько-нибудь существенной помощи в поиске по содержимому документов. Вы легко найдете файл с именем `login.c`, но поиск всех файлов `.c`, содержащих переменную с именем `x`, будет долгим и трудным.

Системы баз данных ориентированы на содержимое. Они поддерживают естественный и удобный способ поиска записей по содержимому. Они очень хорошо связывают вместе несколько записей, имеющих общее содержимое. К сожалению, их преимущества теряются при хранении и извлечении непрозрачных документов.

Обе системы организуют хранение данных на диске так, чтобы обеспечить максимальную эффективность их сохранения и чтения, с учетом конкретных особенностей.

Каждая имеет свою схему индексирования и организации данных. Кроме того, обе переносят соответствующие данные в ОЗУ, где ими можно быстро манипулировать.

Сохраняются ли диски?

Прежде занимающие доминирующие позиции, ныне диски превратились в вымирающий вид. В скором времени их ожидает судьба накопителей на магнитной ленте, гибких дисков и приводов компакт-дисков. Им на смену придет ОЗУ.

Спросите себя: когда диски исчезнут и все данные будут храниться в ОЗУ, как вы будете организовывать их? Вы предпочтете организовать их в таблицы и обращаться к ним, используя запросы на языке SQL? Или организовать их в файлы и обращаться к ним через систему каталогов?

Конечно нет. Вы организуете их в связные списки, деревья, хеш-таблицы, стеки, очереди или любые другие структуры данных и будете обращаться к ним, используя указатели или ссылки, потому что программисты так привыкли.

На самом деле, поразмыслив об этой проблеме, вы поймете, что все это вы уже делаете. Несмотря на то что данные хранятся в базе данных или в файловой системе, вы читаете их в ОЗУ и затем реорганизуете в удобные списки, множества, стеки, очереди, деревья или любые другие структуры по своему усмотрению. Очень маловероятно, что вы оставите данные в форме файлов или таблиц.

Детали

Такое положение дел объясняет, почему я говорю, что база данных — это деталь. Это лишь механизм перемещения данных взад-вперед между поверхностью диска и ОЗУ. База

данных в действительности — не более чем большой мешок с битами, в котором мы храним свои данные. Но мы редко используем данные именно в такой форме.

То есть с архитектурной точки зрения мы не должны беспокоиться о том, какую форму принимают данные, пока хранятся на поверхности вращающегося магнитного диска. В действительности нас вообще не должно волновать наличие или отсутствие диска.

А производительность?

Является ли производительность архитектурной проблемой? Конечно! Но в отношении хранения данных эту проблему можно полностью инкапсулировать и отделить от бизнес-правил. Да, нам нужно, чтобы данные быстро перемещались из хранилища и в хранилище, но это низкоуровневая проблема. Ее можно решить с помощью низкоуровневых механизмов доступа к данным. И она не имеет ничего общего с архитектурой системы.

История

В конце 1980-х годов я возглавлял команду разработчиков программного обеспечения в начинающей компании, которая пыталась создать и продать систему управления сетью, оценившую целостность телекоммуникационных линий Т1. Система извлекала данные из устройств в конечных точках этих линий, а затем запускала ряд алгоритмов для обнаружения проблем.

Мы использовали платформы UNIX и хранили наши данные в простых файлах. Нам не нужна была реляционная база данных, потому что наши данные были слабо связаны между

собой. Намного удобнее было хранить их в простых файлах, в виде деревьев и связных списков. Короче говоря, мы хранили данные в форме, упрощающей их загрузку в ОЗУ.

Мы наняли специалиста по маркетингу — хорошего знающего парня. Но он тут же заявил, что в нашей системе должна иметься реляционная база данных. Этого не требовалось для нормальной работы, и это не было технической проблемой — это была проблема маркетинга.

Для меня это было бессмысленно. Почему я должен превращать связные списки и деревья в кучу записей и таблиц, доступных через SQL? Почему я должен вводить накладные расходы и тратить деньги на приобретение массивной СУРБД, если простых файлов более чем достаточно? Поэтому я вступил в сражение с ним, пустив в ход зубы и когти.

В этой компании работал один инженер-электронщик, который направо и налево расхваливал СУРБД. Он имел свое, техническое обоснование необходимости СУРБД в нашей системе. За моей спиной он встретился с руководством компании, нарисовал изображение дома на шесте и спросил у руководства: «Хотели бы вы жить в доме на шесте?» Его посыл был прост: он считал, что СУРБД, хранящая таблицы в файлах, надежнее простых файлов, которые мы использовали.

Я боролся с ним. Я боролся с маркетологом. Я придерживался своих технических принципов перед лицом невероятного невежества. Я боролся, боролся и боролся.

В конце концов электронщик пошел на повышение и стал руководителем отдела программного обеспечения. Они втиснули СУРБД в эту несчастную систему. И в результате они оказались абсолютно правы, а я ошибался.

Нет, с технической точки зрения, заметьте, я был прав. Я был прав, что боролся с внедрением СУРБД в архитектурное ядро системы. Но я был не прав, потому что клиенты ожидали

получить реляционную базу данных в нашей системе. Они не знали, что с ней делать. Они не представляли, как использовать реляционные данные из нее. Но это не имело значения: клиенты рассчитывали получить СУРБД. Она была галочкой, которую все покупатели программного обеспечения ставили в своих списках. Не было никакого технического обоснования — рациональность здесь вообще была ни при чем. Это была иррациональная, внешняя и полностью необоснованная потребность, но это не делало ее менее реальной.

Откуда это взялось? Причина кроется в высокоэффективных маркетинговых кампаниях, широко проводимых производителями баз данных в то время. Им удалось убедить высокопоставленных руководителей, что их корпоративные «активы данных» нуждаются в защите, а предлагаемые ими системы баз данных являются идеальным средством такой защиты.

Ныне мы наблюдаем похожие маркетинговые кампании. Слово «корпоративный» и понятие «сервис-ориентированная архитектура» имеют больше общего с маркетингом, чем с реальностью.

Что я должен был сделать в описанной ситуации? Я должен был прикрутить СУРБД к системе и дать ей узкий и безопасный канал доступа к данным, сохранив в ядре системы простые файлы. А что я сделал? Я ушел и стал консультантом.

Заключение

Организационная структура и модель данных являются архитектурно значимыми, а технологии и системы, перемещающие данные на вращающуюся магнитную поверхность, — нет. Системы реляционных баз данных, которые заставляют организовывать данные в таблицы и

обращаться к ним посредством SQL, имеют гораздо больше общего с последними, чем с первыми. Данные — значимы, а база данных — это деталь.

31. Веб – это деталь



Вы занимались разработкой в 1990-х годах? Помните, как Всемирная паутина изменила все вокруг? Помните, с каким презрением мы смотрели на наши старые архитектуры клиент/сервер перед лицом новой сверкающей технологии «Веб»?

На самом деле Веб ничего не изменил. По крайней мере не должен был ничего изменить. Веб — это всего лишь последнее из последовательности колебаний, которые наша индустрия пережила с 1960-х годов. Эти колебания заставляют переносить всю вычислительную мощность то на центральные серверы, то на терминалы.

Только за последние десять лет (или около того), пока Веб занял заметное место, мы видели несколько таких колебаний. Сначала мы думали, что вся вычислительная мощность будет сосредоточена в фермах серверов, а браузеры будут просто отображать информацию. Затем мы начали добавлять апплеты в браузеры. Но нам это не понравилось, поэтому мы переместили динамическое содержимое обратно на серверы. Но это нам тоже не понравилось, и мы изобрели Веб 2.0, переместив почти всю обработку обратно в браузер, используя для этого Ajax и JavaScript. Мы зашли так далеко, что создали целые приложения, выполняющиеся в браузерах. И теперь мы снова возбуждены перемещением JavaScript обратно на сервер в виде Node.

(Вздох.)

Бесконечный маятник

Конечно, было бы неправильно думать, что колебания начались с появлением Веб. До Веб существовала архитектура клиент/сервер. До этого были центральные мини-компьютеры с массивами неинтеллектуальных терминалов. До этого были мейнфреймы с интеллектуальными терминалами (зеленого свечения, которые были очень похожи на современные браузеры). До этого были машинные залы и перфокарты...

Так выглядит история. Мы никак не можем определить, где должна быть сосредоточена вычислительная мощность. Мы все

время то централизуем ее, то распределяем. И я думаю, что эти колебания будут еще продолжаться в течение какого-то времени.

С точки зрения истории развития информационных технологий Веб вообще ничего не изменил. Это просто одно из множества колебаний, которые начались еще до того, как многие из нас появились на свет, и продолжатся после того, как многие из нас уйдут на пенсию.

Однако как архитекторы мы должны быть нацелены на долгосрочную перспективу. Эти колебания — лишь кратковременные проблемы, которые не должны проникать в центральное ядро с нашими бизнес-правилами.

Я хочу рассказать вам одну историю о компании Q. Компания Q создала очень популярную систему управления персональными финансами. Это было приложение для настольного компьютера с очень удобным графическим интерфейсом. Мне нравилось пользоваться им.

Затем появился Веб. В следующей версии системы компания Q изменила пользовательский интерфейс — он стал выглядеть и действовать подобно веб-интерфейсу в браузере. Я был в шоке! Какой гений маркетинга решил, что программа управления персональными финансами, действующая на настольном компьютере, должна выглядеть как веб-браузер?

Конечно, я возненавидел новый интерфейс. Все остальные, видимо, тоже, потому что после выпуска нескольких версий компания Q отказалась от внешнего вида, напоминающего браузер, и вернула системе графический интерфейс обычного настольного приложения.

Теперь представьте, что в ту пору вы работали программным архитектором в Q. Вообразите, что какой-то гений от маркетинга убеждает высшее руководство в необходимости переделать весь пользовательский интерфейс,

чтобы он напоминал веб-интерфейс. Что бы вы сделали? Точнее, что вы должны были бы сделать до этого момента, чтобы защитить приложение от этого гения маркетинга?

Вы должны были бы отделить бизнес-правила от пользовательского интерфейса. Я не знаю, как поступили архитекторы в Q. Я хотел бы когда-нибудь услышать их рассказ. Если бы в то время я работал там, я бы точно изолировал бизнес-правила от пользовательского интерфейса, потому что никто не знает, что взбредет в голову гениям маркетинга в следующий раз.

Теперь рассмотрим пример с компанией A, выпускающей замечательные смартфоны. Недавно она выпустила обновленную версию своей «операционной системы» (это так необычно — говорить об операционной системе внутри телефона). Кроме всего прочего модернизация кардинально изменила внешний вид всех приложений. Зачем? Я полагаю, что вновь не обошлось без какого-то гения маркетинга.

Я не эксперт по программному обеспечению в этих устройствах, поэтому не знаю, вызвали ли эти изменения какие-либо трудности для программистов приложений, выполняющихся на смартфонах компании A. Я надеюсь, что архитекторы в A и архитекторы приложений изолировали бизнес-правила и пользовательский интерфейс друг от друга, потому что всегда найдутся гении маркетинга, которые только и ждут, чтобы наброситься на что-то, созданное вами.

Вывод

Вывод из всего вышесказанного прост: пользовательский интерфейс — это деталь. Веб — это пользовательский интерфейс. То есть Веб — это деталь. И как архитектор вы

должны размещать детали, как эта, за границами, отделяющими их от основной бизнес-логики.

Подумайте об этом с такой позиции: *Веб — это устройство ввода/вывода*. Еще в 1960-х годах мы поняли ценность методики создания приложений, не зависящих от устройств ввода/вывода. Побудительные мотивы такой независимости не изменились. Веб не является исключением из этого правила.

Или я не прав? Можно утверждать, что графический интерфейс, как и Веб, настолько уникalen и богат, что абсурдно продолжать создавать архитектуры, независимые от устройства. Когда вы думаете о тонкостях проверки входных данных на JavaScript, особенностях технологии «перетащил и бросил» в AJAX или о том, какие еще виджеты/гаджеты вставить в веб-страницу, легко можно скатиться до утверждения, что независимость устройства непрактична.

Все это верно до определенной степени. Взаимодействия приложения с пользовательским интерфейсом обширны, многообразны и порой очень специфичны для данного интерфейса. Веб-приложение и веб-браузер взаимодействуют совершенно иначе, чем настольное приложение и графический интерфейс. Попытка абстрагироваться от этих взаимодействий, подобно тому, как в UNIX абстрагируются устройства, выглядит невероятной.

Но границу между пользовательским интерфейсом и приложением можно абстрагировать. Бизнес-логику можно рассматривать как набор вариантов использования, каждый из которых выполняет некоторую функцию от имени пользователя. Каждый вариант использования можно описать как комплекс входных данных, выполняемую обработку комплекс и выходных данных.

В какой-то момент в череде взаимодействий пользовательского интерфейса и приложения подготовка

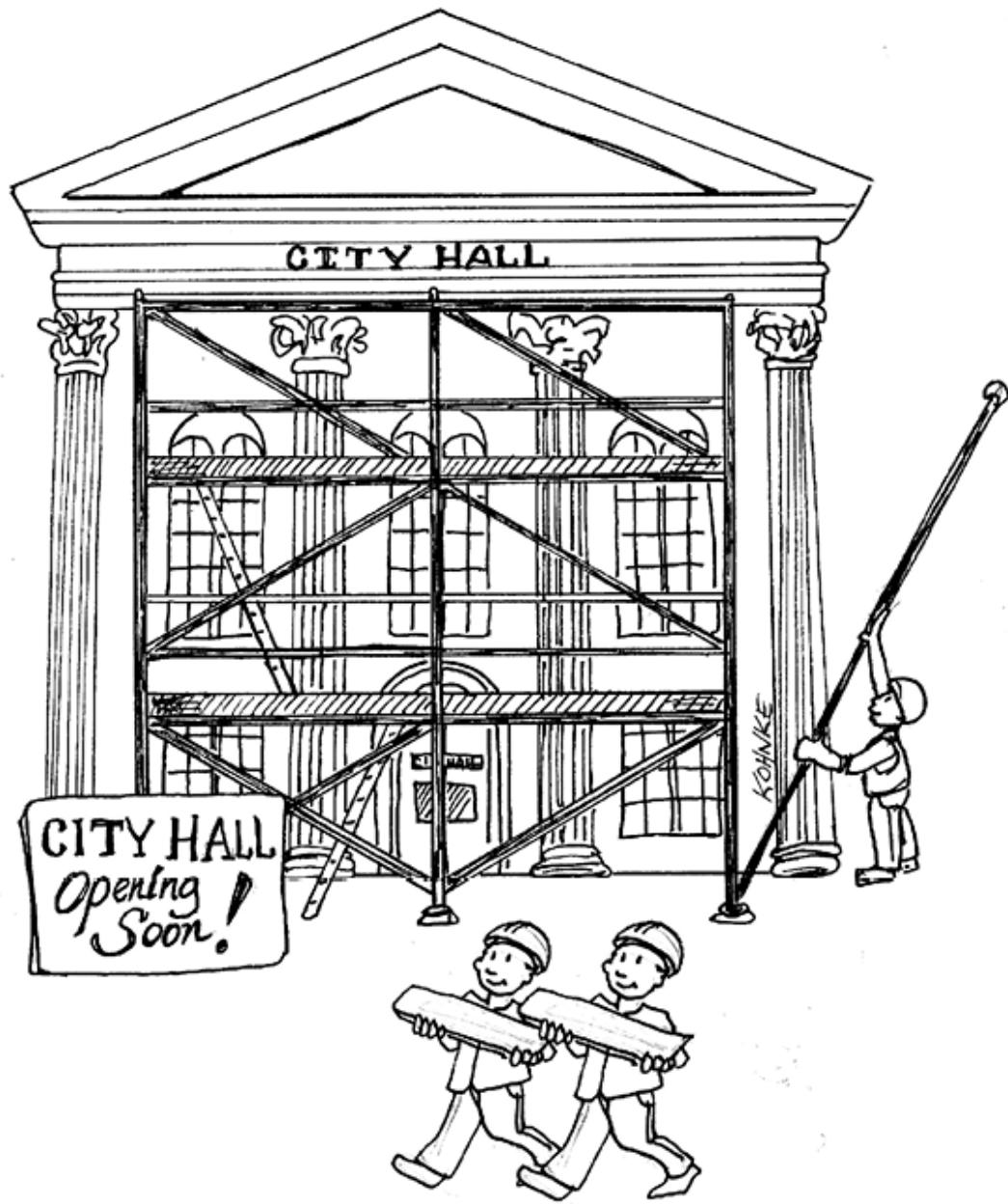
входных данных будет закончена и появится возможность выполнить вариант использования. По завершении полученные данные можно вернуть обратно, включив их во взаимодействия между пользовательским интерфейсом и приложением.

Входные и выходные данные можно поместить в структуры и использовать как входные и выходные значения для процесса, выполняющего вариант использования. При таком подходе можно считать, что каждый вариант использования работает с пользовательским интерфейсом как с устройством ввода/вывода независимо от этого устройства.

Заключение

Достичь такой абстракции очень непросто, и, скорее всего, потребуется несколько итераций, чтобы достичь желаемого результата. Но это возможно. А поскольку мир полон гениев от маркетинга, это не самое сложное, потому что часто бывает необходимо.

32. Фреймворки – это деталь



Фреймворки обрели большую популярность. Вообще говоря, это хорошая штука. Существует много бесплатных, мощных и полезных фреймворков.

Но фреймворки не определяют архитектуру, хотя некоторые пытаются.

Авторы фреймворков

Большинство авторов фреймворков предлагают результаты своего труда бесплатно, потому что хотят быть полезными для сообщества. Они хотят внести свой вклад. Это похвально. Однако, независимо от благородства мотивов, авторы не думают о ваших интересах. И не могут думать, потому что они не знают ни вас, ни ваших проблем.

Но они знают свои проблемы и проблемы своих коллег и друзей. И они пишут свои фреймворки для решения этих проблем, а не ваших.

Конечно, ваши проблемы, скорее всего, частично будут совпадать с этими другими проблемами. Иначе фреймворки не были бы столь популярны. Поэтому фреймворки могут быть очень полезными — в той мере, в какой существует совпадение.

Неравный брак

Отношения между вами и автором фреймворка чрезвычайно асимметричны. Вы должны обладать огромной приверженностью фреймворку, но его автор не несет никаких обязательств перед вами.

Задумайтесь над этим. Когда вы используете фреймворк, вы читаете документацию, предоставленную его автором. В этой документации автор и другие пользователи фреймворка рассказывают вам, как интегрировать ваше программное обеспечение с фреймворком. Обычно это означает, что вы должны обернуть фреймворк своей архитектурой. Автор

рекомендует использовать базовые классы и импортировать инструменты фреймворка в свои бизнес-объекты. Автор настоятельно рекомендует вам *привязать* ваше приложение к фреймворку настолько, насколько это возможно.

Для автора тесная связь с его собственным фреймворком не является проблемой. Для автора такая связь *желательна*, потому что он обладает абсолютным контролем над этим фреймворком.

Более того, автор хочет привязать вас к фреймворку, потому что, привязавшись, будет очень трудно отвязаться. Ничто не подтверждает авторитет автора так, как группа пользователей, готовых порождать свои классы на основе базовых классов автора.

Фактически автор предлагает вам вступить в союз с фреймворком — выразить ему большую и долговременную приверженность. Но при этом сам автор ни при каких обстоятельствах не выражает ответной приверженности вам. Это односторонний союз. Вы принимаете на себя все риски и неудобства; автор фреймворка не берет на себя ничего.

Риски

Какие риски, спросите вы? Вот лишь некоторые из них.

- Часто архитектура фреймворков не отличается особой чистотой. Фреймворкам свойственно нарушать правило зависимости. Авторы предлагают вам наследовать их код в ваших бизнес-объектах — ваших сущностях! Они хотят внедрить свои фреймворки в самый внутренний круг. Как только это произойдет, убрать фреймворк оттуда будет почти невозможно. Надев обручальное кольцо на палец, он останется жить там.

- Фреймворк может помочь с реализацией первых возможностей приложения. Однако, достигнув некоторого уровня зрелости, ваш продукт перерастет фреймворк. Если вы надели это обручальное кольцо, то со временем заметите, что вам все чаще и чаще приходится бороться с фреймворком.
- Фреймворк может развиваться в направлении, бесполезном для вас. Вы можете столкнуться с необходимостью обновления до следующей версии, не несущей ничего, что могло бы пригодиться вам. Вы можете даже обнаружить, что старые особенности, которыми вы пользовались, исчезают или изменяются так, что вам трудно поспевать за ними.
- Может появиться новый, более удачный фреймворк, на который вы захотите перейти.

Решение

И каково решение?

Не заключать союзов с фреймворками!

О, вы можете использовать фреймворк — просто не привязывайтесь к нему. Держите его на расстоянии вытянутой руки. Рассматривайте фреймворк как деталь, принадлежащую одному из внешних кругов архитектуры. Не впускайте его во внутренние круги.

Если фреймворк предложит вам породить свои бизнес-объекты от его базовых классов, скажите «нет»! Определите прокси-классы и держите их в компонентах, являющихся *плагинами* для ваших бизнес-правил.

Не позволяйте фреймворкам проникать в ваш основной код. Внедряйте их в компоненты, которые лишь подключаются

к вашему основному коду, как того требует правило зависимости.

Например, возможно, вам нравится Spring — хороший фреймворк внедрения зависимостей. Возможно, вы используете Spring для автоматического внедрения своих зависимостей. Это замечательно, но вы не должны окроплять аннотациями `@Autowired` свои бизнес-объекты. Бизнес-объекты не должны знать о существовании Spring.

Но вы с успехом можете использовать Spring для внедрения зависимостей в своем компоненте `Main`. Этот компонент может знать о фреймворке Spring, потому что он является самым грязным и самым низкоуровневым компонентом в системе.

Объявляю вас

Есть некоторые фреймворки, с которыми вы просто обязаны вступить в союз. Если, например, вы пишете на C++, вам почти наверняка придется вступить в союз с STL — избежать этого очень сложно. Если вы пишете на Java, вы будете вынуждены вступить в союз со стандартной библиотекой.

Это нормально — но все равно это должно быть осознанное решение. Вы должны понимать, что, заключив союз между фреймворком и вашим приложением, вам придется придерживаться этого фреймворка в течение всего жизненного цикла этого приложения. В радости и в горе, в болезни и в здоровье, в богатстве и в бедности, отказавшись от всех остальных, вы будете использовать этот фреймворк. Этот шаг нельзя делать по легкомыслию.

Заключение

Встретившись с фреймворком, не торопитесь вступать с ним в союз. Посмотрите, есть ли возможность отложить решение. Если это возможно, удерживайте фреймворк за архитектурными границами. Может быть, вам удастся найти способ получить молоко, не покупая корову.

33. Практический пример: продажа видео



Теперь самое время объединить изученные нами правила и принципы создания архитектур и применить их для реализации учебного примера. Пример будет коротким и простым, но его будет вполне достаточно, чтобы

продемонстрировать процесс, которому следует хороший архитектор, и принимаемые им решения.

Продукт

Для этого примера я выбрал продукт, с которым довольно близко знаком: программное обеспечение для веб-сайта, продающего видеоматериалы. Да, он напоминает cleancoders.com — сайт, где я продаю свои обучающие видеофильмы.

Основная идея проста. У нас есть партия видеофильмов, которые требуется продать. Мы продаем их через Веб физическим лицам и компаниям. Физические лица могут заплатить одну, более низкую, цену за потоковое видео, или другую, более высокую, за загрузку копий и их постоянное использование. Бизнес-лицензии распространяются только на потоковую доставку и приобретаются партиями, что позволяет предоставлять скидки.

Физические лица обычно выступают в роли зрителей или покупателей. В компаниях, напротив, есть люди, которые покупают видеофильмы для просмотра другими.

Авторы видеофильмов должны предоставлять свои видеофайлы, сопровождая их описанием и вспомогательными файлами с контрольными вопросами, решениями, исходным кодом и другими материалами.

Администраторам необходимо добавлять новые видеоролики, добавлять видеоролики в серию и удалять их, а также устанавливать цены на различные лицензии.

Администраторы должны добавлять коллекции видеофильмов, удалять коллекции и устанавливать цены на различные лицензии.

Наш первый шаг в определении начальной архитектуры системы — выявление действующих лиц и вариантов использования.

Анализ вариантов использования

На рис. 33.1 изображена схема типичного анализа вариантов использования.

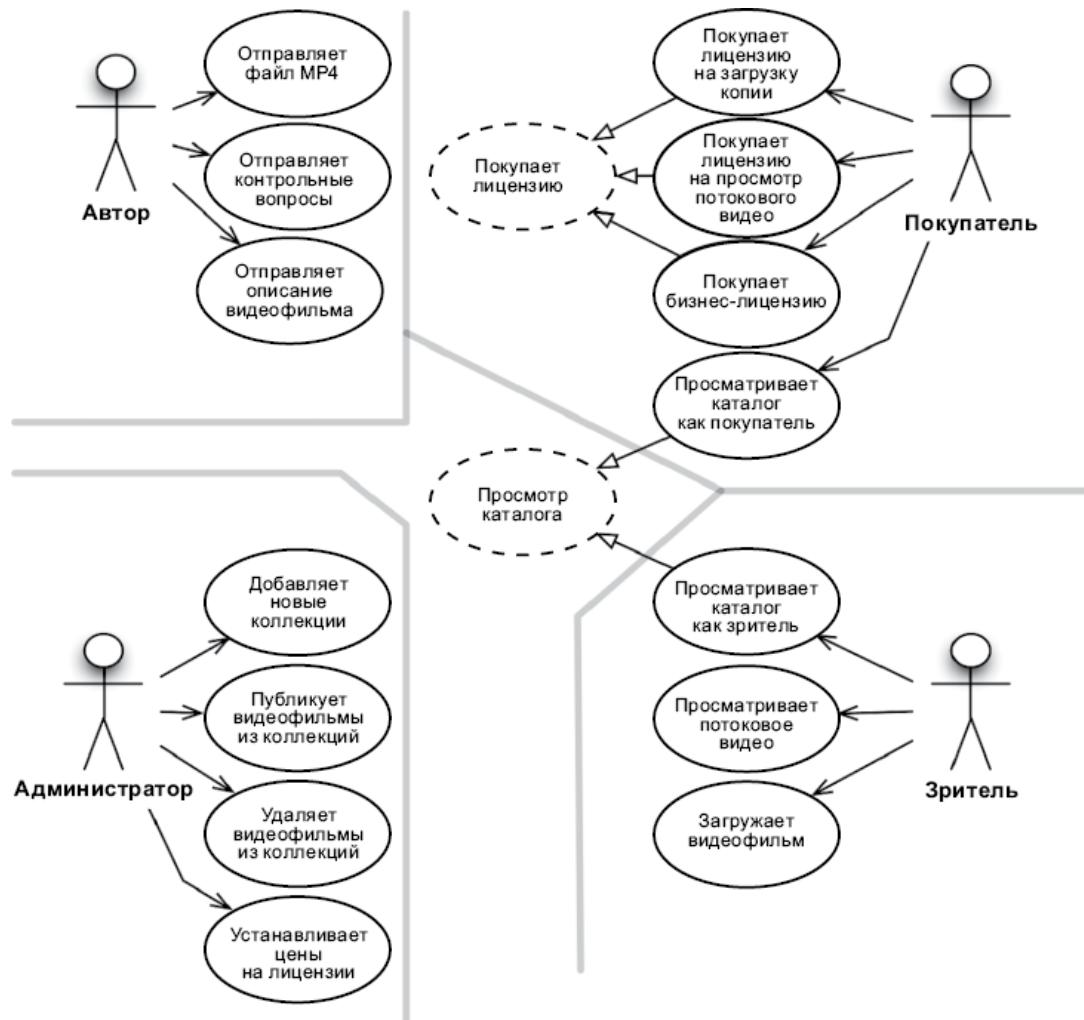


Рис. 33.1. Типичный анализ вариантов использования

На этой схеме изображены четыре действующих лица. В соответствии с принципом единственной ответственности, эти

четыре действующих лица станут основными источниками изменений для системы. Добавление любых новых возможностей или изменение существующих будет выполняться ради обслуживания одного из действующих лиц. Поэтому мы должны разбить систему так, чтобы изменения, предназначенные для одного действующего лица, не затрагивали других.

На рис. 33.1 изображены не все варианты использования. Например, на схеме отсутствуют варианты, отвечающие за вход и выход. Это сделано специально, чтобы упростить задачу и ограничить ее размер. Если бы я включил все возможные варианты использования, эта глава могла бы превратиться в самостоятельную книгу.

Обратите внимание на варианты использования в центре на рис. 33.1, заключенные в пунктирные рамки. Это *абстрактные*⁶² варианты использования. Абстрактным называется такой вариант использования, который определяет общую политику для других вариантов использования. Как видите, абстрактный вариант использования *Просмотр каталога* наследует варианты *Просматривает каталог как зритель* и *Просматривает каталог как покупатель*.

С одной стороны, эту абстракцию можно было не создавать. Я мог бы убрать абстрактный вариант использования из схемы без ущерба для каких-либо особенностей продукта. С другой стороны, эти два варианта использования настолько похожи, что я считал разумным признать сходство и найти способ унифицировать его с самого начала.

Компонентная архитектура

Теперь, когда известны действующие лица и варианты использования, можно создать предварительную

компонентную архитектуру (рис. 33.2).

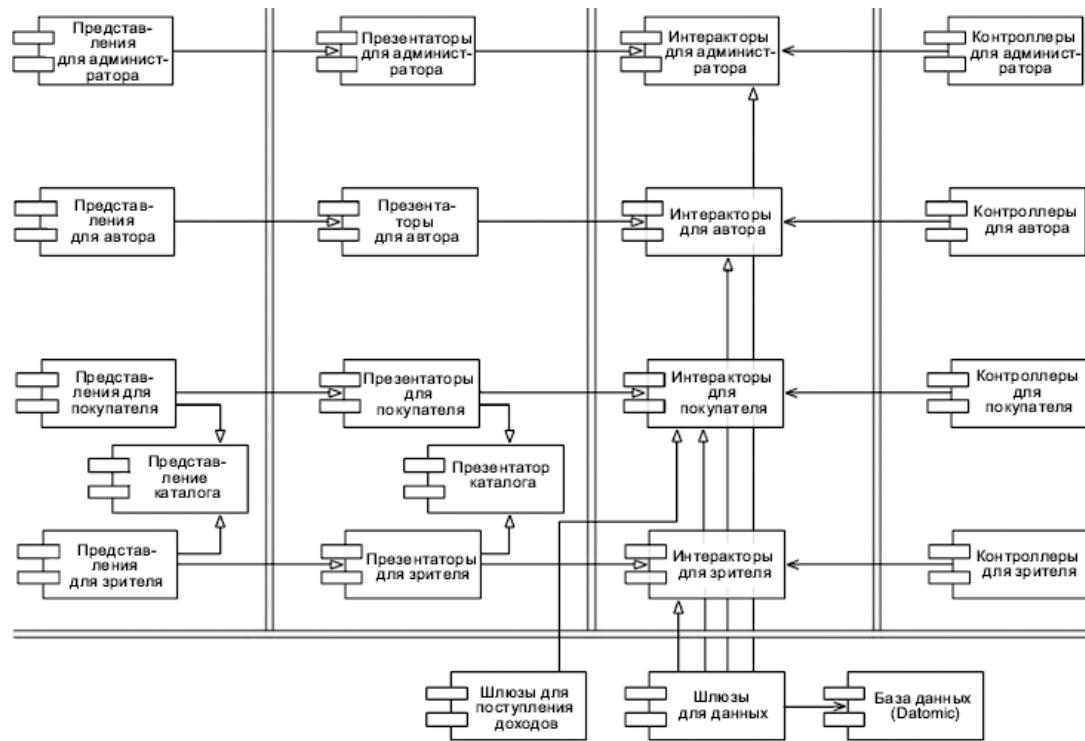


Рис. 33.2. Предварительная компонентная архитектура

Двойными линиями, как обычно, изображены архитектурные границы. Как видите, это типичное деление на представления, презентаторы, интеракторы и контроллеры. Также отметьте, что каждую из этих категорий я разбил на подкатегории по соответствующим им действующим лицам.

Каждый компонент на рис. 33.2 потенциально представляет файл `.jar` или `.dll` и каждый будет содержать соответствующие представления, презентаторы, интеракторы и контроллеры.

Обратите внимание на особые компоненты *Представление каталога* и *Презентатор каталога*. Именно так предполагается реализовать абстрактный вариант использования *Просмотр каталога*. Я предполагаю, что такие представления и презентаторы будут иметь вид абстрактных классов, которые

будут наследовать конкретные классы представлений и презентаторов в производных компонентах.

Стал бы я действительно разбивать систему на все эти компоненты и организовывать их в файлы `.jar` или `.dll`? И да и нет. Конечно, я мог бы помудрить над компиляцией и настроить окружение так, чтобы действительно можно было создавать компоненты, развертываемые независимо. Также я мог бы оставить за собой право объединить компоненты, чтобы при необходимости получить меньшее количество единиц развертывания. Например, компоненты, изображенные на рис. 33.2, легко можно объединить в пять файлов `.jar` — по одному для представлений, презентаторов, интеракторов, контроллеров и утилит соответственно. В этом случае я мог развертывать компоненты, которые, скорее всего, будут меняться независимо друг от друга.

Еще один возможный вариант группировки: объединить представления и презентаторы в один файл `.jar`, а интеракторы, контроллеры и утилиты поместить в свои, отдельные файлы. Другой, еще более простой вариант группировки: создать два файла `.jar` — один с представлениями и презентаторами и другой со всем остальным.

Оставив эти возможности открытыми, позже мы сможем изменить способ развертывания системы, исходя из особенностей ее развития.

Управление зависимостями

Поток управления на рис. 33.2 движется справа налево. Входные данные поступают в контроллеры и затем обрабатываются интеракторами. Презентаторы форматируют

результаты обработки и передают их представлениям для отображения.

Обратите внимание, что не все стрелки направлены справа налево. Фактически большинство из них направлено слева направо. Это объясняется тем, что архитектура следует *правилу зависимости*. Все зависимости пересекают архитектурные границы в одном направлении и всегда направлены в сторону компонентов, содержащих политики более высокого уровня.

Также отметьте, что отношения *использования* (открытые стрелки) совпадают с направлением потока управления, а отношения *наследования* (закрытые стрелки) направлены против потока управления. Это отражает использование принципа открытости/закрытости, который требует, чтобы зависимости были направлены в правильном направлении и изменения в низкоуровневых деталях не затрагивали высокоуровневые политики.

Заключение

Диаграмма архитектуры на рис. 33.2 включает разделение по двум измерениям. Первое — разделение на основе действующих лиц согласно принципу единственной ответственности; второе соответствует правилу зависимости. Цель обоих — разделить компоненты, изменяющиеся по разным причинам и с разной скоростью. Причины в данном случае соответствуют действующим лицам, а скорости — разным уровням политик.

После структуризации кода таким способом появляется возможность смешивать и распределять его по единицам развертывания как угодно. Вы сможете группировать компоненты в любые единицы развертывания, имеющие смысл, и менять правила группировки с изменением условий.

[62](#) Это мой собственный стиль выделения абстрактных вариантов использования. Я мог бы использовать более стандартный UML-стереотип, такой как <<abstract>>, но в данный момент я не считаю полезным следовать таким стандартам.

34. Недостающая глава

Автор: Симон Браун (*Simon Brown*), 2 марта 2017



Все советы, которые вы прочитали к настоящему моменту, безусловно, помогут вам проектировать замечательные приложения, состоящие из классов и компонентов с четко

определенными границами, понятными обязанностями и управляемыми зависимостями. Но как всегда, дьявол кроется в деталях реализации, и действительно, очень легко споткнуться о последнее препятствие, если не уделить ему должного внимания.

Представим, что мы строим книжный онлайн-магазин и один из вариантов использования, который нам предлагается внедрить, — возможность просмотра клиентами состояния своих заказов. Пример, следующий ниже, описывается с позиции языка Java, однако принципы в равной степени применимы к другим языкам программирования. Отложим пока чистую архитектуру в сторону и рассмотрим несколько подходов к проектированию и организации кода.

Упаковка по уровням

Первый и самый простой, пожалуй, подход — организация традиционной многоуровневой архитектуры, в которой код разделяется по функциональному признаку. Этот подход часто называют «упаковкой по уровням». На рис. 34.1 показано, как могла бы выглядеть соответствующая UML-диаграмма классов.

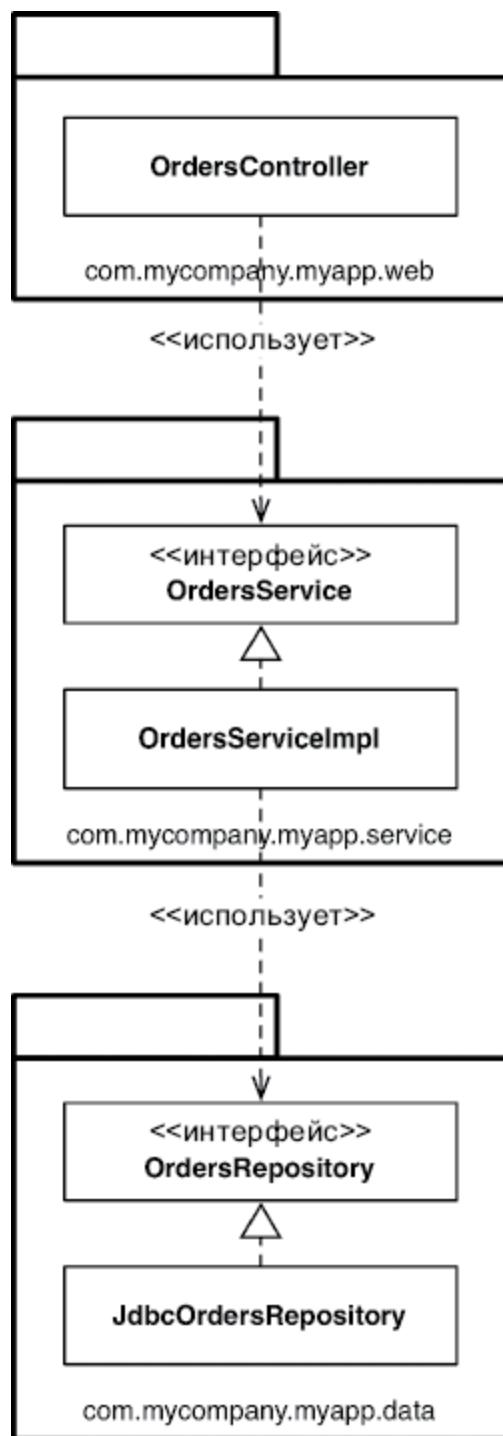


Рис. 34.1. Упаковка по уровням

В такой типичной многоуровневой архитектуре один уровень выделяется для веб-кода, один уровень — для «бизнес-логики» и один уровень — для работы с хранилищем данных. Иными словами, горизонтальные уровни используются как способ группировки по подобию. В «строгой многоуровневой архитектуре» уровни должны зависеть только от следующего смежного уровня. В Java уровни обычно реализуются в виде пакетов. Как показано на рис. 34.1, все зависимости между уровнями (пакетами) направлены вниз. В этом примере у нас имеются следующие Java-типы:

- `OrdersController`: веб-контроллер, иногда что-то вроде MVC-контроллера из Spring, обрабатывающий запросы из Веб.
- `OrdersService`: интерфейс, определяющий «бизнес-логику», связанную с заказами.
- `OrdersServiceImpl`: реализация службы заказов⁶³.
- `OrdersRepository`: интерфейс, определяющий порядок доступа к информации о заказах в хранилище.
- `JdbcOrdersRepository`: реализация интерфейса хранилища.

В своей статье *Presentation Domain Data Layering*⁶⁴ («Многоуровневая организация: представление, бизнес-логика, данные») Мартин Фаулер написал, что такая трехуровневая организация отлично подходит для начального этапа. И он не одинок. Во многих книгах, руководствах, курсах и примерах кода демонстрируются способы создания многоуровневой архитектуры. Это очень быстрый способ без особых

затруднений создать и запустить что-то. Проблема, как указывает Мартин, в том, что с ростом масштаба и сложности программного обеспечения трех больших слоев кода оказывается недостаточно и приходится задумываться о более дробной организации.

Другая проблема, как уже сказал «дядюшка Боб»⁶⁵, — многоуровневая архитектура не кричит о своем практическом назначении. Поместите рядом код двух многоуровневых архитектур из разных предметных областей, и они почти наверняка будут выглядеть пугающе похожими: веб-интерфейсы, службы и хранилища. Многоуровневые архитектуры страдают еще от одного большого недостатка, но мы поговорим о нем позже.

Упаковка по особенностям

Другой вариант организации кода — «упаковка по особенностям». Это разделение по вертикали, основанное на объединении связанных особенностей, предметных понятий и общих корней (если использовать терминологию предметно-ориентированного проектирования). В типичных реализациях, которые мне доводилось видеть, все типы помещаются в один Java-пакет, имя которого отражает идею группировки.

При такой организации, изображенной на рис. 34.2, имеются те же интерфейсы и классы, но они помещаются в единый Java-пакет. Ее легко получить простым рефакторингом из «упаковки по уровням», но теперь верхнеуровневая структура кричит о предметной области. Теперь видно, что кодовая база имеет какое-то отношение к заказам, а не к Веб, не к службам и не к хранилищам. Другим преимуществом является относительная простота поиска кода для изменения, например, когда потребуется изменить вариант использования

«просмотр заказов». Весь код сосредоточен вместе, а не разбросан по разным Java-пакетам^{[66](#)}.

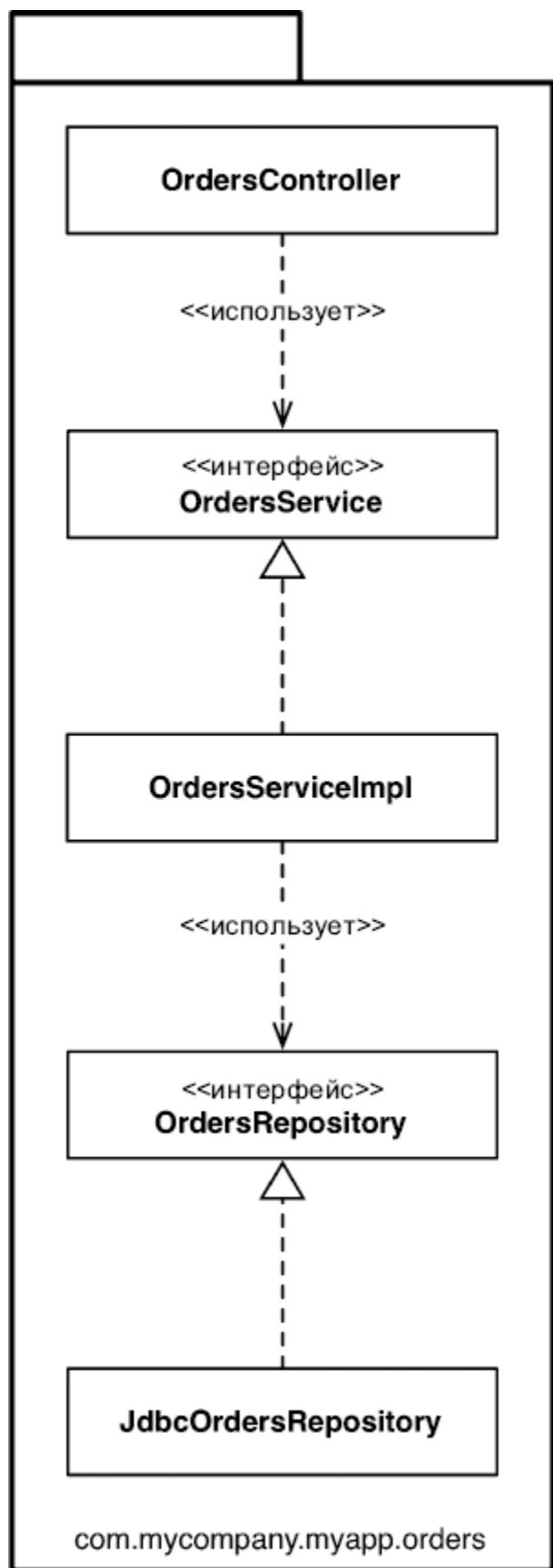


Рис. 34.2. Упаковка по особенностям

Мне часто встречаются команды разработчиков, испытывающие трудности с разделением на горизонтальные уровни («упаковка по уровням») и переключающиеся на разделение по вертикали («упаковка по особенностям»). По моему мнению, оба подхода не оптимальны. Дочитав книгу до этого места, многие из вас подумают, что можно сделать лучше, и они будут правы.

Порты и адаптеры

Как говорил «дядюшка Боб», подходы, такие как «порты и адаптеры», «гексагональная архитектура», «граница, управление, сущность» и др., придуманы с целью создания архитектур, в которых прикладной/предметный код независим и отделен от технических деталей реализации, таких как фреймворки и базы данных. Такие базы кода часто состоят из двух областей: «внутренняя» (предметная) и «внешняя» (инфраструктура), как показано на рис. 34.3.



Рис. 34.3. Кодовая база с внутренней и внешней областями

«Внутренняя» область включает все предметные понятия, а «внешняя» отвечает за взаимодействия с внешним миром (то есть содержит пользовательские интерфейсы, базы данных, механизмы интеграции со сторонними продуктами). Главное правило при такой организации: «внешняя» область зависит от «внутренней», но никогда наоборот. На рис. 34.4 изображена версия реализации случая использования «просмотр заказов».

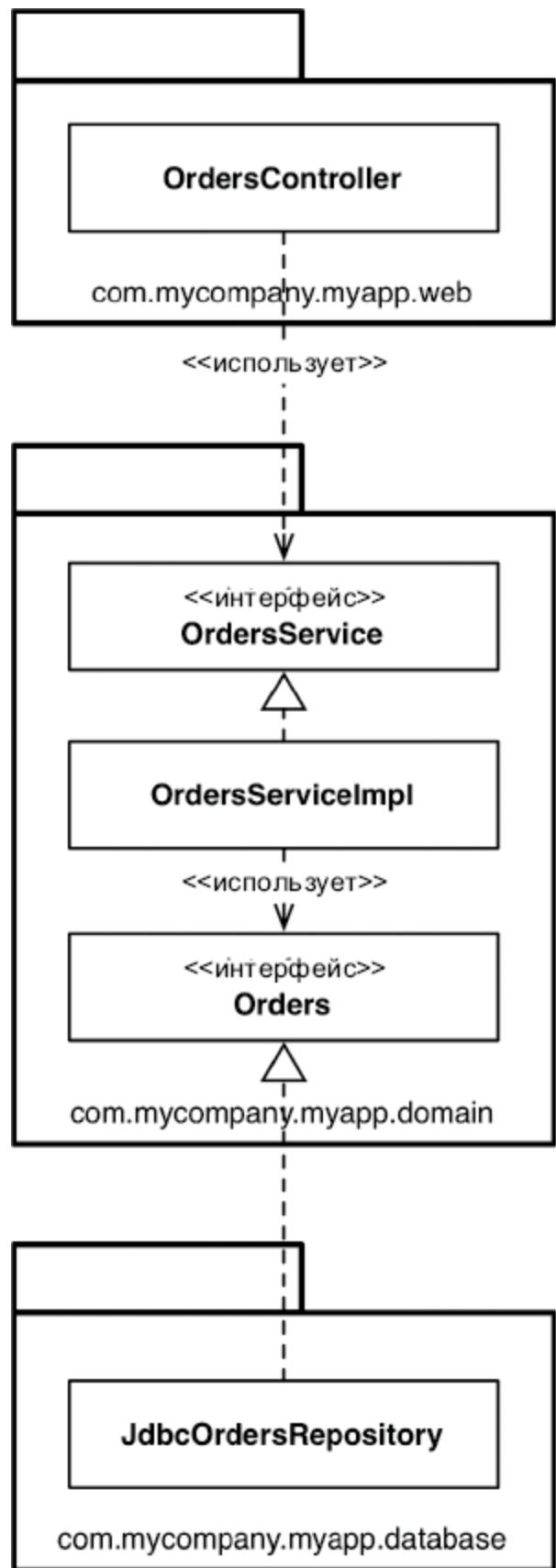


Рис. 34.4. Случай использования
«просмотр заказов»

Пакет `com.mycompany.myapp.domain` в этой версии — «внутренний», а другие пакеты — «внешние». Обратите внимание, что зависимости указывают в сторону «внутренней» области. Внимательный читатель наверняка заметит, что класс `OrdersRepository` переименован в `Orders`. Это объясняется влиянием правил предметно-ориентированного проектирования, которые требуют всему, что находится «внутри», давать простые имена из словаря «универсального предметного языка». Так, в предметных дискуссиях мы говорим «заказы», а не «хранилище заказов».

Также стоит отметить, что это существенно упрощенная версия UML-диаграммы классов, потому что в ней отсутствуют такие элементы, как интеракторы и объекты для передачи данных через границы зависимостей.

Упаковка по компонентам

В целом я согласен с рассуждениями о принципах SOLID, эквивалентности повторного использования (REP), согласованного изменения (CCP) и совместного повторного использования (CRP) и большинством советов в этой книге, но я пришел к немного другим выводам, касающимся организации кода. Поэтому я представляю еще один вариант, который я называю «упаковка по компонентам». Прежде всего, в своей карьере я много лет посвятил созданию корпоративного программного обеспечения, в основном на Java, во многих прикладных областях. Эти программные системы сильно отличались. Подавляющее их число были

основаны на Веб, но имелись также клиент-серверные⁶⁷, распределенные, основанные на обмене сообщениями и некоторые другие системы. Хотя технологии отличались, всех их объединяла традиционная многоуровневая архитектура.

Я уже упоминал пару причин, почему многоуровневую архитектуру следует считать неудовлетворительной, но это еще не все. Цель многоуровневой архитектуры — отделить код, выполняющий схожие функции. Веб-интерфейс отделяется от бизнес-логики, которая, в свою очередь, отделяется от механизмов доступа к данным. Как было показано на UML-диаграмме классов, с точки зрения реализации уровень примерно соответствует пакету. С точки зрения доступности кода, чтобы `OrdersController` мог зависеть от интерфейса `OrdersService`, последний должен быть объявлен общедоступным, потому что класс и интерфейс находятся в разных пакетах. Аналогично общедоступным должен быть объявлен интерфейс `OrdersRepository`, так как он используется классом `OrdersServiceImpl`, находящимся за пределами пакета, определяющего функции доступа к хранилищу.

В настоящей многоуровневой архитектуре стрелки зависимостей всегда должны быть направлены вниз. Уровни должны зависеть только от соседнего, нижележащего уровня. В результате введения некоторых правил, определяющих, как должны зависеть элементы в базе кода, получается красивый, чистый, ациклический граф зависимостей. Большая проблема в том, что мы можем хитрить, вводя некоторые нежелательные зависимости, и при этом получать замечательный ациклический граф зависимостей.

Предположим, что на работу был нанят новый специалист, он присоединился к вашей команде и вы поручаете ему реализовать другой вариант использования, связанный с

заказами. Как всякий новичок, этот человек хочет произвести впечатление и реализовать порученное ему задание максимально быстро. Посидев несколько минут с чашкой кофе, новичок замечает существующий класс `OrdersController` и решает, что это именно тот код, который должен использоваться новой веб-страницей, порученной ему. Но ему нужны некоторые данные о заказах из базы данных. Новичка озаряет: «О, здесь уже есть интерфейс `OrdersRepository`. Я могу просто внедрить реализацию в мой контроллер. Отлично!» Спустя несколько минут он создает действующую веб-страницу. Но получившаяся UML-диаграмма выглядит, как показано на рис. 34.5.

Стрелки зависимостей все еще направлены вниз, но теперь для некоторых вариантов использования `OrdersController` минует `OrdersService`. Такую организацию часто называют *нестрогой многоуровневой архитектурой*, так как уровням позволено перепрыгивать через смежные, соседние уровни. В некоторых ситуациях это делается намеренно, например, если вы пытаетесь следовать шаблону CQRS⁶⁸. Во многих других случаях нежелательно перепрыгивать через уровень бизнес-логики, особенно если эта бизнес-логика отвечает за авторизацию доступа к отдельным записям, например.

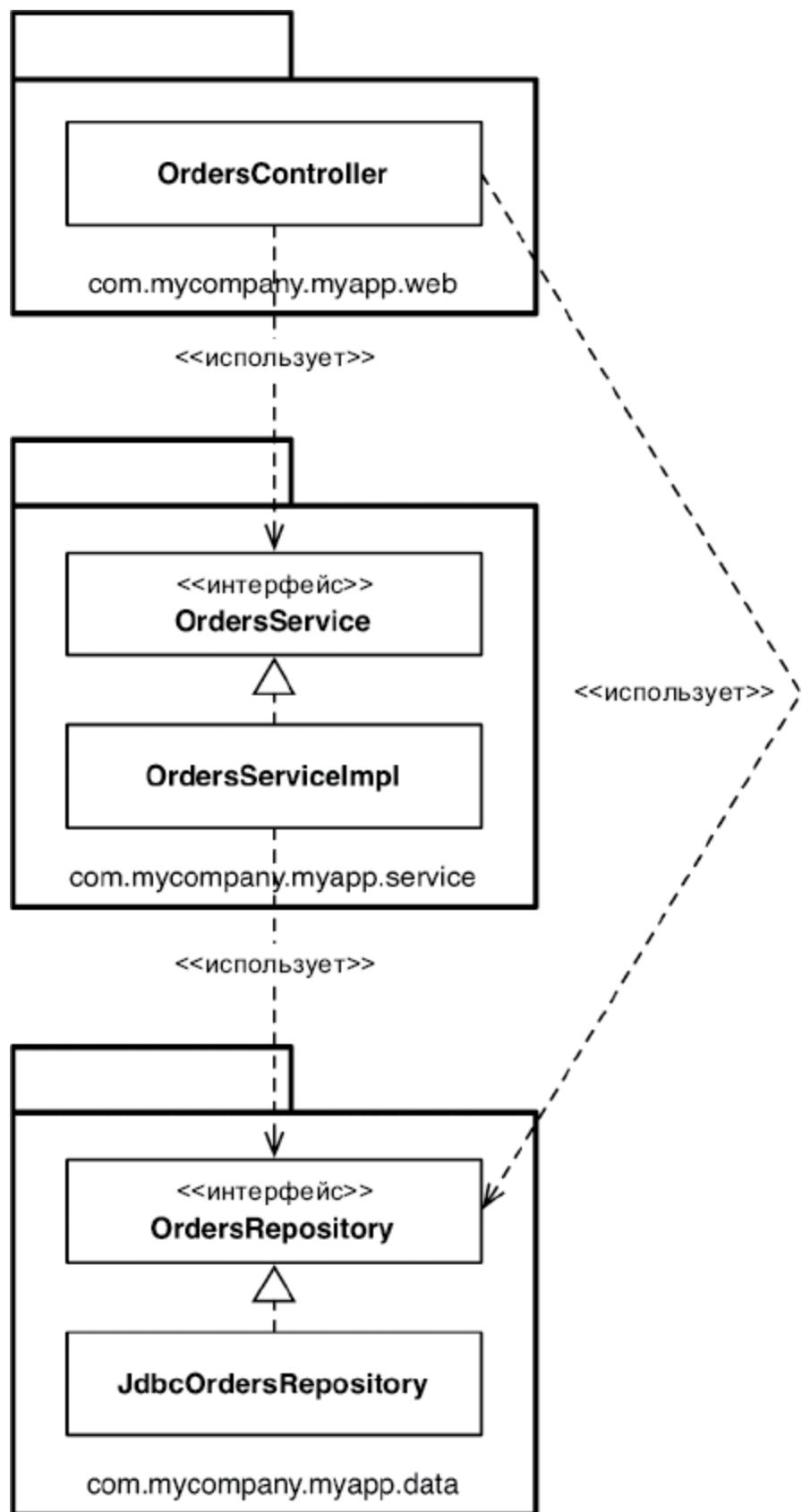


Рис. 34.5. Нестрогая многоуровневая архитектура

Новый вариант использования работает, но наверняка реализован не так, как вам хотелось бы. Нечто подобное я видел во многих командах, которые посещал как консультант, и обычно это проявляется, когда команды начинают выяснять, как выглядит база кода, часто в первый раз.

В такой ситуации необходимо установить правило — архитектурный принцип, — которое гласит, например: «Веб-контроллеры никогда не должны обращаться к хранилищу непосредственно». Проблема, конечно, заключается в исполнении правила. Многие команды, с которыми я встречался, заявляли: «Мы обеспечиваем соблюдение этого принципа строгой дисциплиной и обзорами кода, потому что доверяем нашим разработчикам». Это хорошо, что есть такая уверенность, но все мы знаем, что происходит, когда бюджет и сроки начинают приближаться к концу.

Намного меньше команд сообщали мне, что они используют инструменты статического анализа (например, NDepend, Structure101, Checkstyle) для автоматической проверки и выявления архитектурных нарушений во время сборки. Возможно, вы уже видели такие правила; обычно они имеют форму регулярных выражений или строк с шаблонными символами, которые указывают: «типы в пакете `**/web` не должны использоваться типами в `**/data`» и проверяются после этапа компиляции.

Это немного грубоватый подход, но он может помочь, сообщая о нарушениях архитектурных принципов, которые (по вашему мнению) должны помешать вашей команде разработчиков выполнить сборку. Проблема обоих подходов в том, что они чреваты ошибками и цикл обратной связи дольше, чем хотелось бы. Если отключить проверку, в результате код может превратиться в «большой ком грязи»⁶⁹.

Лично я хотел бы, чтобы за соблюдением архитектурных принципов следил компилятор.

Это ведет нас к варианту «упаковка по компонентам». Цель этого гибридного подхода, обсуждавшегося до сих пор, — упаковать все обязанности, связанные с одним крупным компонентом, в единый Java-пакет. Речь идет о сервис-ориентированном представлении программной системы, что, собственно, мы наблюдаем в архитектурах микрослужб. Подобно портам и адаптерам, интерпретирующими Веб как всего лишь еще один механизм доставки, методика «упаковка по компонентам» помогает отделить пользовательский интерфейс от этих крупных компонентов. На рис. 34.6 показано, как мог бы выглядеть вариант «просмотр заказов».

По сути, этот подход связывает «бизнес-логику» и код для работы с хранилищем в единое нечто, что мы называем «компонентом». Выше в книге дядюшка Боб дал такое определение компонента:

Компоненты — это единицы развертывания. Они представляют наименьшие сущности, которые можно развертывать в составе системы. В Java — это jar-файлы.

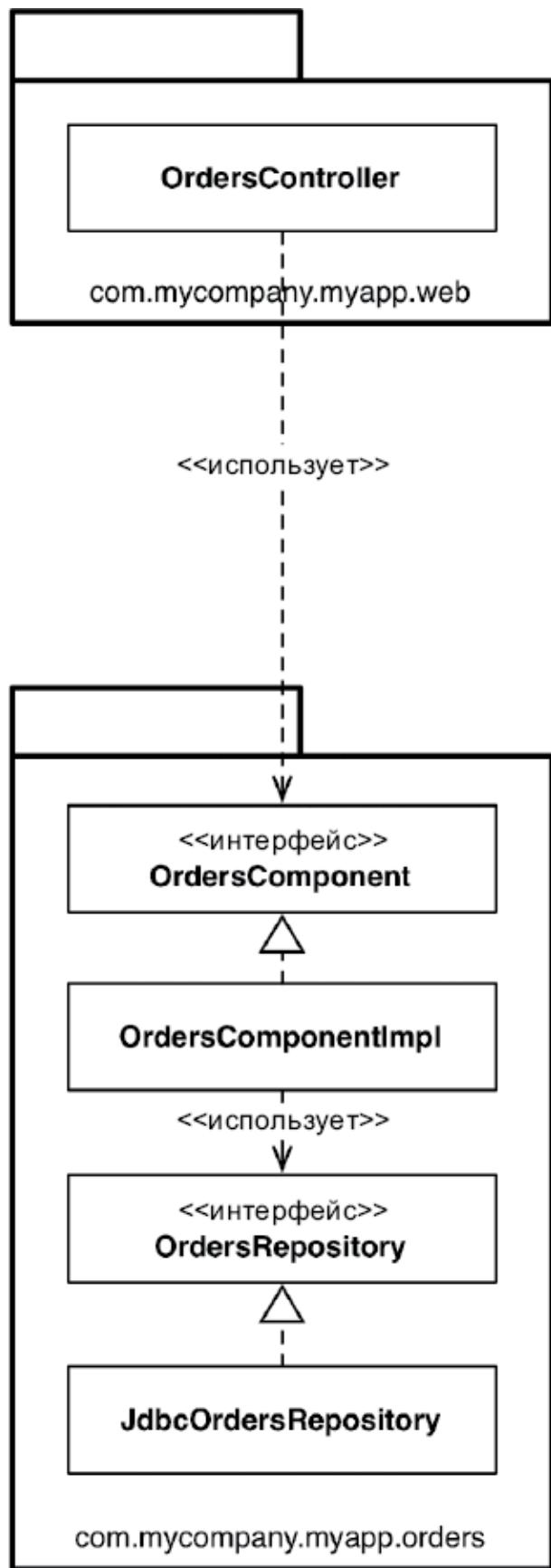


Рис. 34.6. Вариант использования «просмотр заказов»

Мое определение компонента немного отличается: «Группа функциональных возможностей, находящихся за общим чистым интерфейсом, которые постоянно находятся внутри среды выполнения, такой как приложение». Это определение взято из описания модели C4 программной архитектуры («C4 software architecture model»)⁷⁰, которая определяет простую иерархическую организацию статических структур программной системы в терминах контейнеров, компонентов и классов (или кода). В этом описании говорится, что программная система состоит из одного или нескольких контейнеров (например, веб-приложений, мобильных приложений, обычных приложений, баз данных, файловых систем), каждый из которых содержит один или несколько компонентов, которые, в свою очередь, реализуются одним или несколькими классами (или кодом). Находится ли каждый компонент в отдельном jar-файле — это уже второстепенный вопрос.

Ключевое преимущество подхода «упаковки по компонентам» заключается в размещении всего кода, например, имеющего отношение к обработке заказов, в одном месте — в компоненте `OrdersComponent`. Задачи внутри компонента все еще разделены, то есть бизнес-логика отделена от функций доступа к хранилищу, но это разделение является уже деталью реализации компонентов, о которой потребителям знать не обязательно. Это напоминает организацию микрослужб или сервис-ориентированную архитектуру, когда имеется отдельная служба `OrdersService`, инкапсулирующая все, что связано с обработкой заказов, отличаясь лишь режимом разделения.

Организацию монолитного приложения в виде набора тщательно проработанных компонентов можно рассматривать как шаг в направлении архитектуры микрослужб.

Дьявол в деталях реализации

На первый взгляд кажется, что все четыре подхода представляют собой разные способы организации кода и поэтому могут считаться разными архитектурными стилями. Это ощущение начинает быстро укрепляться, если нет понимания деталей реализации.

Я постоянно наблюдаю чересчур свободное использование модификатора доступа `public` в таких языках, как Java. Похоже, что разработчики используют ключевое слово `public` инстинктивно, не думая. Этот инстинкт хранится в нашей мышечной памяти. Если не верите, взгляните на примеры кода в книгах, руководствах и открытых фреймворках на сайте GitHub. Эта тенденция, похоже, никак не связана с архитектурным стилем, используемым для организации кода, будь то горизонтальные уровни, порты и адаптеры или что-то еще.

Объявление всех типов общедоступными означает отказ от возможностей инкапсуляции, предлагаемых языком программирования. Как результат, это открывает возможность любому написать код с реализацией конкретного класса, нарушающий используемый архитектурный стиль.

Организация и инкапсуляция

Взглянем на эту проблему с другой стороны. Если все типы в Java-приложении объявить общедоступными, пакеты превратятся в простой механизм организации (в группировки,

как папки), утратив свойства инкапсуляции. Так как общедоступные типы могут беспрепятственно использоваться в любой точке приложения, вы фактически можете игнорировать пакеты, потому что они практически не несут никакой ценности. В результате, если вы игнорируете пакеты (потому что они не имеют никаких средств инкапсуляции и сокрытия), становится совершенно неважно, какой архитектурный стиль вы пытаетесь воплотить. Если все типы объявить общедоступными, пакеты Java на UML-диаграммах, представленных выше, превращаются в ненужную деталь. По сути, при злоупотреблении подобными объявлениями все четыре архитектурных подхода, представленные выше в этой главе, становятся практически неотличимыми друг от друга (рис. 34.7).

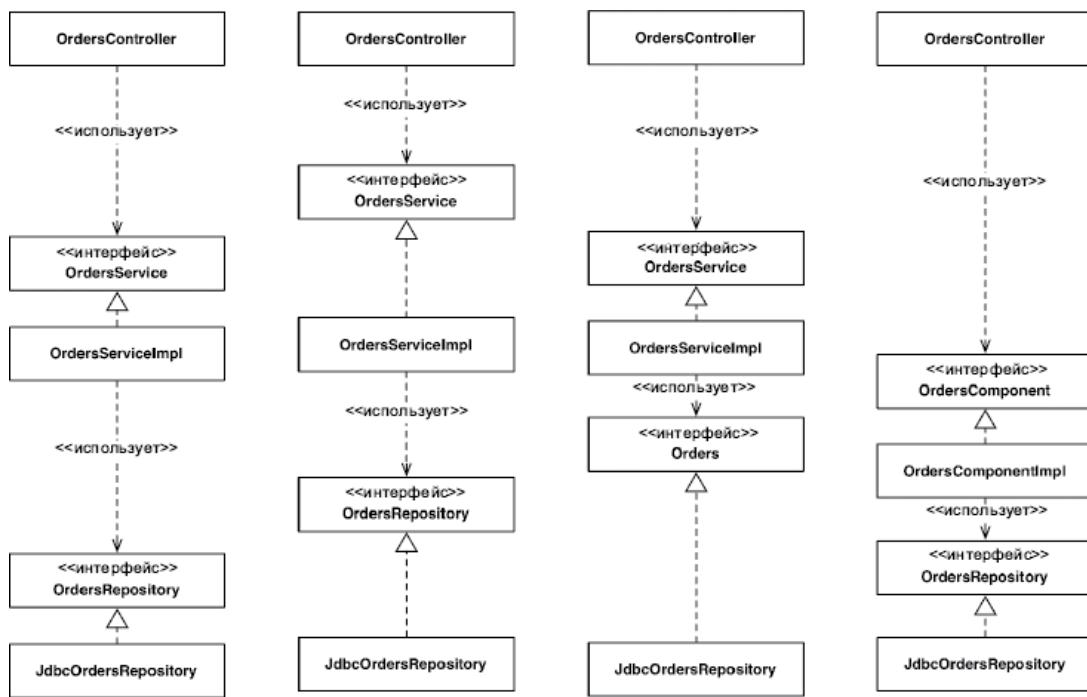


Рис. 34.7. Все четыре архитектурных стиля идентичны

Обратите внимание на стрелки между типами на рис. 34.7: они не зависят от архитектурного стиля, который вы пытаетесь воплотить. Концептуально стили очень разные, но

синтаксически они идентичны. Кроме того, можно даже утверждать, что после объявления всех типов общедоступными эти четыре стиля превращаются лишь в четыре способа описания традиционной архитектуры с горизонтальными уровнями. Это ловкий фокус, и, конечно же, никто не будет объявлять все свои Java-типы общедоступными. За исключением случаев, когда это действительно делается. И я видел их.

Модификаторы доступа в Java не идеальны⁷¹, но их игнорирование может вызывать проблемы. Порядок распределения типов Java по пакетам фактически может иметь большое значение для доступности (или недоступности) этих типов, когда модификаторы доступа применяются соответственно. Если вернуть пакеты и отметить те типы (сделав их бледнее на диаграмме), которым можно дать более ограничивающий модификатор, картина станет интереснее (рис. 34.8).

Двигаясь слева направо в подходе «упаковка по уровням», интерфейсы `OrdersService` и `OrdersRepository` должны быть объявлены общедоступными, потому что имеют входящие зависимости от классов, находящихся за пределами пакета, в котором объявлены эти интерфейсы. Классам с реализациями (`OrdersServiceImpl` и `JdbcOrdersRepository`), напротив, можно придать более ограниченную видимость (на уровне пакета). Никто не должен знать об их существовании; они являются деталями реализации.

В подходе «упаковка по особенностям» единственной точкой входа в пакет является `OrdersController`, поэтому доступ ко всему остальному можно ограничить рамками пакета. Важно отметить, что в такой ситуации никакой другой код, находящийся за пределами этого пакета, не сможет

получить никакой информации о заказах в обход контроллера. Иногда это может быть нежелательно.

В подходе с портами и адаптерами интерфейсы OrdersService и Orders имеют входящие зависимости из других пакетов, поэтому они должны быть объявлены общедоступными. И снова, доступность классов реализации

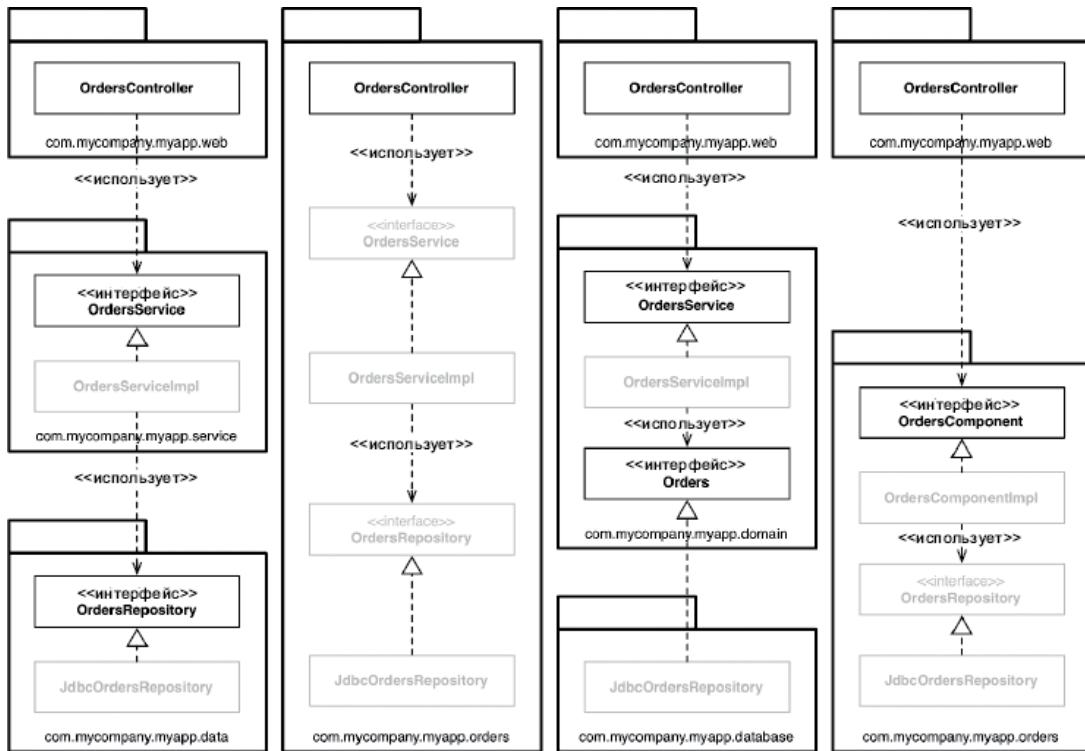


Рис. 34.8. Бледно-серым цветом выделены типы с более ограничивающим модификатором доступа

можно ограничить рамками пакета и внедрять зависимости во время выполнения.

Наконец, в подходе «упаковка по компонентам» интерфейс OrdersComponent имеет входящую зависимость от контроллера, но доступ ко всему остальному можно ограничить рамками пакета. Чем меньше общедоступных типов, тем меньше число потенциальных зависимостей. В данном случае код за пределами этого пакета не имеет

возможности⁷² напрямую использовать интерфейс `OrdersRepository` или его реализацию, поэтому соблюдение архитектурного принципа можно переложить на компилятор. То же самое можно проделать в .NET с помощью ключевого слова `internal`, но при этом придется создать отдельные сборки для всех компонентов.

Для большей ясности отмечу, что все, о чем рассказывалось здесь, относится к монолитному приложению, когда весь код находится в едином дереве исходных текстов. Если вы создаете такое приложение, я рекомендовал бы проводить в жизнь архитектурные принципы, опираясь на компилятор, а не полагаться на самодисциплину и инструменты, выполняющиеся после компиляции.

Другие режимы разделения

Кроме средств, поддерживаемых языком программирования, часто существуют другие способы разделения зависимостей в исходном коде. Для Java имеются свои инфраструктуры, такие как OSGi, и новейшая система модулей в Java 9. При правильном использовании системы модулей позволяют разделить типы, объявленные как `public`, и *публикуемые* (`published`) типы. Например, можно создать модуль `Orders`, в котором все типы объявлены как `public`, но опубликовать только ограниченное число этих типов для внешних потребителей. Это долгожданное нововведение, и я с энтузиазмом воспринимаю появление новой системы модулей Java 9, которая даст нам еще один инструмент для создания хорошего программного обеспечения и вновь разбудит в людях интерес к дизайнерскому мышлению.

Другая возможность разделить зависимости на уровне исходного кода — создать *несколько разных деревьев с исходным*

кодом. Для портов и адаптеров, например, можно было бы создать три таких дерева:

- Исходный код с предметной и бизнес-логикой (то есть все, что не зависит от выбора технологии и фреймворков): `OrdersService`, `OrdersServiceImpl` и `Orders`.
- Исходный код веб-интерфейса: `OrdersController`.
- Исходный код, реализующий хранение данных: `JdbcOrdersRepository`.

Последние два дерева имеют зависимости времени компиляции от кода с предметной и бизнес-логикой, который сам по себе ничего не знает о веб-интерфейсе и особенностях хранения данных. С точки зрения реализации это можно сделать, настроив отдельные модули или проекты в конфигурации инструмента сборки (например, Maven, Gradle, MSBuild). В идеале этот шаблон можно применить для отделения деревьев с исходным кодом всех компонентов приложения.

Однако это слишком идеалистическое решение, потому что такое разбиение исходного кода влечет за собой проблемы производительности, сложности и сопровождения.

Более простой подход, используемый теми, кто реализует архитектуру портов и адаптеров, заключается в создании двух деревьев с исходным кодом:

- Предметный код («внутренняя» область).
- Инфраструктурный код («внешняя» область).

Его хорошо иллюстрирует диаграмма (рис. 34.9), которую многие используют для обобщения архитектуры портов и адаптеров с ее зависимостью времени компиляции инфраструктурного кода от предметного.



Рис. 34.9. Предметный и инфраструктурный код

Такой подход к организации исходного кода тоже дает положительные результаты, но вы должны помнить о потенциальных компромиссах. Это то, что я называю «окружным антишаблоном портов и адаптеров». В Париже (Франция) имеется окружная автодорога с названием Boulevard Périphérique (бульвар Периферик), позволяющая обогнуть Париж, не утруждая себя сложностями движения внутри города. Включение всего инфраструктурного кода в единое

дерево подразумевает потенциальную возможность для кода из любой части приложения (например, веб-контроллера) напрямую вызывать код из другой части приложения (например, функции для работы с базой данных), без пересечения предметной области. Это особенно верно, если забыть применить в том коде соответствующие модификаторы доступа.

Заключение: недостающий совет

Основная цель этой главы — подчеркнуть, что любые, самые лучшие дизайнерские намерения можно уничтожить в мгновение ока, если не учитывать тонкости стратегии реализации. Подумайте, как ваш дизайн должен отображаться в структуру кода, как организовать этот код и какие режимы разделения применять во время выполнения и компиляции. Страйтесь оставлять открытыми любые возможности, но будьте прагматичными и учитывайте численность вашей команды, навыки ее членов, сложность решения и ограничения по времени и бюджету. Также подумайте, как использовать компилятор для принудительного соблюдения выбранного архитектурного стиля, и следите за связями в других областях, таких как модели данных. Дьявол кроется в деталях реализации.

[63](#) Это ужасное имя для класса, но, как будет показано ниже, в действительности это не имеет большого значения.

[64](https://martinfowler.com/bliki/PresentationDomainDataLayering.html) <https://martinfowler.com/bliki/PresentationDomainDataLayering.html>.

[65](#) Псевдоним «дядюшка Боб» (Uncle Bob) принадлежит автору этой книги Роберту Мартину. — Примеч. пер.

[66](#) Ценность этого преимущества нивелируется навигационными возможностями современных IDE, но, похоже, в последнее время наблюдается возрождение популярности легковесных текстовых редакторов, причины которого я, в силу возраста, не могу понять.

[67](#) На первой работе, после окончания университета в 1996 году, я занимался созданием клиент-серверных приложений для настольных компьютеров с использованием технологии PowerBuilder и суперпродуктивного языка 4GL, прекрасно подходящего для создания приложений баз данных. Спустя пару лет я создавал клиент-серверные приложения на Java, в которых мне приходилось создавать собственные подключения к базе данных (это было еще до появления JDBC) и наши собственные инструменты с графическим интерфейсом поверх AWT. Для вас это «прогресс»!

[68](#) В шаблоне CQRS (Command Query Responsibility Segregation — разделение ответственности команд и запросов) для чтения и изменения данных используются разные шаблоны.

[69](#) <http://www.laputan.org/mud/>.

[70](#) Дополнительные сведения можно найти по адресу <https://www.structurizr.com/help/c4>.

[71](#) Например, в Java невозможно ограничить доступ на основе отношений пакетов и подпакетов, хотя все мы привыкли считать пакеты иерархическими структурами. Любые создаваемые иерархии отражаются только на именах пакетов и структуре каталогов на диске.

[72](#) Можно, конечно, схитрить и воспользоваться механизмом рефлексии в Java, но не делайте так, пожалуйста!

VII. Приложение

Архитектурная археология



Чтобы определить принципы хорошей архитектуры, совершим путешествие по последним 45 годам и познакомимся с некоторыми проектами, над которыми я работал начиная с 1970 года. Некоторые из этих проектов

представляют определенный интерес с архитектурной точки зрения. Другие интересны извлеченными уроками, повлиявшими на последующие проекты.

Это приложение несколько автобиографично. Я старался придерживаться темы обсуждения архитектуры, но, как в любой автобиографической истории, иногда в нее вторгаются другие факторы. ;-)

Профсоюзная система учета

В конце 1960-х годов компания *ASC Tabulating* подписала контракт с местным профсоюзом водителей грузовиков на разработку системы учета. Компания ASC решила реализовать эту систему на машине GE Datanet 30, изображенной на рис. П.1.



Рис. П.1. GE Datanet 30 (фотографию предоставил Эд Телен (Ed Thelen), ed-thelen.org)

Как можно видеть на фотографии, это была огромная⁷³ машина. Она занимала целую комнату и требовала поддержания определенного микроклимата.

Эта ЭВМ была построена еще до появления интегральных микросхем. Она была собрана на дискретных транзисторах, и в ней имелось даже несколько радиоламп (хотя они использовались только в усилителях для управления ленточными накопителями).

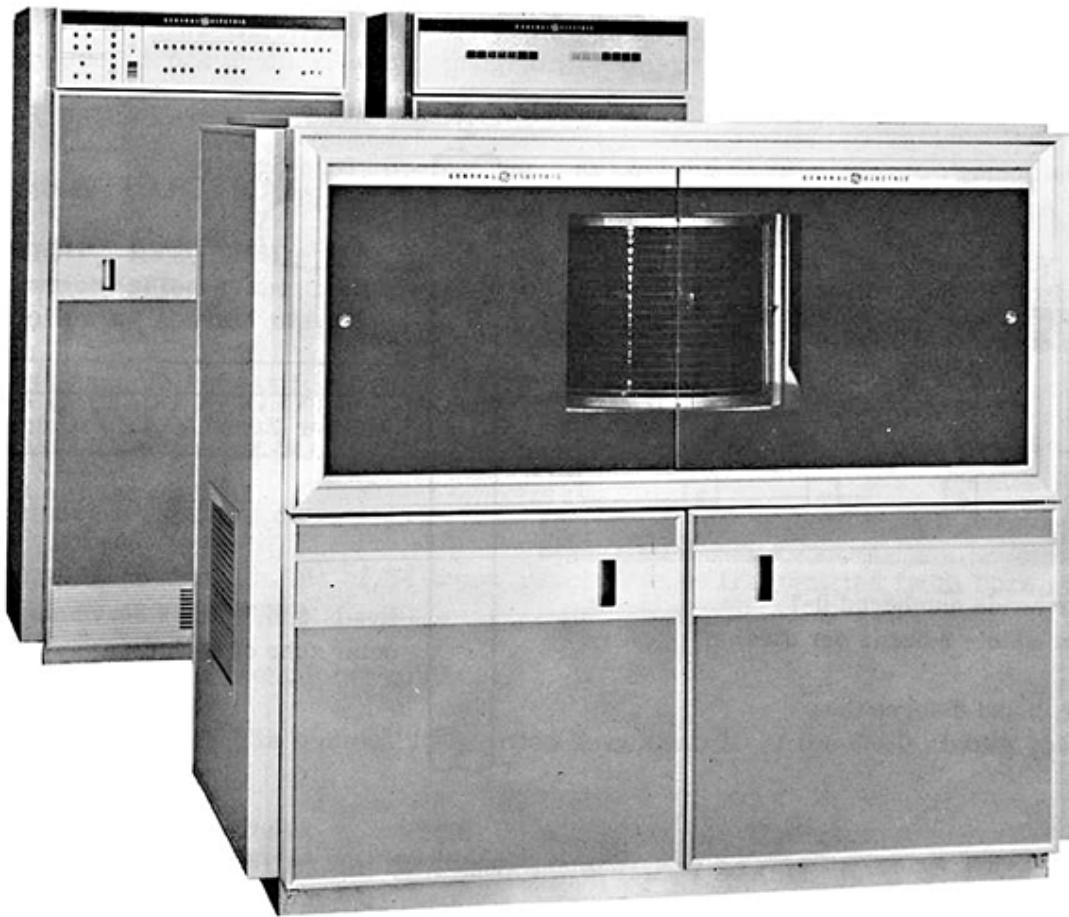
По сегодняшним меркам машина была огромная, медлительная и примитивная. В ней имелось $16 \text{ K} \times 18$ бит оперативной памяти с временем цикла порядка 7 микросекунд⁷⁴. Она занимала большую комнату с климатической установкой. Имела приводы для магнитной ленты с семью дорожками и жесткий диск емкостью около 20 Мбайт.

Этот диск был настоящим монстром. Вы можете увидеть его на рис. А.2, но эта фотография не позволяет оценить размеры чудовища. Он был выше моего роста. Пластины были 36 дюймов (примерно 91 сантиметр) в диаметре и 3/8 дюйма (9,5 миллиметра) в толщину. Одна из пластин изображена на рис. А.3.

Теперь подсчитайте пластины на первой фотографии. Их было более десятка. Для каждой имелась своя лапа с головкой, приводившаяся в движение пневматическим приводом. В процессе работы можно было видеть, как головки перемещаются поперек пластин. Время позиционирования головки колебалось от половины секунды до секунды.

Когда этот зверь включался, он рычал, как самолет. Пол ходил ходуном, пока тот набирал скорость⁷⁵.

ЭВМ Datanet 30 была знаменита своей возможностью асинхронно управлять большим количеством терминалов с относительно высокой скоростью. Именно это требовалось компании ASC.



MASS RANDOM ACCESS DATA STORAGE UNIT

Рис. П.2. Жесткий диск с пластинами (фотографию предоставил Эд Телен (Ed Thelen), ed-thelen.org)

Компания ASC находилась в Лейк Блафф, штат Иллинойс, в 30 милях к северу от Чикаго. Офис профсоюза размещался в центре Чикаго. Профсоюз нанял примерно десять человек для ввода данных в систему через терминалы CRT⁷⁶ (рис. П.4). Они также могли печатать отчеты на телетайпах ASR35 (рис. П.5).

Терминалы CRT поддерживали скорость обмена 30 символов в секунду. Это была хорошая скорость для конца 1960-х, потому что модемы в те дни были очень простенькими.

ASC арендовала у телефонной компании с десяток выделенных телефонных линий и вдвое больше 300-бодовых модемов для соединения Datanet 30 с этими терминалами.

В те времена компьютеры поставлялись без операционной системы. Они не имели даже файловых систем. У вас имелся только ассемблер.

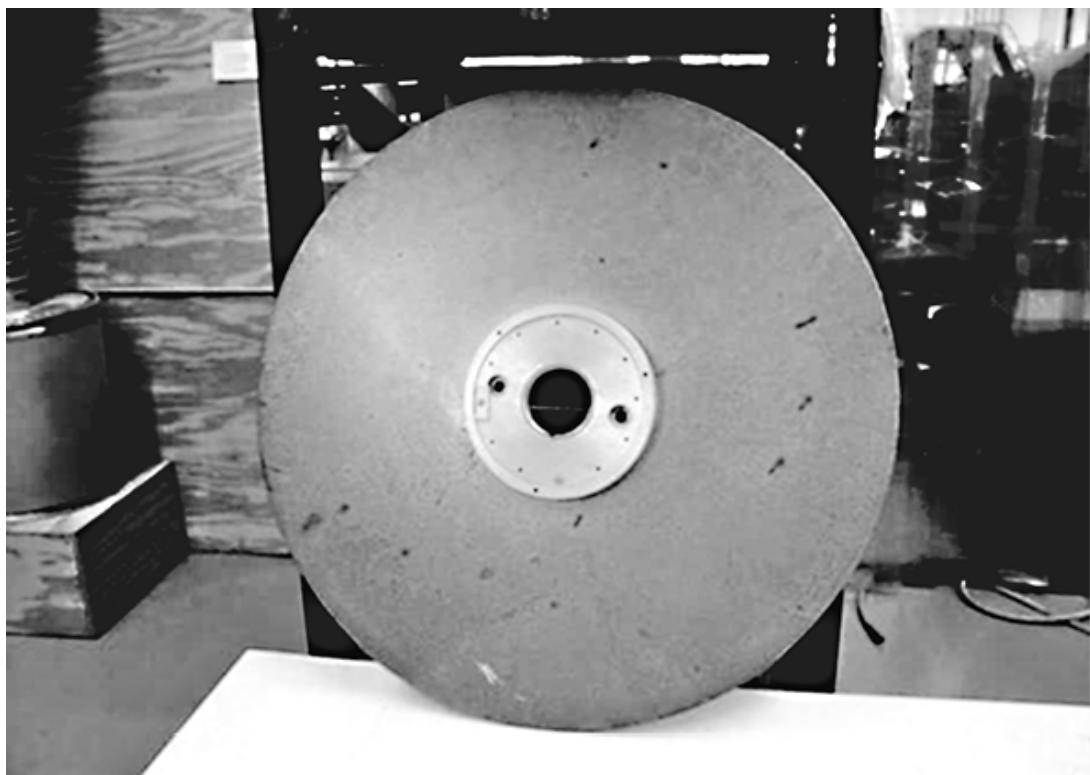


Рис. П.3. Одна из пластин диска: 3/8 дюйма толщиной, 36 дюймов в диаметре (фотографию предоставил Эд Телен (Ed Thelen), ed-thelen.org)



Рис. П.4. Терминал CRT (фотографию предоставил Эд Телен (Ed Thelen), ed-thelen.org)

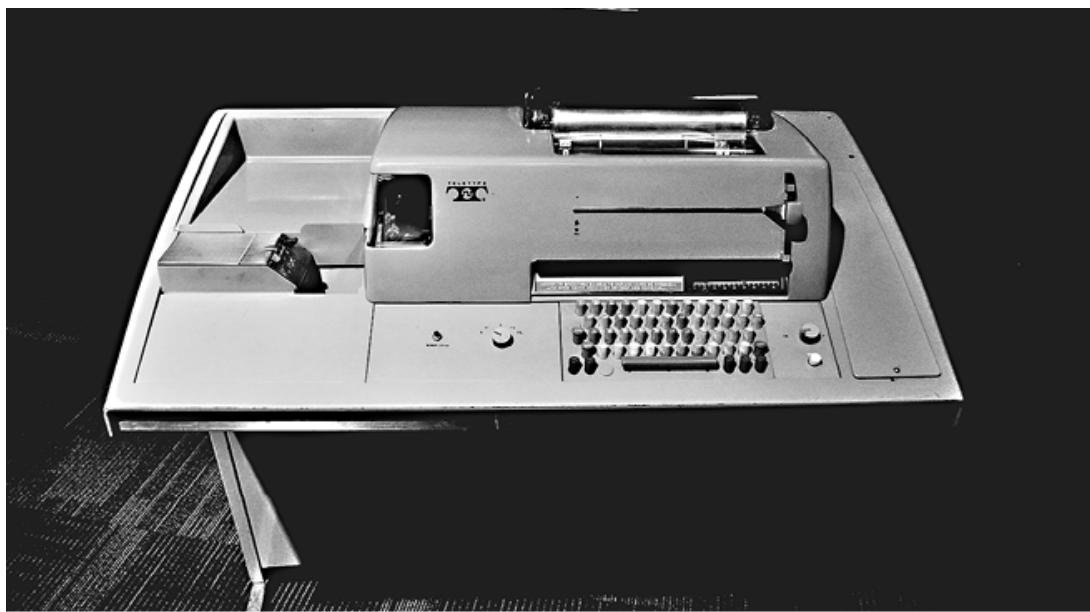


Рис. П.5. Телетайп ASR35 (с разрешения Джо Мейбла (Joe Mabel))

Если вам нужно было сохранить данные на диске, вы записывали их на диск. Не в файл. Не в каталог. Вы должны были определить дорожку, пластину и сектор для сохраняемых данных и затем управлять диском, чтобы записать туда данные. Да, это означало необходимость писать свой драйвер диска.

В системе учета имелось три вида записей с информацией об агентах, работодателях и членах профсоюза. Для этих записей поддерживались все четыре CRUD-операции⁷⁷, но кроме этого система включала операции для рассылки квитанций на уплату взносов, определения изменений в общем реестре и другие.

Первоначальная версия системы была написана на ассемблере консультантом, которому чудом удалось впихнуть ее в 16 К.

Как нетрудно догадаться, такая большая ЭВМ, как Datanet 30, была очень дорогой в обслуживании и эксплуатации. Услуги консультанта, поддерживавшего программное обеспечение, тоже обходились очень дорого. Более того, на рынке уже появились и стали набирать популярность более дешевые мини-компьютеры.

В 1971 году, когда мне было 18, компания ASC наняла меня и двух моих друзей-гиков, чтобы переписать систему учета для мини-компьютера Varian 620/f (рис. П.6). Компьютер стоил недорого. Наши услуги стоили недорого. Для ASC это было отличной сделкой.

Машина Varian имела 16-битную шину и 32 К × 16 оперативной памяти. Длительность цикла составляла примерно 1 микросекунду. Эта машина была намного мощнее, чем Datanet 30. В ней использовалась дико успешная дисковая технология 2314, разработанная в IBM, позволявшая хранить 30

мегабайт на пластинах, имевших всего 14 дюймов в диаметре, которые уже не могли пробивать бетонные стены!

Конечно, у нас все еще не было операционной системы. Не было файловой системы. Не было и высокоуровневого языка программирования. У нас имелся только ассемблер. Но мы справились с заданием.

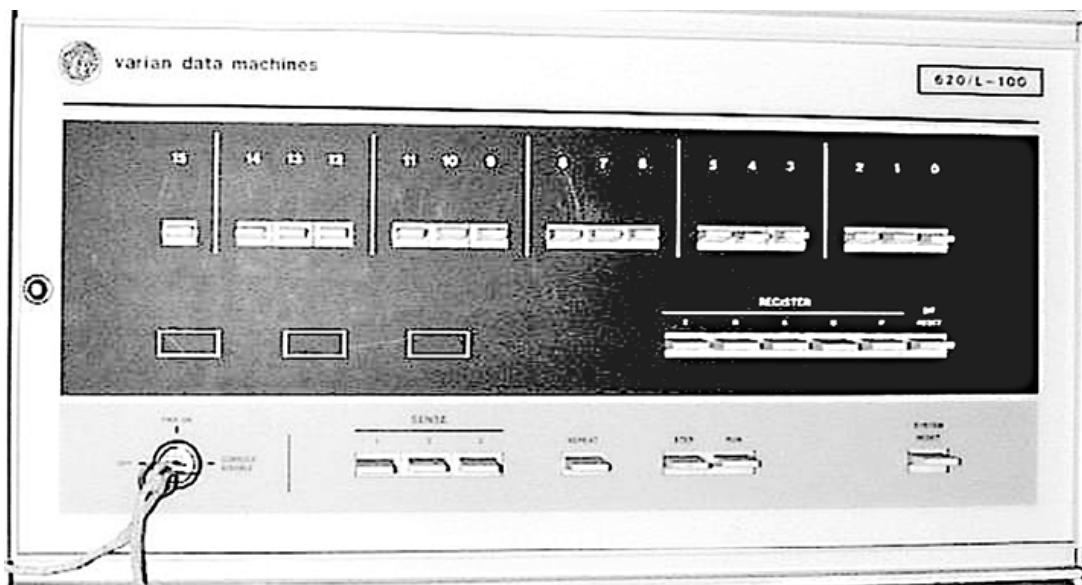


Рис. П.6. Мини-компьютер Varian 620/f (взято с сайта The Minicomputer Orphanage)

Вместо попытки втиснуть всю систему в 32 К, мы создали систему оверлеев. Приложения могли загружаться с диска в блок памяти, выделенной для оверлеев. Они могли выполняться в этой памяти и вытесняться со своими данными в памяти обратно на диск, чтобы дать возможность поработать другим программам.

Программы могли загружаться в область оверлеев, выполняться ровно столько, сколько необходимо для заполнения выходных буферов, и затем выгружаться на диск, чтобы освободить память для следующей программы.

Конечно, когда пользовательский интерфейс работает со скоростью 30 символов в секунду, программы тратят массу

времени на ожидание. У нас в запасе оставалось достаточно времени, чтобы программы могли загружаться и записываться на диск, обеспечивая максимальную скорость обмена с терминалами. Никто и никогда не жаловался на проблемы с временем отклика.

Мы написали вытесняющего диспетчера задач, управляющего прерываниями и вводом/выводом. Мы написали приложения; мы написали драйверы диска и драйверы терминалов, драйверы накопителей на магнитной ленте и все остальное в этой системе. В этой системе не было ни одного бита, написанного не нами. Это был тяжелый труд в течение множества 80-часовых недель, но мы запустили этого зверя за 8 или 9 месяцев.

Система имела простую архитектуру (рис. П.7). Когда приложение запускалось, оно генерировало данные до заполнения выходного буфера заданного терминала. Затем диспетчер задач выгружал это приложение и загружал новое. При этом диспетчер продолжал выводить информацию со скоростью 30 символов в секунду почти до его опустошения. Затем он вновь загружал приложение, чтобы снова заполнить буфер.

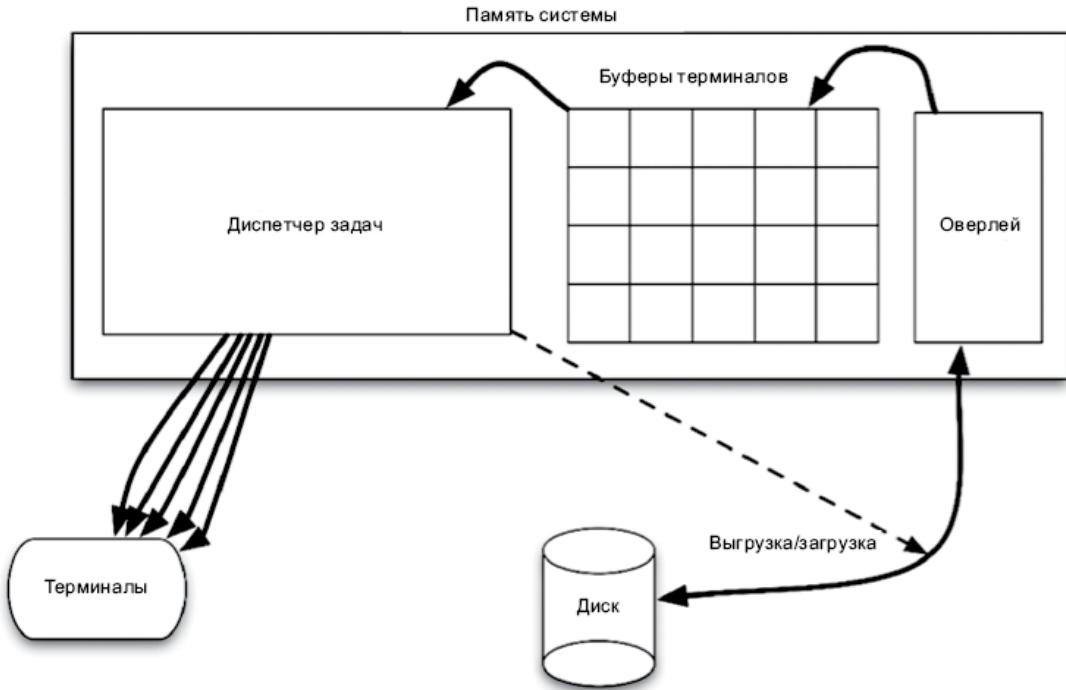


Рис. П.7. Архитектура системы

В этой системе есть две границы. Первая — вывод символов. Приложения не знали, что их вывод посыпается терминалам со скоростью 30 символов в секунду. В действительности для приложений вывод символов был полностью абстрагирован. Приложения просто передавали строки диспетчеру задач, а тот заботился о загрузке их в буфера, отправке символов терминалам и загрузке приложений в память и выгрузке их из памяти.

Зависимости пересекали эту границу в прямом направлении, то есть их направленность совпадала с направленностью потока управления. Приложения имели зависимости времени компиляции от диспетчера задач, и поток управления следовал от приложений в сторону диспетчера. Граница оберегает приложения от просачивания в них информации о типе устройства, в который производится вывод.

Вторая граница пересекалась зависимостями в обратном направлении. Диспетчер задач мог запускать приложения, но не имел зависимости времени компиляции от них. Поток управления следовал от диспетчера к приложениям. Полиморфный интерфейс, инвертирующий зависимость, был прост: каждое приложение запускалось переходом по одному и тому же адресу в области оверлея. Граница оберегает диспетчера от просачивания в него информации об устройстве приложений, кроме адреса точки запуска.

Laser Trim

В 1973 году я поступил на работу в компанию *Teradyne Applied Systems* (TAS) в Чикаго. Это было подразделение корпорации *Teradyne Inc.* со штаб-квартирой в Бостоне. Мы занимались системой, управлявшей довольно мощными лазерами для обработки электронных компонентов с высокой точностью.

В ту пору производители электронных компонентов использовали метод шелкотрафаретной печати на керамической подложке. Подложки имели размер примерно 1 квадратный дюйм. Компонентами были обычные резисторы — устройства, создающие сопротивление электрическому току.

Величина сопротивления резистора зависела от множества факторов, включая состав и геометрию. Чем шире был резистор, тем меньшее сопротивление он оказывал.

Наша система позиционировала керамическую подложку в жгуте проводов, соединяющих датчики с резисторами. Система должна была измерять сопротивление и затем с помощью лазера отсекать части резистора, делая его тоньше и тоньше, пока не будет достигнута желаемая величина сопротивления с точностью до десяти процентов.

Мы продавали эти системы производителям, а также использовали некоторые свои системы для производства небольших партий по заказам менее крупных производителей.

Система работала на компьютере M365. Это было время, когда многие компании строили свои компьютеры: корпорация Teradyne строила M365 и передавала их своим подразделениям. M365 был усовершенствованной версией PDP-8 — популярного в те дни мини-компьютера.

M365 управлял координатным столом, который перемещал керамические подложки между датчиками. Также он управлял системой измерения и лазером. Позиционирование лазера осуществлялось посредством вращающихся X-Y зеркал. Компьютер также управлял мощностью лазера.

Среда разработки для M365 была довольно примитивной. Этот компьютер не имел диска. Данные сохранялись на картриджах с магнитной лентой, которые выглядели как старые 8-дорожечные аудиокассеты. Лента и приводы производились компанией Tri-Data.

Так же как 8-дорожечные аудиокассеты того времени, лента была склеена в петлю. Привод мог прокручивать ее только в одном направлении — в них не было функции перемотки! Если требовалось установить ленту в начальную позицию, нужно было прокручивать ее вперед до достижения «точки загрузки».

Лента прокручивалась со скоростью примерно 1 фут в секунду (примерно 30 сантиметров в секунду). То есть петля из ленты длиной 25 футов (чуть больше 7,5 метра) перематывалась до точки загрузки самое большее за 25 секунд. По этой причине Tri-Data выпускала картриджи с лентой разной длины, от 10 до 100 футов (примерно от 3 до 30 метров).

На передней панели M365 имелась кнопка, по нажатии которой производилась загрузка начальной программы в память и ее запуск. Эта программа могла прочитать первый

блок с ленты и запустить его. Обычно в этом блоке хранился загрузчик, загружавший операционную систему, хранящуюся в остальных блоках на ленте.

Операционная система запрашивала у пользователя имя программы для запуска. Эти программы хранились на ленте, сразу вслед за операционной системой. Пользователь мог ввести имя программы — например, ED-402 Editor, — а операционная система отыскивала ее на ленте, загружала и запускала.

Консолью служил ASCII CRT терминал с зеленым свечением, шириной 72 символа⁷⁸ и высотой 24 строки. Он мог отображать только символы верхнего регистра.

Чтобы отредактировать программу, нужно было загрузить редактор ED-402 Editor и затем вставить ленту с исходным кодом. Редактор позволял прочитать с ленты в память один блок с исходным кодом и вывести его на экран. В одном блоке можно было сохранить до 50 строк кода. Для внесения изменений нужно было переместить курсор в требуемую строку и ввести текст, примерно так, как это делается в редакторе vi. По завершении требовалось записать блок на другую ленту и прочитать следующий блок с исходной ленты. Вы должны были продолжать эти манипуляции, пока не закончите.

Не было никакой возможности прокрутить блоки в обратном направлении. Правка программы производилась линейно, от начала до конца. Чтобы вернуться в начало, нужно было закончить копирование исходного кода на выходную ленту и затем начать новый сеанс редактирования уже с этой лентой в качестве исходной. Неудивительно, что при таких ограничениях мы сначала писали свои программы на бумаге, вносили все правки вручную красным карандашом и только

потом правили программу блок за блоком, сверяясь с пометками в листинге на бумаге.

Закончив правку программы, мы возвращались в операционную систему и вызывали ассемблер. Ассемблер читал код с исходной ленты и записывал двоичный код на другую ленту, при этом выводил листинг на наш последовательный принтер.

Ленты не были на 100% надежными, поэтому мы выполняли запись сразу на две ленты. Это увеличивало вероятность, что хотя бы одна из них не будет содержать ошибок.

Наша программа состояла примерно из 20 000 строк кода, а ее компиляция занимала примерно 30 минут. Шансы получить в это время ошибку чтения с ленты были 1 : 10. Если ассемблер сталкивался с такой ошибкой, он издавал сигнал на консоли и затем начинал выводить поток ошибок на принтер. Этот сигнал можно было услышать во всей лаборатории. Также можно было услышать проклятия несчастного программиста, только что узнавшего, что ему придется вновь запустить 30-минутный процесс компиляции.

Программа имела типичную для тех дней архитектуру. У нас имелась главная операционная программа (*Master Operating Program*), которую мы называли «the MOP». Она осуществляла управление базовыми функциями ввода/вывода и предоставлялаrudиментарную «командную оболочку» для консоли. Многие подразделения Teradyne использовали общий исходный код MOP, адаптируя его под собственные нужды. Как следствие, мы пересыпали друг другу изменения в исходном коде в форме бумажных листингов с поправками, которые затем (очень тщательно) вносили вручную.

Измеряющим оборудованием, координатными столами и лазером управляла специальная утилита. Граница между этим уровнем и MOP была запутанной. Уровень утилиты часто

вызывал уровень MOP, а специально модифицированная версия MOP часто вызывала утилиту. В действительности мы не разделяли их на два уровня. Для нас это был лишь некоторый код, добавляемый нами в MOP и создающий тесные связи.

Затем появился уровень изоляции. Этот уровень поддерживал интерфейс виртуальной машины для прикладных программ, которые писались на совершенно другом, предметно-ориентированном языке (Domain-Specific Language; DSL). Язык включал операции перемещения лазера и координатного стола, выполнения реза и измерений и т.д. Наши клиенты писали на этом языке свои программы управления лазером, а изолирующий уровень выполнял их.

Этот подход не предусматривал создания машинно-независимого языка программирования для управления лазером. В действительности язык имел множество связей с уровнями, лежащими ниже. Скорее этот подход дал прикладным программистам более «простой» язык, чем ассемблер M365, на котором они описывали свои задания для лазерной обработки.

Система могла загружать такие программы с ленты и выполнять их. По сути, наша система была операционной системой для приложений лазерной обработки.

Система была написана на ассемблере M365 и компилировалась в единственную единицу компиляции с абсолютным двоичным кодом.

Границы в этом приложении в лучшем случае были мягкими. Даже граница между системным кодом и приложениями на DSL не имела строгой силы. Повсюду были связи.

Но это было типично для программного обеспечения начала 1970-х.

Контроль алюминиевого литья под давлением

В середине 1970-х годов, когда организация ОПЕК вводила эмбарго на поставки нефти, а дефицит бензина провоцировал на заправочных станциях драки между озлобленными водителями, я поступил на работу в *Outboard Marine Corporation* (ОМС). Это была головная компания, объединившая *Johnson Motors* и *Lawnboy Lawntowers*.

ОМС имела большое производство в Уокигане, штат Иллинойс, по производству литых алюминиевых деталей для всех видов двигателей и изделий, изготавливаемых компанией. Алюминий расплавлялся в огромных печах и затем в ковшах перевозился к десяткам и десяткам машин для литья. Каждой машиной управлял человек-оператор, отвечавший за установку изложниц, выполнение цикла литья и извлечение готовых деталей. Зарплата операторов зависела от количества отлитых деталей.

Меня наняли для разработки проекта по автоматизации цеха. ОМС приобрела ЭВМ IBM System/7, которая была ответом IBM на появление мини-компьютеров. Они связали этот компьютер со всеми машинами для литья в цехе, чтобы мы могли подсчитать время работы и количество циклов каждой машины. Наша задача заключалась в том, чтобы собрать всю эту информацию и вывести ее на зеленые экраны терминалов 3270.

Программы для этой машины писались на языке ассемблера. И снова весь код, выполнявшийся на этом компьютере, был написан нами до последнего бита. У нас не было ни операционной системы, ни библиотек подпрограмм, ни фреймворков. Это был просто код.

Причем это был код, управляемый прерываниями и действующий в режиме реального времени. Каждый раз, когда какая-то машина завершала цикл, мы обновляли пакет

статистических данных и посыпали сообщение большой ЭВМ IBM 370, где работала программа на CICS-COBOL, выводившая эти данные на зеленые экраны.

Я ненавидел эту работу. Боже, как я ее ненавидел. Нет, сама *работа* была интересной! Но культура... Достаточно сказать, что я был обязан носить галстук.

Я старался. Я очень старался. Но я был очень недоволен этой работой, и мои коллеги знали об этом. Они понимали это, потому что я забывал важные даты или просыпал в дни, когда надо было рано вставать, чтобы прийти на важное совещание. Это была единственная работа, связанная с программированием, с которой меня уволили, — и поделом.

С архитектурной точки зрения в этой системе нет ничего поучительного, кроме одного. ЭВМ System/7 имела очень интересную инструкцию *установки программного прерывания SPI* (Set Program Interrupt). Она позволяла вызывать прерывание процессора, чтобы обработать любые другие низкоприоритетные прерывания, стоящие в очереди. В современном языке Java имеется схожий аналог, который называется `Thread.yield()`.

4-TEL

В октябре 1976 года, после увольнения из ОМС, я вернулся в *Teradyne*, но в другое подразделение, где я проработал следующие 12 лет. Там я занимался проектом под названием 4-TEL. Его целью было еженощное тестирование всех телефонных линий, обслуживаемых компанией, и создание отчета с перечислением всех линий, требующих ремонта. Это позволяло сосредоточить внимание обслуживающего персонала на конкретных телефонных линиях.

Эта система начинала свой путь с той же архитектуры, что и система Laser Trim. Это было монолитное приложение, написанное на языке ассемблера и не имевшее каких-то значительных границ. Но в то время, когда я вернулся в компанию, все должно было измениться.

Система использовалась сотрудниками сервисного центра (Service Center; SC). Сервисный центр охватывал несколько телефонных станций (Central Offices; CO), каждый из которых мог обслуживать до 10 000 телефонных линий. Аппаратура переключения линий и измерения уровня сигнала должна была размещаться в телефонных станциях (CO). Поэтому мы установили там компьютеры M365. Мы называли их тестерами линий в телефонных станциях (Central Office Line Testers; COLTs). Еще один M365 находился в сервисном центре (SC); он назывался компьютером зоны обслуживания (Service Area Computer; SAC). К компьютеру SAC было подключено несколько модемов для обмена данными с несколькими компьютерами COLT на скорости 300 бод (30 символов в секунду).

Сначала всю работу выполняли компьютеры COLT, включая все взаимодействия с консолями, обслуживание меню и составление отчетов. Компьютер SAC играл роль простого мультиплексора, получавшего данные от компьютеров COLT и выводившего их на экран.

Проблема такой организации состояла в том, что скорость 30 символов в секунду действительно была слишком маленькой. Работникам не очень нравилось наблюдать, как появляются символы на экране, особенно если учесть, что их интересовал небольшой объем ключевых данных. Кроме того, в те дни оперативная память в M365 стоила очень дорого, а программа была большой.

Поэтому было решено отделить часть программы, осуществляющую тестирование линий, от части,

анализирующей результаты и печатающей отчеты. Последнюю предполагалось перенести на компьютер SAC, а первая должна была продолжать работать на компьютерах COLT. Это должно было позволить использовать в качестве COLT машины поменьше, с меньшим объемом памяти, и значительно повысить скорость вывода информации на терминал, потому что отчеты должны были генерироваться на компьютере SAC.

Результат превзошел самые смелые ожидания. Информация на экране обновлялась очень быстро (после соединения с соответствующим компьютером COLT), а объем памяти в компьютерах COLT значительно уменьшился.

Граница получилась очень четкой и надежной. Компьютеры SAC и COLT обменивались очень короткими пакетами. Эти пакеты были очень простой формой предметно-ориентированного языка с такими командами, как «DIAL XXXX» или «MEASURE».

Загрузка M365 осуществлялась с магнитной ленты. Накопители на магнитной ленте были дорогими и не очень надежными, особенно в промышленном окружении телефонных станций. Кроме того, сама машина M365 стоила довольно дорого в сравнении с остальной электроникой в компьютерах COLT. Поэтому мы приступили к реализации проекта по замене M365 микрокомпьютером на базе микропроцессора 8085.

Новый компьютер состоял из процессорной платы с микропроцессором 8085, платы ОЗУ с 32 Кбайт памяти и трех плат с ПЗУ, содержащих 12 Кбайт памяти, доступной только для чтения. Все эти платы помещались в шасси с измерительным оборудованием, благодаря чему можно было убрать громоздкое шасси, в котором размещалась ЭВМ M365.

Платы ПЗУ содержали по 12 микросхем Intel 2708 EPROM (Erasable Programmable Read-Only Memory — стираемое

программируемое постоянное запоминающее устройство, СППЗУ)^{[79](#)}. На рис. П.8 показано, как выглядела такая микросхема. Мы записывали программы в эти микросхемы, вставляя их в специальное устройство, которое называлось программатором ППЗУ и управлялось нашей средой разработки. Информацию на микросхемах можно было стирать, подвергая их облучению ультрафиолетовым светом большой интенсивности^{[80](#)}.

Мой друг и я занялись переводом программ для COLT с языка ассемблера M365 на язык ассемблера 8085. Перевод выполнялся вручную и занял почти 6 месяцев. В результате получилось около 30 Кбайт кода 8085.

Наша среда разработки имела 64 Кбайт ОЗУ и не имела ПЗУ, поэтому скомпилированный двоичный код мы могли быстро загрузить в ОЗУ и протестировать.

Получив работоспособную программу, мы переключались на использование СППЗУ (EPROM). Мы программировали 30 микросхем и вставляли их в соответствующие гнезда в трех платах ПЗУ. Каждая микросхема подписывалась, поэтому мы точно знали, какую из них в какое гнездо нужно вставить.

30 Кбайт программного кода — это был единый двоичный блок длиной 30 Кбайт. Чтобы записать этот код в микросхемы ПЗУ, мы просто делили двоичный образ на 30 сегментов по 1 Кбайт и записывали каждый сегмент в микросхему с соответствующей надписью.

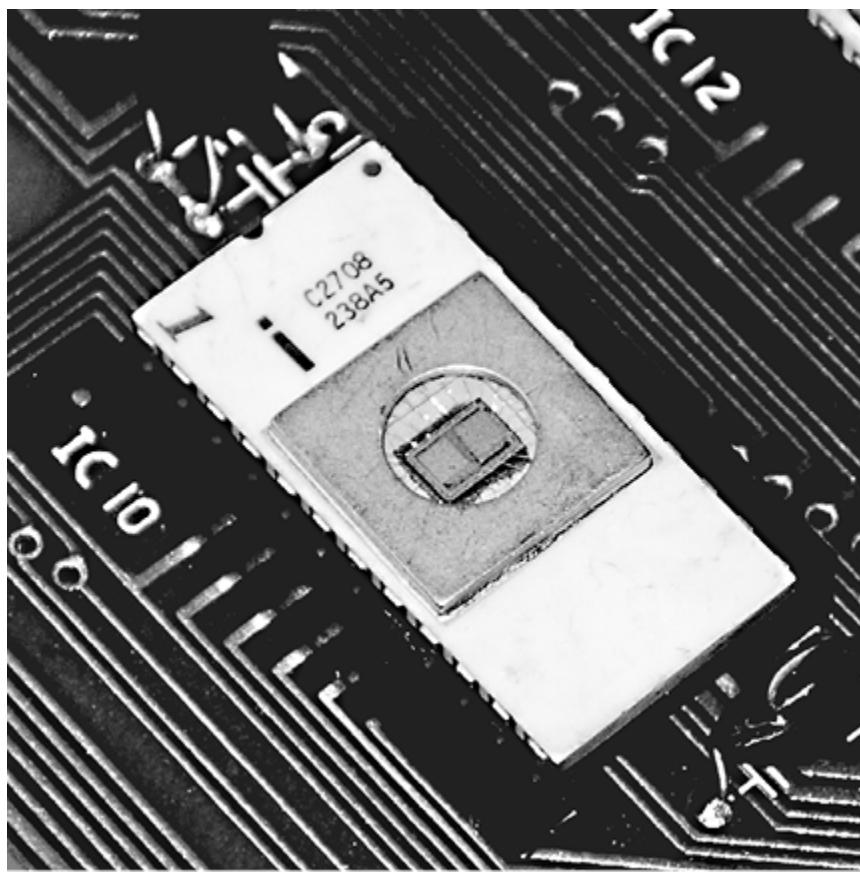


Рис. П.8. Микросхема СППЗУ (EPROM)

Эта схема прекрасно работала, и мы начали массовое производство оборудования и развертывание системы в поле.

Но программное обеспечение в первую очередь является программным⁸¹. Требовалось добавлять новые возможности, исправлять ошибки. А так как базовая система разрасталась, логистика обновления программного обеспечения путем программирования 30 микросхем на каждый экземпляр и замены всех 30 микросхем в каждом офисе превращалась в кошмар.

Возможны были все виды проблем. Иногда микросхемы подписывались неправильно или наклейки с подписями отваливались. Иногда инженер службы эксплуатации мог по ошибке заменить не ту микросхему или поломать один из

выводов новой микросхемы. Как следствие, инженерам приходилось носить с собой все 30 микросхем.

Зачем менять все 30 микросхем? Каждый раз, когда добавлялся или удалялся код из 30-килобайтного блока выполняемого кода, изменялись адреса всех машинных инструкций. Изменялись также адреса подпрограмм и функций. То есть затрагивалась каждая микросхема, каким бы простым ни было изменение.

Однажды ко мне зашел мой начальник и попросил решить эту проблему. Он сказал, что нужно найти какой-то способ изменения микропрограммы без замены всех 30 микросхем ПЗУ. Мы в коллективе обсудили эту проблему и приступили к проекту «Векторизация». На его реализацию ушло 3 месяца.

Идея была до смешного простой. Мы разбили 30 Кбайт программного кода на 32 файла с исходным кодом, компилирующихся независимо в блоки меньше одного 1 Кбайт. В начало каждого файла с исходным кодом мы вставили инструкцию, сообщающую компилятору, в какой адрес должен компилироваться данный код (например, ORG C400 для микросхемы ПЗУ, вставляемой в гнездо C4).

Также в начало каждого файла с исходным кодом мы добавили структуру фиксированного размера с адресами всех подпрограмм в этом блоке (вектор переходов). Эта структура имела размер 40 байт, поэтому могла хранить до 20 адресов. Это означало, что блок для одной микросхемы не мог содержать более 20 подпрограмм.

Затем мы создали особую область в ОЗУ, которую называли массивом векторов. Она содержала 32 таблицы по 40 байт — достаточный объем для хранения указателей на начало каждого блока в отдельных микросхемах.

Наконец, мы заменили вызовы подпрограмм в каждом блоке косвенными вызовами через соответствующий вектор в

ОЗУ.

Когда происходила загрузка программы, в массив векторов в ОЗУ загружались векторы переходов из всех микросхем ПЗУ, а затем осуществлялся переход в точку запуска главной программы.

У нас все получилось. Теперь, когда исправлялась ошибка или добавлялось что-то новое, мы могли перекомпилировать только один или два файла, записать их на соответствующие микросхемы и передать только эти микросхемы инженеру службы эксплуатации для замены.

Мы сделали блоки кода *независимо развертываемыми*. Мы изобрели полиморфную диспетчеризацию. Мы изобрели объекты.

Это была архитектура с подключаемыми модулями (плагинами). Мы подключали микросхемы. Мы разработали ее так, чтобы новые возможности можно было включать в наши продукты установкой микросхем с этими возможностями в соответствующие гнезда. Управляющее меню появлялось автоматически, и так же автоматически происходила привязка новых возможностей к приложению.

Конечно, тогда мы не знали о принципах объектно-ориентированного программирования и также ничего не знали об отделении пользовательского интерфейса от бизнес-правил. Но кое-какие основы были заложены, и они здорово нам помогли.

Описанный подход дал одно неожиданное преимущество: мы получили возможность обновлять микропрограмму через модемное соединение. Обнаружив ошибку, мы могли через модемное соединение связаться с устройством и посредством специальной программы-монитора изменить вектор в ОЗУ, ссылающийся на подпрограмму с ошибкой, подставив адрес в пустой области ОЗУ, и затем загрузить в эту область ОЗУ

исправленную подпрограмму, вводя машинные коды в шестнадцатеричном формате.

Это было большим благом для службы эксплуатации и для наших клиентов. Если у них возникала проблема, им не требовалось заказывать у нас срочную отправку микросхем с исправленным кодом. Систему можно было исправить немедленно, а новые микросхемы установить в ближайший период обслуживания.

Компьютер зоны обслуживания

Роль компьютера зоны обслуживания (Service Area Computer; SAC) в 4-TEL играл мини-компьютер M365. Эта система взаимодействовала со всеми компьютерами COLT посредством выделенных или коммутируемых линий. Она могла отдавать компьютерам COLT команды на выполнение проверки телефонных линий, принимать результаты и анализировать их для выявления любых проблем.

Выбор ремонтников для отправки

Одним из экономических обоснований для создания этой системы было эффективное распределение ремонтников. Ремонтники делились на три категории согласно требованиям профсоюза: обслуживающие телефонные станции, кабели и точки подключения. Ремонтники, обслуживающие телефонные станции, исправляли проблемы в телефонных станциях. Ремонтники, обслуживающие кабели, исправляли проблемы, связанные с нарушением целостности кабелей, соединяющих телефонные станции с клиентами. Ремонтники, обслуживающие точки подключения, исправляли проблемы в помещениях клиентов и в линиях, соединяющих внешний кабель с этими помещениями.

Когда клиент сообщал о проблеме, наша система могла диагностировать ее и определить, каких ремонтников следует направить на ее исправление. Это экономило телефонным компаниям уйму денег, потому что выбор не того ремонтника означал задержку для клиента и впустую потраченное время на поездку ремонтника.

Код, делающий выбор, был разработан и написан талантливым программистом, но жутким собеседником. Процесс создания этого кода можно описать так: «Три недели он смотрел в потолок, потом два дня из него, как из рога изобилия, лился код, а потом он уволился».

Никто не понимал этот код. Каждый раз, пытаясь добавить что-то новое или исправить ошибку, мы что-то да ломали в нем. А поскольку этот код был одним из основных экономических преимуществ нашей системы, каждая новая найденная ошибка вызывала глубокое замешательство.

В конце концов наше руководство предложило нам просто «заморозить» этот код и никогда не менять его. Этот код *официально стал незыблемым*.

Этот опыт показал мне, насколько ценным может быть хороший, чистый код.

Архитектура

Система была написана в 1976 году на ассемблере M365. Это была единственная, монолитная программа, состоявшая примерно из 60 000 строк кода. Операционная система была собственной разработки. Она реализовала невытесняющую многозадачность на основе опроса. Мы назвали ее MPS, от *Multiprocessing System* (*многозадачная система*). Процессор M365 не имел встроенного стека, поэтому локальные переменные задач хранились в специальной области памяти и

копировались при каждом переключении контекста. Доступ к общим переменным регулировался с применением блокировок и семафоров. Проблемы реентерабельности и состояния гонки преследовали нас постоянно.

Бизнес-правила системы не были изолированы ни от логики управления устройствами, ни от логики пользовательского интерфейса. Например, код управления модемом можно было найти повсюду в бизнес-правилах и в пользовательском интерфейсе. В системе не было даже намека на попытку собрать код в модули или использовать абстрактные интерфейсы. Модемы управлялись на уровне битов кодом, разбросанным по всей системе.

То же можно сказать о пользовательском интерфейсе и управлении терминалами. Код, управляющий сообщениями и форматированием текста, не был изолирован. Его можно было найти повсюду в кодовой базе из 60 000 строк.

Модемные модули, которые мы использовали, были предназначены для монтажа на печатных платах. Мы закупали их у сторонней компании и монтировали на наши собственные платы. Они стоили очень дорого. Поэтому, спустя несколько лет, мы решили спроектировать свой модем. Мы, члены группы программирования, попросили конструктора использовать те же битовые форматы для управления новым модемом. Мы объяснили, что код управления модемом разбросан по всей системе и что в будущем этой системе предстоит работать с обоими типами модемов. Мы умоляли и умоляли его: «Пожалуйста, сконструируй новый модем так, чтобы с точки зрения программного управления он ничем не отличался от старого».

Но когда мы получили новый модем, управление им было структурировано иначе. Не просто чуть-чуть иначе, а совершенно иначе.

Спасибо тебе, конструктор.

И что же нам было делать? Мы могли просто взять и заменить все старые модемы новыми. Мы должны были организовать в своих системах управление модемами обоих видов. Программное обеспечение должно было работать с обоими типами модемов одновременно. Мы были обречены окружать код, управляющий модемами, флагами и специальными случаями. Но количество таких мест исчислялось сотнями!

В конечном итоге мы выбрали еще более плохое решение.

Запись данных в последовательную шину, управлявшую всеми нашими устройствами, включая модемы, осуществляла единственная подпрограмма. Мы изменили ее так, чтобы она распознавала битовые шаблоны для управления старым модемом и транслировала их в битовые шаблоны управления новым модемом.

Это было непросто. Команды управления модемами состояли из последовательностей операций записи в разные адреса ввода/вывода в последовательнойшине. Новая подпрограмма должна была интерпретировать последовательности этих команд и транслировать их в другие последовательности, с другими адресами ввода/вывода, другими временными задержками и иным расположением установленных и сброшенных битов.

Мы реализовали это решение, но это было самое худшее из всего, что только можно представить. Но благодаря этому я понял ценность абстрактных интерфейсов и изоляции аппаратуры от бизнес-правил.

Великая модернизация

К началу 1980-х годов идея создания собственных миникомпьютеров и собственных компьютерных архитектур стала выходить из моды. На рынке появилось много более стандартных микрокомпьютеров, использовать которые было проще и дешевле, чем продолжать полагаться на компьютерные архитектуры из конца 1960-х годов. Это, а также жуткая архитектура программного обеспечения SAC вынудили наше техническое руководство инициировать полную реорганизацию системы SAC.

Новая система должна была быть написана на С для дисковой ОС UNIX, действующей на микрокомпьютере с процессором Intel 8086. Наши конструкторы приступили к созданию нового компьютерного оборудования, а отобранная группа программистов, «The Tiger Team», занялась разработкой новой системы.

Не буду утомлять вас подробностями первого провала. Скажу лишь только, что первая команда «Tiger Team» полностью провалилась, потратив два или три человека-года на проект, который так и не был закончен.

Спустя год или два, примерно в 1982 году, была предпринята новая попытка. Ее целью была полная и всеобщая переделка SAC на С, UNIX и нашем заново спроектированном оборудовании с впечатляюще производительным процессором 80286. Мы назвали новый компьютер «Deep Thought»⁸².

Прошли годы, потом еще несколько лет, затем еще несколько и еще. Я не знаю, когда, наконец, была развернута первая система SAC на базе UNIX; полагаю, что это случилось уже после моего ухода (в 1988 году). На самом деле я вообще не уверен, что она была хоть когда-то развернута.

Почему так долго? Просто потому, что команде, занимавшейся созданием новой системы, было сложно угнаться за огромной командой, активно развивавшей старую

систему. Вот лишь некоторые из трудностей, с которыми они столкнулись.

Европа

Примерно в то же время, когда система SAC переписывалась на С, компания начала расширять продажи в Европе. Руководство не могло ждать, пока закончится переделка программного обеспечения, поэтому в Европе стали развертываться старые системы на M365.

Проблема в том, что телефонные системы в Европе сильно отличались от телефонных систем в США. Организация труда рабочих и служащих также существенно отличалась. Поэтому один из наших лучших программистов был отправлен в Великобританию для руководства группой британских разработчиков, занимающихся адаптацией программного обеспечения SAC под европейские требования.

Разумеется, не предпринималось никаких серьезных попыток интегрировать произведенные изменения с программным обеспечением в США. Эта работа велась задолго до появления сетей, позволявших передавать большие объемы кода через океан. Британские разработчики просто взяли за основу код из США и исправляли его под свои нужды.

Это, конечно, вызывало трудности. Ошибки обнаруживались по обе стороны Атлантики, и их требовалось исправлять на обеих же сторонах. Но модули значительно изменились, поэтому было трудно определить, будет ли исправление, сделанное в США, работать в Великобритании.

После нескольких лет изнуряющей гонки и появления скоростной линии, связавшей офисы в США и Великобритании, была предпринята серьезная попытка интегрировать две системы, переместив все различия в настройки. И первая, и

вторая, и третья попытки провалились. Две базы кода, хотя и очень похожие, имели слишком много отличий для успешной реинтеграции, особенно в быстро меняющейся рыночной среде тех лет.

Между тем разработчики из команды «Tiger Team», пытавшиеся переписать все на С и UNIX, заметили, что им также приходится иметь дело с этой двойственностью Европа/США. И конечно, это не способствовало ускорению их движения вперед.

В заключение о SAC

Я мог бы продолжать рассказывать истории об этой системе, но это неприятные для меня воспоминания. Достаточно сказать, что многие из этих горьких уроков в моей карьере программиста были получены в процессе погружений в ужасный ассемблерный код SAC.

Язык С

Оборудование на базе микропроцессора 8085, которое мы использовали в проекте 4-Tel Micro, дало нам относительно недорогую компьютерную платформу, пригодную для реализации многих разных промышленных проектов. Мы имели 32 Кбайт ОЗУ и еще 32 Кбайт ПЗУ и чрезвычайно гибкую и мощную схему управления периферией. Чего у нас не было, так это гибкого и удобного языка программирования. На ассемблере 8085 было просто неинтересно писать код.

Кроме того, ассемблер, которым мы пользовались, был написан нашими же программистами. Он работал на наших ЭВМ M365 и использовал ленточный накопитель на картриджах, описанный в разделе «Laser Trim».

Так сложилось, что наш ведущий инженер-электронщик убедил генерального директора в необходимости иметь настоящий компьютер. На самом деле он не знал, что с ним делать, но он имел большое политическое влияние. Поэтому мы купили PDP-11/60.

Я, в то время скромный программист, был в восторге. Я точно знал, что я хочу сделать с этим компьютером. Я решил, что это будет *моя машина*.

Когда пришли инструкции и руководства, за несколько месяцев до того, как пришла сама машина, я забрал их домой и буквально проглотил. К моменту, когда был доставлен компьютер, я довольно глубоко изучил, как действует аппаратное и программное обеспечение, настолько глубоко, насколько это возможно в домашних условиях.

Я помогал писать заказ на поставку. В частности, я указал, что новый компьютер должен иметь дисковый накопитель. Я решил, что мы должны купить два дисковых привода, в которые можно устанавливать сменные пакеты дисков емкостью по 25 мегабайт каждый⁸³.

Пятьдесят мегабайт! Этот объем казался неисчерпаемым! Я помню, как по ночам я гулял по коридорам офиса, подобно злой ведьме Бастинде, и восклицал: «Пятьдесят мегабайт! Хаха-ха-ха-ха-ха-ха-ха!»

Руководитель службы эксплуатации здания отгородил небольшую комнату, в которой могло поместиться шесть терминалов VT100. Я украсил ее стены картинами с изображениями космоса. Наши программисты могли бы использовать эту комнату для написания и компиляции кода.

Когда пришла машина, я потратил несколько дней на ее установку, подключение терминалов и проверку работоспособности всех ее компонентов. Этот труд был в радость.

Мы закупили стандартные ассемблеры для 8085 в компании *Boston Systems Office* и переписали код 4-Tel Micro с использованием нового синтаксиса. Мы построили систему кросс-компиляции, позволявшую нам выгружать скомпилированный двоичный код из PDP-11 в наши вычислительные комплексы на базе 8085 и в программаторы ПЗУ. И план удался — все работало как часы.

C

Но у нас оставалась еще проблема в виде языка ассемблера 8085, который мы продолжали использовать. Это было ложкой дегтя в бочке меда. Я уже слышал про «новый» язык, широко использовавшийся в *Bell Labs*. Они назвали его «C». Поэтому я купил книгу *The C Programming Language*⁸⁴ Кернигана и Ритчи. Как и руководства для PDP-11, несколькими месяцами раньше, я проглотил эту книгу.

Я был поражен простотой и элегантностью этого языка. Он обладал мощностью языка ассемблера, но открывал доступ к этой мощности, предоставляя более удобный синтаксис. Я был в восторге.

Я купил компилятор С в компании *Whitesmiths* и запустил его на PDP-11. Он производил код на языке ассемблера, синтаксис которого был совместим с компилятором 8085 от *Boston Systems Office*. То есть у нас появилась возможность писать программы на С для 8085! Мы были готовы к работе.

Теперь оставалась единственная проблема — убедить программистов, пишущих на ассемблере, что они должны перейти на С. Но эту кошмарную историю я расскажу в другой раз...

BOSS

Наша платформа на процессоре 8085 не имела операционной системы. Мой опыт работы с системой MPS на ЭВМ M365 и знание простейших механизмов прерываний в IBM System 7 подсказывали, что нам нужен простой диспетчер задач для 8085. Поэтому я задумал написать BOSS: Basic Operating System and Scheduler (базовая операционная система с планировщиком)[85](#).

Значительная часть BOSS была написана на С. Она могла конкурентно выполнять несколько задач. Многозадачность не была вытесняющей — переключение происходило не по прерываниям, а так же, как в системе MPS для M365, с помощью простого механизма опроса. Опрос происходил всякий раз, когда задача блокировалась в ожидании события.

Вызов, блокирующий задачу, выглядел в BOSS так:

```
block(eventCheckFunction);
```

Этот вызов приостанавливал текущую задачу, помещал eventCheckFunction в список опроса и связывал ее с только что заблокированной задачей. Затем выполнялся цикл опроса, в котором последовательно вызывались функции из списка, пока одна из них не возвращала true. Затем возобновлялось выполнение задачи, связанной с этой функцией.

То есть, как я говорил выше, это был простой невытесняющий диспетчер задач.

Это программное обеспечение стало основой для большого количества проектов, разрабатывавшихся в следующие несколько лет. Но одним из первых был pCCU.

pCCU

Конец 1970-х — начало 1980-х годов было шумным временем для телефонных компаний. Одной из причин волнений стала цифровая революция.

В предыдущем веке коммутационный узел и телефон клиента связывала пара медных проводов. Эти провода связывались в кабели, образующие разветвленную сеть по всей стране. Иногда их подвешивали на столбах, иногда прокладывали под землей.

Медь — дорогой металл, и телефонные компании владели тоннами, буквально тоннами этого металла в виде проводов, опутывающих страну. Капиталовложения были огромными. Большую часть этих капиталовложений можно было сэкономить передачей телефонных разговоров через цифровые соединения. Одна пара медных проводов могла бы переносить сотни диалогов в цифровой форме.

В ответ телефонные компании приступили к замене старого аналогового коммутационного оборудования современными цифровыми коммутаторами.

Наш продукт 4-Tel тестировал медные провода, но не мог тестировать цифровые соединения. В цифровой среде все еще использовались медные провода, но они были намного короче, чем раньше, и сосредоточены в основном рядом с телефонами клиентов. Сигнал от телефонной станции передавался в цифровой форме до местного распределительного пункта, где преобразовывался обратно в аналоговую форму и доставлялся до клиента по обычной паре медных проводов. Это означало, что наша измеряющая аппаратура должна находиться там, где начинаются медные провода, но устройство соединения с ним должно было оставаться в телефонной станции. Проблема в том, что все наши компьютеры COLT объединяли устройство соединения и аппаратуру измерения в одном корпусе. (Мы

могли бы сэкономить целое состояние, узнав об этой очевидной архитектурной границе на несколько лет раньше!)

В результате мы задумали продукт с новой архитектурой: CCU/CMU (COLT control unit/COLT measurement unit — модуль управления COLT/модуль измерения COLT). По задумке модуль CCU должен находиться в телефонной станции и обеспечивать выбор телефонных линий для тестирования. Модуль CMU должен находиться в местных распределительных пунктах и измерять уровень сигнала в медных проводах, идущих к телефону клиента.

Проблема состояла в том, что на каждый модуль CCU приходилось много модулей CMU. Информация о том, какой модуль CMU использовать для каждого телефонного номера, содержалась в самом цифровом коммутаторе. То есть модуль CCU должен был опросить цифровой коммутатор, чтобы определить, с каким модулем он должен взаимодействовать.

Мы пообещали телефонным компаниям, что создадим эту новую архитектуру к моменту их перехода. Мы знали, что это займет месяцы, если не годы, поэтому чувствовали себя раскованно. Мы также знали, что для разработки нового программно-аппаратного комплекса CCU/CMU потребуется несколько человеко-лет.

Ловушка планирования

С течением времени у нас постоянно всплывали какие-то неотложные вопросы, для решения которых мы были вынуждены откладывать разработку архитектуры CCU/CMU. Мы были уверены в своем решении, потому что телефонные компании тоже все время откладывали разработку цифровых коммутаторов. Заглядывая в их графики, мы были уверены, что

у нас еще масса времени, поэтому мы постоянно откладывали нашу разработку.

Но настал день, когда мой начальник вызвал меня к себе в кабинет и сказал: «*Один из наших клиентов развертывает цифровой коммутатор в следующем месяце. К тому времени у нас должен быть рабочий комплекс CCU/CMU*».

Я был в ужасе! Как за месяц выполнить работы, требующие одного человека-года? Но у моего начальника был план...

На самом деле нам не нужна была полная архитектура CCU/CMU. Телефонная компания, устанавливающая цифровой коммутатор, была маленькой. У них была только одна телефонная станция и всего два местных распределительных пункта. Важно отметить, что «местные» распределительные пункты были не совсем местными. Фактически они являлись старыми добрыми аналоговыми коммутаторами, к которым было подключено несколько сотен клиентов. К тому же эти коммутаторы принадлежали к типу, который успешно можно было тестиировать компьютерами COLT. Но самое замечательное, что телефонные номера клиентов включали всю информацию, необходимую для определения распределительного пункта. Если номер телефона включал цифру 5, 6 или 7 в определенной позиции, это означало, что он подключен к распределительному пункту 1; иначе — к распределительному пункту 2.

Итак, как объяснил мне мой начальник, нам в действительности нужна не полная архитектура CCU/CMU, а только простой компьютер в телефонной станции, связанный модемными линиями с двумя стандартными компьютерами COLT в распределительных пунктах. Компьютер SAC мог бы связываться с нашим компьютером в телефонной станции, а тот, в свою очередь, мог бы декодировать телефонный номер и посыпать команды на выполнение тестирования в компьютер

COLT, находящийся в соответствующем распределительном пункте.

Так родилась система pCCU.

Это был первый продукт, написанный на С и использующий BOSS, развернутый у клиента. На разработку мне понадобилось что-то около недели. Эта история ничем не выделяется с архитектурной точки зрения, но она служит хорошим предисловием для следующего проекта.

DLU/DRU

В начале 1980-х годов в числе наших клиентов была телефонная компания из Техаса. Она обслуживала обширную географическую область. Область была настолько большой, что для ее обслуживания требовалось несколько сервисных центров с ремонтниками. В этих центрах находились люди, которым требовались терминалы, связанные с нашим компьютером SAC.

Возможно, вы подумали, что это даже не проблема, но вспомните, что история происходила в начале 1980-х годов. Удаленные терминалы были редким явлением. Хуже того, оборудование SAC предполагало размещение терминалов поблизости. Наши терминалы фактически подключались к нашей собственной высокоскоростной последовательнойшине.

У нас имелась возможность подключить удаленные терминалы только через модемы, которые в начале 1980-х годов могли передавать данные со скоростью не более 300 бит в секунду. Наши клиенты были недовольны такой низкой скоростью.

В то время уже существовали высокоскоростные модемы, но они стоили очень дорого и им требовалось «условно»

постоянное соединение. Качество коммутируемых соединений было определено недостаточно высоким.

Наши клиенты требовали найти решение. Нашим ответом стала система DLU/DRU.

Аббревиатура DLU/DRU расшифровывалась как «Display Local Unit» (локальное устройство отображения) и «Display Remote Unit» (удаленное устройство отображения). Устройство DLU — это компьютерная плата, включаемая в шасси SAC и играющая роль платы диспетчера терминала. Но вместо управления последовательнойшиной, соединяющей локальные терминалы, это устройство принимало поток символов и мультиплексировало его через единственное модемное соединение с пропускной способностью 9600 бит/с.

Устройство DRU размещалось удаленно, у клиента. Оно подключалось к другому концу модемного соединения с пропускной способностью 9600 бит/с и включало оборудование для управления терминалами, подключенными к разработанной нами последовательнойшине. Оно демультиплексировало символы, принимаемые из модема, и передавало их соответствующим локальным терминалам.

Странно, правда? Нам пришлось разрабатывать решение, настолько обыденное в наши дни, что о нем никто не задумывается. Но тогда...

Нам пришлось даже придумать свой протокол связи, потому что тогда стандартные протоколы не были общедоступны в виде исходных кодов. Фактически все происходило задолго до того, как у нас появилось подключение к Интернету.

Архитектура

Система имела очень простую архитектуру, но в ней были некоторые интересные особенности, которые я хотел бы

подчеркнуть. Во-первых, оба устройства были сконструированы на нашей технологии 8085, программное обеспечение для обоих было написано на С и использовало BOSS. Но на этом их сходство заканчивается.

Над проектом работали два человека. Я, как руководитель проекта, и Майк Карев (Mike Carew), мой близкий друг. Я взял на себя проектирование и разработку DLU, а Майк — DRU.

Архитектура DLU основывалась на модели потока данных. Каждая задача выполняла небольшую узкоспециализированную работу и передавала свои результаты следующей задаче в конвейере, используя очередь. Представьте модель конвейеров и фильтров в UNIX. Архитектура получилась сложной. Добавлять данные в очередь могла одна задача, а извлекать их из нее — несколько.

Представьте сборочную линию. На каждом участке такой сборочной линии выполняется единственная, простая, узкоспециализированная операция. После выполнения операции на одном участке продукт перемещается по конвейеру к следующему. Иногда сборочная линия может разветвляться на несколько линий. Иногда несколько линий могут сливаться в одну линию. Такова была архитектура устройства DLU.

Устройство DRU, созданное Майком, было основано на совершенно другой схеме. Он создал по одной задаче на каждый терминал и просто выполнял в ней всю работу, имеющую отношение к данному терминалу. Никаких очередей. Никаких потоков данных. Просто несколько больших и одинаковых задач, каждая из которых управляет своим терминалом.

Это полная противоположность сборочной линии. В данном случае можно провести аналогию с несколькими опытными

сборщиками, каждый из которых собирает свой продукт целиком.

В то время я думал, что моя архитектура лучше. Майк, конечно, думал, что его архитектура лучше. У нас было много интересных дискуссий на эту тему. Но, так или иначе, мы оба неплохо поработали. И мне оставалось лишь уяснить, что программные архитектуры могут быть совершенно разными, но одинаково эффективными.

VRS

На протяжении 1980-х годов появлялись все более и более новые технологии. Одной из таких технологий было *голосовое управление компьютером*.

Одной из функций системы 4-Tel было оказание помощи ремонтнику в поиске места повреждения кабеля. Процедура выглядела так:

- Тестировщик, работающий на телефонной станции, с помощью нашей системы определял расстояние в футах до точки повреждения с точностью около 20%. Затем он направлял ремонтника к точке доступа, ближайшей к повреждению.
- Ремонтник, прибыв на место, вызывал тестировщика и просил начать процесс определения места повреждения. Тестировщик запускал процедуру поиска неисправности в системе 4-Tel. Система измеряла электрические характеристики поврежденной линии и выводила на экран сообщения, описывающие действия, которые требовалось выполнить дальше, такие как вскрыть кабель или закоротить кабель.

- Тестирующий сообщал ремонтнику, какие операции требуется выполнить, а ремонтник сообщал, когда та или иная операция была выполнена. После этого тестирующий сообщал системе, что затребованная операция выполнена, и она возобновляла тестирование.
- После двух-трех операций система рассчитывала новое расстояние до повреждения. После этого ремонтник мог переехать в указанное место и процесс повторялся вновь.

Представьте, насколько проще было бы, если бы ремонтник, поднявшись на столб или опору, мог бы сам управлять системой. Именно эту возможность дали нам новые голосовые технологии. Ремонтник мог бы вызвать систему непосредственно, управлять ею с помощью тонального набора и прослушивать ответы, читаемые приятным голосом.

Название

Компания провела небольшой конкурс по выбору названия для новой системы. Одним из самых необычных предложений было имя SAM CARP. Оно расшифровывалось как «Still Another Manifestation of Capitalist Avarice Repressing the Proletariat» (еще одно проявление капиталистической алчности, подавляющей пролетариат). Разумеется, это название не было выбрано.

Еще одно название — Teradyne Interactive Test System (интерактивная тест-система Tradyne) — тоже не было выбрано.

Также не было выбрано название Service Area Test Access Network (сеть доступа к тест-системе зоны обслуживания).

Победило название VRS: Voice Response System (система с голосовым ответом).

Архитектура

Мне не довелось работать над этой системой, но я был в курсе происходящего. Историю, которую я собираюсь рассказать, вы узнаете из вторых рук, но это не повлияло на ее правдивость.

Это был период эйфории, связанной с микрокомпьютерами, операционными системами UNIX, С и базами данных SQL. Мы были полны решимости использовать все это.

Из множества баз данных мы выбрали UNIFY — систему управления базами данных для UNIX, что было идеально для нас.

База данных UNIFY поддерживала также новую технологию с названием *Embedded SQL*, позволявшую внедрять команды SQL в виде строк прямо в код на языке С. Что мы и не преминули сделать, причем повсюду.

Я имею в виду, это было так необычно — иметь возможность поместить код SQL прямо в программный код, в любое место, куда захотите. И куда мы захотели? Да повсюду! В результате код SQL оказался размазан ровным слоем по всему программному коду.

В те времена SQL еще не был солидным стандартом. Каждый производитель добавлял в язык SQL какие-то свои особенности. Поэтому нестандартный код SQL и нестандартные вызовы UNIFY API можно было увидеть повсюду в программном коде.

Но все работало замечательно! Система оказалась успешной. Ремонтники пользовались ею, и телефонные компании полюбили ее. Жизнь улыбалась нам.

Затем поддержка продукта UNIFY прекратилась.

Ой-ё!

Поэтому мы решили переключиться на SyBase. Или это была Ingress? Я не помню. Но важно не это, а то, что нам пришлось

отыскать в коде на С весь код SQL и вызовы нестандартного API и заменить их аналогичным кодом, взаимодействующим с новой базой данных.

Через три месяца мы прекратили бесплодные попытки. Мы не могли заставить систему работать с новой базой данных. Мы оказались настолько привязанными к UNIFY, что не было никакой надежды реструктурировать код с более или менее разумными издержками.

В результате мы наняли сторонних специалистов, поддерживавших UNIFY для нас, заключив с ними контракт на техническое обслуживание. И конечно, с каждым годом затраты на обслуживание росли.

В заключение о VRS

Таким способом я узнал, что базы данных — это деталь, которую следует изолировать от общей бизнес-цели системы. Это также одна из причин, почему мне не нравится зависимость от сторонних систем.

Электронный секретарь

В 1983 году наша компания оказалась на стыке компьютерных, телекоммуникационных и голосовых систем. Наш генеральный директор считал, что такое положение может способствовать разработке новых продуктов. Он поручил команде из трех человек (включая меня) придумать, спроектировать и реализовать новый продукт для компании.

Нам не потребовалось много времени, чтобы прийти к идее создания электронного секретаря (Electronic Receptionist; ER). Суть была проста. Когда вы звонили в компанию, электронный секретарь (ER) поднимал трубку и спрашивал, с кем бы вы

хотели поговорить. Вы могли ответить на вопрос, нажимая кнопки на телефоне в режиме тонального набора, и таким способом сообщить имя человека, а ER соединял вас с ним. То есть пользователи могли позвонить на номер электронного секретаря и, используя простые тональные команды, связаться с нужным человеком, где бы тот ни находился. Фактически система могла попробовать несколько альтернативных номеров.

Когда кто-то звонил электронному секретарю и набирал RMART (мой код), тот, в свою очередь, звонил по первому номеру в моем списке. В случае неудачи он звонил по следующему номеру, и т.д. Если связаться со мной не удалось, электронный секретарь мог бы записать голосовое сообщение для меня.

Затем электронный секретарь предпринимал периодические попытки найти меня и доставить сообщение, оставленное для меня кем-то другим.

Это была первая система голосовой почты, и мы⁸⁶ получили патент на нее.

Мы собрали все необходимое оборудование для этой системы — компьютерную плату, плату памяти, платы для связи и записи голоса и все остальное. Роль компьютерной платы играла плата компьютера *Deep Thought* (Думатель) на процессоре Intel 80286, о котором я уже рассказывал.

Для каждой телефонной линии была создана отдельная голосовая плата. Эти платы содержали телефонный интерфейс, аппаратуру для кодирования/декодирования голоса, некоторый объем памяти и микрокомпьютер Intel 80186.

Программное обеспечение для главной компьютерной платы было написано на С. В качестве операционной системы использовалась MP/M-86, одна из первых многозадачных

дисковых систем, управляемых из командной строки. MP/M — это UNIX для бедных.

Программное обеспечение для голосовых плат было написано на ассемблере и действовало без операционной системы. Взаимодействие компьютера Deep Thought с голосовыми платами осуществлялось через общую память.

Архитектуру этой системы в наши дни назвали бы *сервис-ориентированной*. Каждая телефонная линия обслуживалась отдельным процессом, действующим под управлением MP/M.

Когда поступал входящий звонок, запускался начальный процесс для обработки и звонок передавался ему. По мере перехода обслуживания звонка из одной стадии в другую запускался соответствующий процесс-обработчик и управление передавалось ему.

Сообщения передавались между этими службами через дисковые файлы. Текущая выполняющаяся служба могла определить, какую службу запустить далее; записать необходимую информацию в дисковый файл; выполнить команду для запуска этой службы и затем завершиться.

Я впервые занимался созданием такой системы. В действительности это был первый раз, когда я выступал в роли главного архитектора продукта. Все, что имело отношение к программному обеспечению, было моим — и все работало как надо.

Я не могу сказать, что архитектура этой системы была «чистой» в том смысле, в каком предполагает эта книга; она не была архитектурой «сменных модулей» (плагинов). Однако в ней имелись явные признаки истинных границ. Службы развертывались независимо, и каждая отвечала за определенную предметную область. В системе имелись процессы высокого и низкого уровня, и многие зависимости простирались в правильном направлении.

Конец электронного секретаря

К сожалению, попытки продать этот продукт оказались неудачными. Компания *Teradyne* специализировалась на производстве контрольно-измерительного оборудования. Мы не представляли, как проникнуть на рынок кабинетского оборудования.

После неоднократных попыток, продолжавшихся в течение более двух лет, наш генеральный директор отказался от них и — к сожалению — отозвал заявку на патент. Патент был получен другой компанией, подавшей заявку через три месяца после нашей; так мы сдали рынок голосовой почты и электронной переадресации звонков.

Ой-ёй!

Зато теперь вы не сможете поставить мне в вину появление всех этих раздражающих машин, отправляющих наше существование.

Система командирования ремонтников

Электронный секретарь потерпел неудачу как продукт, но у нас осталось программное обеспечение и оборудование, которые мы могли бы использовать для расширения линейки имеющихся продуктов. Кроме того, успех VRS на рынке убедил нас, что мы должны предложить систему с голосовым ответом для организации взаимодействий с ремонтниками, которые не зависят от наших систем тестирования.

Так родилась CDS — система командирования ремонтников (*Craft Dispatch System*). По сути, система CDS была электронным секретарем (ER), но ориентированным на решение узкого круга задач в области управления ремонтниками.

Когда в телефонной линии обнаруживалась проблема, в сервисный центр посыпалась заявка. Заявки хранились в автоматизированной системе. Когда ремонтник, работающий в поле, завершал обслуживание заявки, он звонил в сервисный центр, чтобы получить следующее назначение. Оператор сервисного центра извлекал следующую заявку и читал ее вслух ремонтнику.

Мы приступили к автоматизации этого процесса. Наша цель состояла в том, чтобы создать систему CDS, которой мог бы позвонить ремонтник, работающий в поле, и запросить следующее назначение. Система должна была обратиться к системе заявок и зачитать извлеченную из нее заявку. Система CDS могла бы следить за назначением заявок ремонтникам и информировать систему заявок о приеме заявки к исполнению.

Эта система имела немало интересных особенностей, связанных с взаимодействием с системой заявок, системой управления предприятием и системами автоматизированного тестирования.

Опыт разработки сервис-ориентированной архитектуры ER пробудил во мне желание еще раз опробовать эту идею. Машина состояний для обработки заявок оказалась намного сложнее, чем машина состояний для обработки звонков в электронном секретаре. Я приступил к созданию архитектуры, которая в наши дни называется *архитектурой микрослужб*.

Каждый переход между этапами обслуживания любого звонка, каким бы незначительным он ни был, вынуждал систему запускать новую службу. В действительности все переходы описывались в текстовом файле, который читала система. Каждое событие, поступающее в систему по телефонной линии, вызывало соответствующий переход между службами. Существующий процесс мог запускать для

обработки события новый процесс, в соответствии с текущим состоянием; затем существующий процесс мог завершиться или ждать в очереди.

Такое решение позволяло нам изменять порядок выполнения операций без изменения кода (принцип открытости/закрытости). Мы легко могли добавлять новые службы независимо от других и внедрять их в поток, просто изменяя текстовый файл с описаниями переходов. Мы могли делать это даже в процессе работы системы. Иными словами, у нас появился механизм горячей замены и эффективный язык выполнения бизнес-процессов (Business Process Execution Language; BPEL).

Старый прием использования дисковых файлов для взаимодействий между службами, реализованный в электронном секретаре, был слишком медленным для этих быстро сменяющих друг друга служб, поэтому мы изобрели механизм общей памяти, который назвали 3DBB⁸⁷. Механизм 3DBB позволял обращаться к данным по именам; в качестве имен использовались названия, присвоенные экземплярам машины состояний.

Общая память прекрасно подходила для хранения строк и констант, но ее нельзя было использовать для хранения сложных структур данных. Причина была чисто технической. Каждый процесс в MP/M находился в собственном сегменте памяти. Указатели на данные в одном сегменте не имели смысла в другом. Как следствие, данные в общей памяти не могли содержать указатели. Строки можно хранить без ограничений, но деревья, связные списки и любые другие структуры данных с указателями — нет.

Система заявок получала заявки из разных источников. Некоторые добавлялись автоматически, некоторые — вручную. Заявки, добавляемые вручную, создавались операторами,

общавшимися с клиентами. По мере того как клиент описывал проблему, оператор вводил жалобы и наблюдения в структурированный поток текста. Он выглядел примерно так:

```
/rno 8475551212 /noise /dropped-calls
```

Идея должна быть понятна. Символ `/` начинал новую тему. За этим символом следовал код, за кодом — параметры. Таких кодов были *тысячи*, а в описании каждой заявки их могло быть до нескольких десятков. Хуже того, операторы часто допускали опечатки или ошибались в форматировании. Люди прекрасно справлялись с интерпретацией таких заявок, но не машины.

Перед нами стояла задача — декодировать эти строки, интерпретировать, исправить возможные ошибки и преобразовать их в голосовые сообщения, которые могли бы прослушивать ремонтники по телефону. Для этого кроме всего прочего требовалось реализовать очень гибкий анализ и формат представления данных. Данные в этом формате должны были передаваться через общую память, которая могла обслуживать только строки.

Итак, в короткие перерывы между посещениями клиентов я изобрел схему, которую назвал FLD: *Field Labeled Data* (данные с маркированными полями). В наши дни мы назвали бы этот формат XML или JSON. Формат представления отличался, но идея была той же. Схема FLD представляла данные в виде бинарного дерева, которые ассоциировали имена с данными в рекурсивной иерархии. Данные в формате FLD можно было получить с применением простого API и преобразовать в строковое представление, идеально подходящее для передачи через общую память.

То есть уже в 1985 году микрослужбы обменивались информацией через общую память — аналог сокетов, — используя формат, аналогичный XML.

Ничто не ново под луной.

Clear Communications

В 1988 году группа сотрудников *Teradyne* покинула компанию, чтобы основать новую фирму под названием *Clear Communications*. Я присоединился к ним несколькими месяцами позже. Нашей целью было создание программного обеспечения для системы, осуществлявшей контроль качества связи по линиям T1 — цифровым линиям для междугородной связи по всей стране. Мы видели ее как огромный монитор с картой США и пересекающей ее сеткой линий T1, которые начинали мигать красным, если обнаруживались проблемы.

В 1988 году графические пользовательские интерфейсы только стали появляться. Модели Apple Macintosh было всего пять лет. Windows в ту пору не стоила доброго слова. Но компания *Sun Microsystems* строила станции Sparc, имевшие превосходный графический интерфейс X-Window. Поэтому мы пошли с Sun, а значит, с языком С и операционной системой UNIX.

Это был период становления новой компании. Мы работали по 70–80 часов в неделю. У нас было видение. У нас была мотивация. У нас была воля. У нас была энергия. У нас был опыт. Мы были равны. Мы мечтали стать миллионерами. Мы были безрассудны.

Код на С лился из нас как из рога изобилия. Мы вталкивали и впихивали его то туда, то сюда. Мы строили огромные воздушные замки. У нас были процессы, очереди сообщений и великолепные архитектуры. Мы написали семиуровневый ISO-стек взаимодействий с нуля — вплоть до канального уровня передачи данных.

Мы писали код графического интерфейса. ГРЯЗНЫЙ КОД! Бог мой! Мы писали СЛИШКОМ ГРЯЗНЫЙ КОД.

Я лично написал на С функцию с именем `gi()` длиной в 3000 строк; ее имя расшифровывалось как Graphic Interpreter (графический интерпретатор). Это был шедевр грязи. Это не единственный грязный код, который я написал в Clear, но этот был самым позорным.

Архитектура? Вы шутите? Это был период становления. У нас не было времени на архитектуру. Только код, черт возьми! *Код, от которого зависело наше будущее благополучие!*

Поэтому мы писали, писали и писали код. А через три года мы прогорели. Нет, мы смогли продать один или два экземпляра системы. Но рынок не особенно интересовался нашим грандиозным видением, и наши инвесторы были уже сыты по горло.

В тот момент я ненавидел свою жизнь. Я видел, как прахом идут все мои усилия и рушатся мои мечты. У меня стали возникать конфликты на работе, конфликты дома из-за работы и конфликты с самим собой.

И тогда я принял телефонный звонок, который все изменил.

Обстановка

За два года до этого звонка произошло два важных события.

Во-первых, мне удалось настроить ииср-соединение с соседней компанией, имевшей ииср-соединение с другим объектом, подключенным к Интернету. Это, конечно же, были коммутируемые соединения. Наша основная Sparc-станция (стоявшая на моем рабочем столе) использовала модем со скоростью передачи 1200 бит/с для соединения с нашим ииср-хостом дважды в день. Это дало нам электронную почту и

доступ к Netnews (первой социальной сети, где люди обсуждали разные интересные вопросы).

Во-вторых, Sun выпустила компилятор C++. Я интересовался языком C++ и объектно-ориентированным программированием начиная с 1983 года, но компилятор тогда было трудно найти. Поэтому, как только представилась возможность, я сразу же сменил язык. Я оставил функции на C из 3000 строк и начал писать код на C++ для Clear. И я учился...

Я читал книги. Конечно, я прочитал *The C++ Programming Language*⁸⁸ и *The Annotated C++ Reference Manual*⁸⁹ Бьёрна Страуструпа. Я прочитал прекрасную книгу Ребекки Вирфс-Брок о проектировании, основанном на ответственности: *Designing Object Oriented Software*. Я прочитал OOA, OOD и OOP Петера Коуда. Я прочитал *Smalltalk-80* Адель Голдберг. Я прочитал *Advanced C++ Programming Styles and Idioms*⁹⁰ Джеймса О. Коплиена. Но самое главное, пожалуй, я прочитал *Object Oriented Design with Applications*⁹¹ Гради Буча.

Какое имя! Гради Буч. Как можно забыть такое имя. Более того, он был *главным научным консультантом* в компании с названием *Rational!* Как я хотел быть *главным научным консультантом!* И поэтому я читал его книгу. И учился, учился, учился...

Одновременно с учебой я начал вступать в дискуссии на Netnews, подобно тому, как ныне люди дискутируют в Facebook. Я участвовал в обсуждениях на тему C++ и ООП. В течение двух лет я освобождался от разочарований, которые приобретал на работе, обсуждая с сотнями пользователей Usenet лучшие особенности языка и принципы проектирования. Через какое-то время многое для меня стало проясняться.

Именно в одной из таких дискуссий были заложены принципы SOLID.

И все эти обсуждения и, возможно, даже некоторый смысл, который я вносил, сделали меня заметным...

Дядюшка Боб

В *Clear* работал один молодой инженер, Билли Фогель. Билли всем давал прозвища. Меня он называл дядюшкой Бобом. Мое имя действительно Боб, но я подозреваю, что он намекал на Дж. Р. «Боба» Доббса (рис. П.9).



Рис. П.9. Дж. Р. «Боб» Доббс (J. R. «Bob» Dobbs)

Сначала я терпел это. Но проходили месяц за месяцем, и его непрекращающееся: «дядюшка Боб... дядюшка Боб», — на фоне неудач и разочарования новой компанией стали вызывать раздражение.

А потом однажды зазвонил телефон.

Телефонный звонок

Это был сотрудник из кадрового агентства. Он узнал мое имя, подыскивая человека, знающего C++ и принципы объектно-ориентированного проектирования. Я не знаю, как он его

узнал, но предполагаю, что это как-то связано с моим присутствием в Netnews.

Он сказал, что у него есть свободная вакансия в Кремниевой долине, в компании *Rational*. Они искали специалиста для помощи в разработке CASE-инструмента⁹².

Кровь отхлынула от моего лица. Я знал, что это за компания. Не знаю, откуда я это узнал, но я знал. Это была компания Гради Буча. Я увидел возможность оказаться в одной команде с Гради Бучем!

ROSE

Я поступил на работу в компанию *Rational*, как программист по контракту, в 1990 году. Я работал над продуктом с названием ROSE. Это был инструмент, позволявший программистам рисовать диаграммы Буча — диаграммы, которые Гради использовал в книге *Object-Oriented Analysis and Design with Applications* (пример такой диаграммы изображен на рис. П.10).

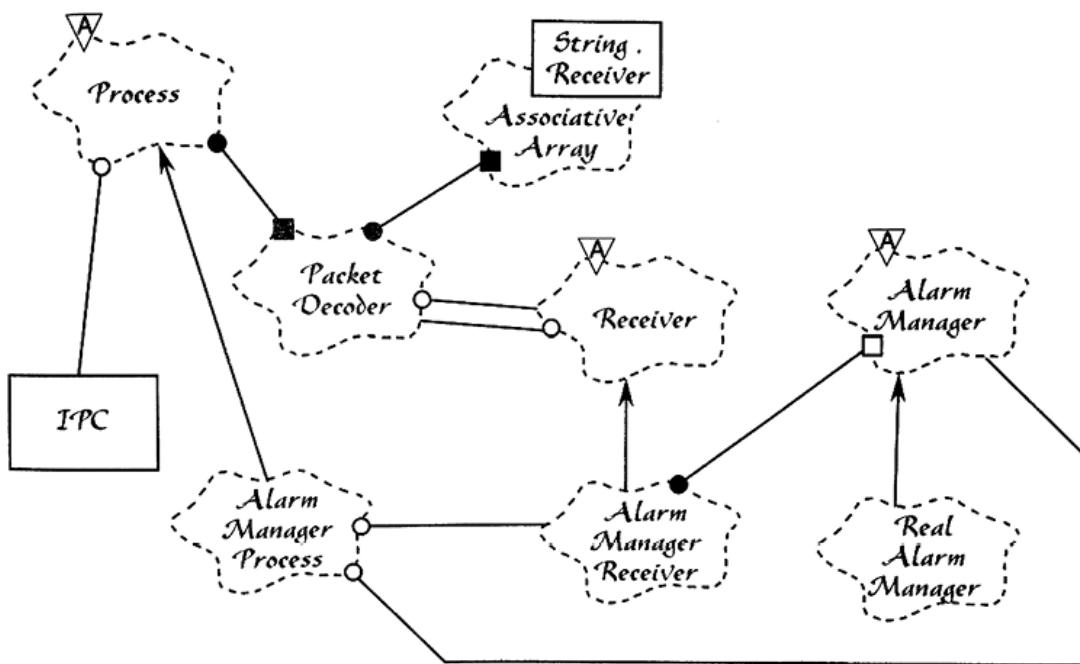


Рис. П.10. Диаграмма Буча

Диаграммы Буча были очень выразительными. Они предвосхитили такие диаграммы, как UML.

Продукт ROSE имел архитектуру — настоящую архитектуру. Она состояла из истинных уровней, и все зависимости между уровнями были ориентированы как должно. Архитектура обеспечила этому продукту возможность раздельного выпуска, разработки и развертывания его компонентов.

О, он не был идеальным. Мы еще многое не понимали в архитектурных принципах. Например, мы не создали истинную структуру со сменными модулями (плагинами).

Мы также попались на одно из самых неудачных увлечений тех дней — мы использовали так называемую объектно-ориентированную базу данных.

Но в целом опыт был отличным. Я проработал полтора прекрасных года с командой из *Rational* над ROSE. Это был один из самых познавательных опытов в моей профессиональной жизни.

Продолжение дискуссий

Конечно, я не прекратил участвовать в обсуждениях на Netnews. В действительности я резко увеличил свое присутствие в сети. Я начал писать статьи для *C++ Report*. И с помощью Гради приступил к своей первой книге: *Designing Object-Oriented C++ Applications Using the Booch Method*.

Одна мысль беспокоила меня. Пусть она была недостойной, но тем не менее. Никто не называл меня «дядюшкой Бобом». Я обнаружил, что мне этого не хватает. Поэтому я допустил ошибку, добавив подпись «Uncle Bob» (дядюшка Боб) в мои электронные письма и сообщения в Netnews. И это имя

прилипло ко мне. В конце концов я понял, что это довольно неплохой бренд.

...Под любым другим именем

ROSE — это гигантское приложение на C++. Оно состояло из уровней со строго соблюдаемым правилом зависимости. Это не то правило, что я описал в данной книге. Мы ориентировали наши зависимости не в сторону политик более высокого уровня, а в более традиционном направлении — в направлении потока управления. Пользовательский интерфейс зависел от представления, которое зависело от правил выполнения операций с данными, которые зависели от базы данных. В результате эта, не совсем удачная ориентация зависимостей способствовала кончине продукта.

Архитектура ROSE была похожа на архитектуру хорошего компилятора. Графическая нотация «преобразовывалась» во внутреннее представление; затем это представление обрабатывалось правилами и сохранялось в объектно-ориентированной базе данных.

Объектно-ориентированные базы данных были относительно новой идеей, и мир ОО был взволнован возможными перспективами. Каждый объектно-ориентированный программист желал иметь объектно-ориентированную базу данных в своей системе. Идея была относительно простой и глубоко идеалистической. База данных хранила объекты, а не таблицы. База данных должна была выглядеть как ОЗУ. Когда производилось обращение к объекту, он просто возникал в памяти. Если этот объект ссылался на другой объект, тот другой объект тоже появлялся в памяти при первой попытке обратиться к нему. Это было похоже на волшебство.

Выбор этой базы данных стал, пожалуй, самой большой нашей ошибкой. Мы желали волшебства, а в результате получили большой, медленный, навязчивый, дорогостоящий сторонний фреймворк, который превратил нашу жизнь в ад, препятствуя нашему движению вперед почти на всех уровнях.

Выбор базы данных был не единственной нашей ошибкой. Гораздо более серьезной ошибкой фактически стало излишнее усердие в отношении к проектированию архитектуры. У нас получилось намного больше уровней, чем я описал здесь, и каждый привносил свои накладные расходы на взаимодействия. Это отрицательно сказалось также на продуктивности команды.

В итоге, после многих человеко-лет работы, тяжелой борьбы и выпуска двух слабых версий, весь инструмент был заменен маленьkim приложением, написанным небольшой командой из Висконсина.

Так я узнал, что великие архитектуры иногда приводят к великим провалам. Архитектура должна быть достаточно гибкой, чтобы подстраиваться под размер задачи. Разработка архитектуры для уровня предприятия, когда в действительности нужен маленький и удобный инструмент для настольного компьютера, — это верный рецепт провала.

Регистрационные экзамены для архитекторов

В начале 1990-х годов я стал настоящим консультантом. Я ездил по миру и обучал людей новым объектно-ориентированным подходам. В своих консультациях я строго ограничивался дизайном и архитектурой объектно-ориентированных систем.

Одним из моих первых клиентов была компания *Educational Testing Service* (ETS). Она имела контракт с Национальным

советом регистрационной коллегии по архитектуре (National Council of Architects Registry Board, NCARB) по проведению регистрационных экзаменов для новых кандидатов в архитекторы.

Любой желающий стать зарегистрированным архитектором (проектирующим здания) в США или Канаде должен сдать регистрационный экзамен. На этом экзамене кандидату предлагается решить несколько архитектурных задач, связанных с проектированием зданий. Кандидату передается ряд требований для проектирования общественной библиотеки, ресторана или церкви и затем предлагается нарисовать комплект архитектурных схем.

Результаты собираются и сохраняются, пока не соберется комиссия ведущих архитекторов для оценки. Создание комиссии — большое, дорогостоящее событие и вечный источник задержек и недопонимания.

Совет NCARB хотел автоматизировать процесс и организовать сдачу экзаменов с использованием компьютера, а оценку поданных решений производить с помощью другого компьютера. Совет NCARB предложил компании ETS разработать соответствующее программное обеспечение, а ETS наняла меня для создания команды разработчиков с целью разработки этого продукта.

Компания ETS разбила проблему на 18 экзаменационных заданий. Для каждого требовалось создать приложение с графическим интерфейсом в стиле систем автоматизированного проектирования, которое кандидат мог бы использовать для оформления своего решения. Анализ и оценку решений должны производить другие 18 приложений.

Мы с моим партнером Джимом Ньюкирком заметили, что эти 36 приложений имеют огромное сходство. Все 18 экзаменационных приложений с графическим интерфейсом

должны реализовать схожие операции и использовать схожие механизмы. Другие 18 приложений, оценивающие решения, должны использовать один и тот же математический аппарат. Учитывая большое количество общих элементов, Джим и я решили разработать многократно используемую инфраструктуру, которая должна лечь в основу всех 36 приложений. Фактически мы продали эту идею компании ETS, заявив, что на разработку первого приложения потребуется много времени, зато все последующие будут выпущены в течение нескольких недель.

Прочитав эти строки, многие из вас могли бы выразить крайнее удивление. Но читатели постарше наверняка помнят, как объектно-ориентированный подход обещал «многократное использование». В ту пору мы все были убеждены, что если писать хороший, чистый, объектно-ориентированный код на C++, в результате получится много-много кода, пригодного для многократного использования.

Итак, мы приступили к разработке первого приложения — самого сложного в пакете. Мы назвали его *Vignette Grande*.

Мы вдвоем работали над *Vignette Grande* с прицелом на создание инфраструктуры многократного использования. Нам понадобился год. К концу этого года у нас были 45 000 строк инфраструктурного и 6000 строк прикладного кода. Мы принесли этот продукт в ETS, и они заключили с нами контракт на создание других 17 приложений в ограниченный срок.

Мы с Джимом наняли еще трех разработчиков и приступили к созданию следующих нескольких экзаменационных приложений.

Но что-то пошло не так. Мы обнаружили, что наша инфраструктура оказалась почти непригодна для многоразового использования. Она не вписывалась в новые приложения. Имелись шероховатости, мешавшие работе.

Мы были сильно обескуражены, но считали, что знаем, как поступить с этим. Мы пришли в ETS и сказали, что задерживаемся — что 45 000-строчную инфраструктуру требуется переписать или хотя бы скорректировать и что для этого потребуется еще много времени.

Я думаю, излишне говорить, что в ETS не очень обрадовались этой новости. Тем не менее мы начали все сначала. Мы отложили старую инфраструктуру и приступили к созданию сразу четырех новых экзаменационных приложений. Мы заимствовали некоторые идеи и код из старой инфраструктуры, но переделывали их так, чтобы они вписывались во все четыре приложения без изменений.

На это ушел еще один год. Мы написали другую инфраструктуру с 45 000 строк кода плюс четыре экзаменационных приложения, содержащих от 3000 до 6000 строк каждое. Разумеется, отношения между приложениями и инфраструктурой следовали правилу зависимости. Приложения были плагинами для инфраструктуры. Все высокоуровневые политики находились в инфраструктуре. Прикладной код просто «склеивал» разные операции.

Отношения между инфраструктурой и приложениями оценки решений оказались намного сложнее. Высокоуровневая политика оценки находилась в приложении. Инфраструктура оценки подключалась как плагин к приложению оценки.

Конечно, приложения обоих видов компоновались статически, поэтому в наших головах полностью отсутствовало понятие «плагин». И тем не менее зависимости были выстроены в полном соответствии с правилом зависимости.

Закончив эти четыре приложения, мы приступили к следующим четырем. На этот раз они были готовы через несколько недель, как мы и предсказывали. Непредусмотренная задержка отняла у нас почти год, поэтому, чтобы

ускорить процесс и уложиться в график, мы наняли еще одного программиста.

Мы уложились в срок и выполнили свои обязательства. Наш клиент был счастлив. Мы были счастливы. Жизнь была прекрасна.

Но мы получили хороший урок: нельзя создать универсальную инфраструктуру, не создав прежде работающую инфраструктуру. Универсальные инфраструктуры должны создаваться одновременно с *несколькими* приложениями, использующими их.

Заключение

Как я сказал в начале, это приложение несколько автобиографично. Я описал наиболее яркие моменты в проектах, которые, по моему мнению, имели большое архитектурное влияние. И конечно, я привел несколько эпизодов, не имеющих прямого отношения к технической стороне книги, но которые тем не менее имели большое значение.

Разумеется, это далеко не полная история. В моей карьере было много других проектов. Кроме того, я намеренно ограничил освещение истории началом 1990-х годов, потому что у меня есть еще одна книга, посвященная событиям конца 1990-х годов.

Я надеюсь, что вам понравился этот небольшой экскурс по волнам моей памяти и вы смогли почерпнуть что-то новое из моего опыта.

[73](#) В ASC нам рассказали историю, как транспортировалась эта ЭВМ. Ее везли на большом грузовике с полуприцепом вместе с мебелью. По пути грузовик, двигаясь на

высокой скорости, зацепил крышей мост. ЭВМ не пострадала, но она смешилась вперед и вдребезги раздавила мебель.

[74](#) Сегодя мы могли бы сказать, что она работала с тактовой частотой 142 КГц.

[75](#) Представьте массу диска, какой кинетической энергией он обладал! Однажды мы заметили металлическую стружку, вылетевшую из корпуса диска. Мы вызвали ремонтника, и он тут же попросил выключить диск. Закончив ремонт, он сказал, что износился один из подшипников. Затем он рассказал нам, как однажды вовремя не отремонтированный диск сорвался с креплений, пробил бетонную стену и застрял в автомобиле на стоянке рядом.

[76](#) С электронно-лучевыми трубками зеленого свечения, способные отображать только символы ASCII.

[77](#) CRUD — аббревиатура, обозначающая набор основных операций с данными: Create (создание), Read (чтение), Update (изменение) и Delete (удаление). — Примеч. пер.

[78](#) Магическое число 72 пришло из эпохи перфокарт Hollerith, содержащих по 80 символов каждая. Последние 8 символов «резервировались» под порядковый номер на случай, если вы уроните и рассыпите колоду.

[79](#) Да, я понимаю, что это оксюморон.

[80](#) Для этого в микросхемах имелись прозрачные пластиковые окошки, через которые можно было видеть кремниевые кристаллы внутри и стирать данные ультрафиолетом.

[81](#) Да, я знаю, что после записи программного обеспечения в ПЗУ оно превращается в микропрограмму, но микропрограмма не перестает быть программным обеспечением.

[82](#) «Думатель», так назывался компьютер из фантастического романа Дугласа Адамса, давший «ответ на главный вопрос жизни, вселенной и всего такого» ([https://ru.wikipedia.org/wiki/Автостопом_по_галактике_\(серия_романов\)](https://ru.wikipedia.org/wiki/Автостопом_по_галактике_(серия_романов))). — Примеч. пер.

[83](#) RKO7.

[84](#) Последнее переиздание на русском языке: Брайн Керниган, Денис Ритчи. Язык программирования С.М.: Вильямс, 2016. — Примеч. пер.

[85](#) Позднее эта аббревиатура получила другую расшифровку: Bob's Only Successful Software (успешное программное обеспечение Боба).

[86](#) Держателем патента стала наша компания. В нашем контракте с работодателем однозначно говорилось, что права на любые наши изобретения будут принадлежать компании. Мой начальник сказал мне: «Вы продали нам это за один доллар, но мы не выплатили вам этот доллар».

[87](#) Three-Dimensional Black Board — трехмерная черная доска. Если вы родились в 1950-х годах, вам наверняка более знакомой покажется фраза: Drizzle, Dazzle, Druzzle, Drone.

[88](#) Страуструп Б. Язык программирования C++. М.: Бином, 2017. — Примеч. пер.

[89](#) Эллис М., Страуструп Б. Справочное руководство по языку программирования C++ с комментариями. М.: Мир, 1992. — Примеч. пер.

[90](#) Копlien Д., Программирование на C++. Классика CS. СПб.: Питер, 2004. — Примеч. пер.

[91](#) Гради Буч, Роберт А. Максимчук, Майкл У. Энгл, Бобби Дж. Янг, Джим Коналлен, Келли А. Хьюстон. Объектно-ориентированный анализ и проектирование с примерами приложений. М.: Вильямс, 2010. — Примеч. пер.

[92](#) Computer Aided Software Engineering — автоматизация разработки программного обеспечения.