ЧИСТАЯ АРХИТЕКТУРА

ИСКУССТВО РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

РОБЕРТ МАРТИН

Библиотека программиста (Питер)

Роберт Мартин

Чистая архитектура. Искусство разработки программного обеспечения

«Питер»
2018

УДК 004.3
ББК 32.973.2-018

Мартин Р. С.

Чистая архитектура. Искусство разработки программного обеспечения / Р. С. Мартин — «Питер», 2018 — (Библиотека программиста (Питер))

ISBN 978-0-13-449416-6

«Идеальный программист» и «Чистый код» — легендарные бестселлеры Роберта Мартина — рассказывают, как достичь высот профессионализма. «Чистая архитектура» продолжает эту тему, но не предлагает несколько вариантов в стиле «решай сам», а объясняет, что именно следует делать, по какой причине и почему именно такое решение станет принципиально важным для вашего успеха. Роберт Мартин дает прямые и лаконичные ответы на ключевые вопросы архитектуры и дизайна. «Чистую архитектуру» обязаны прочитать разработчики всех уровней, системные аналитики, архитекторы и каждый программист, который желает подняться по карьерной лестнице или хотя бы повлиять на людей, которые занимаются данной работой. В форматах: a4.pdf и ios.ePub представлены файлы от издательства.

УДК 004.3
ББК 32.973.2-018

ISBN 978-0-13-449416-6

Содержание

Роберт Мартин
Чистая архитектура
Искусство разработки программного обеспечения

2018
Переводчик *А. Макарова*
Технический редактор *Н. Суслова*
Литературный редактор *Е. Герасимова*
Художники *Л. Егорова, С. Заматевская, Р. Яцко*
Корректоры *С. Беляева, Н. Викторова*
ISBN 978-5-4461-0772-8

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

Предисловие

О чем 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 Дженга, 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 Шекспир*[1]

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.

---

[1] Перевод Т. Гнедич. – *Примеч. ред.*

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.

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 1022 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 сде-

[2] Фильм, вышедший в США в 1970 году, в нашей стране известный под названием «Колосс: Проект Форбина». – *Примеч.*

лана 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 ее[5] 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 C 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 Алан Тьюринг написал первый программный код.

Изменилось только одно: тогда, в прошлом, мы не знали этих правил. Поэтому нарушали 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.

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)
Джейсон 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!

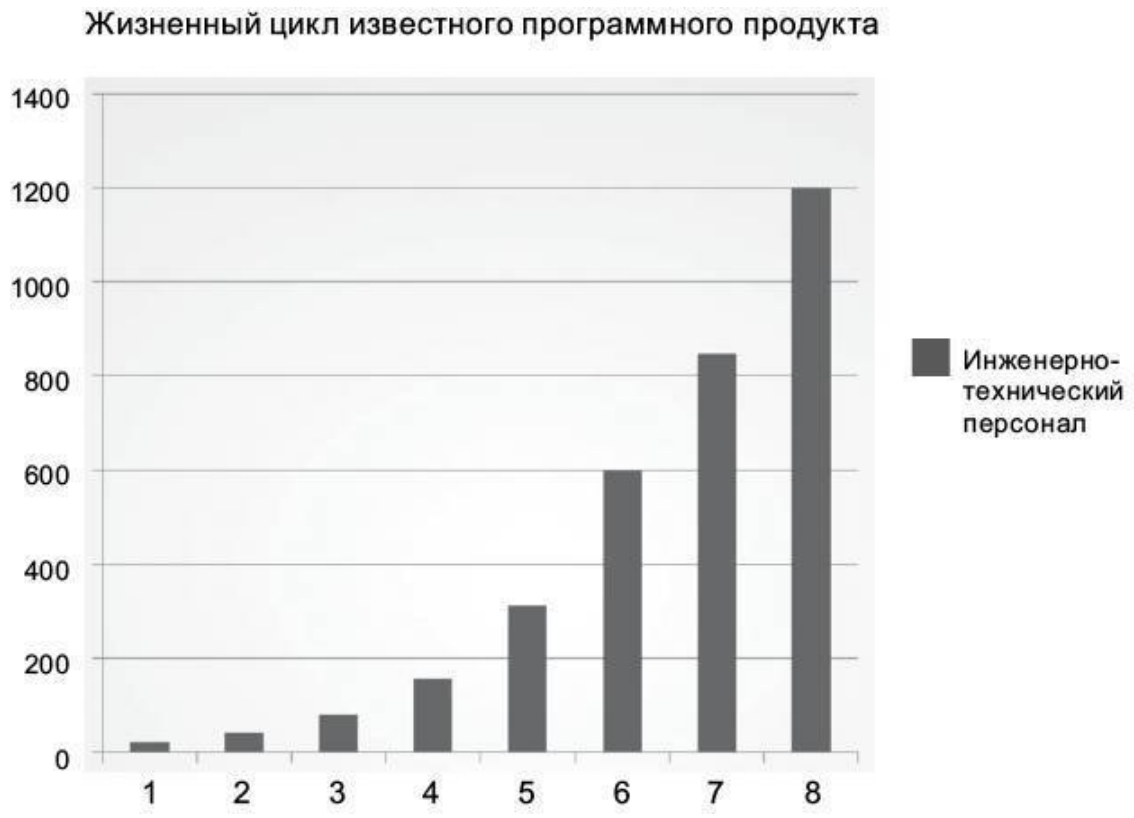


Рис. 1.1. Рост численности инженерно-технического персонала. Воспроизводится с разрешения автора презентации Джейсона Гормана (Jason Gorman)

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

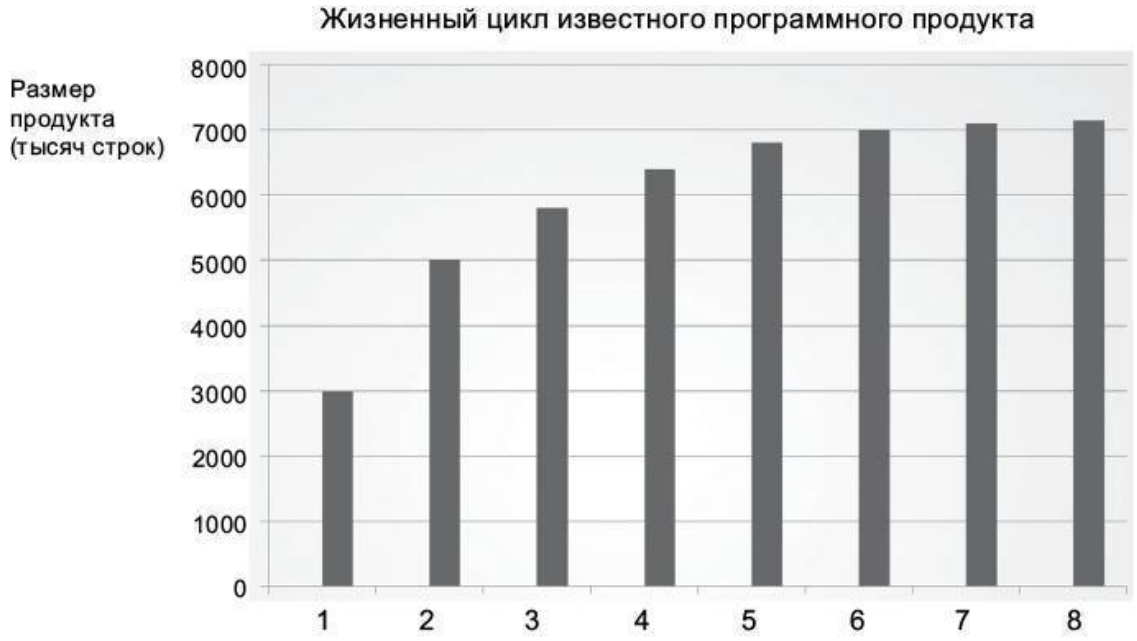


Рис. 1.2. Продуктивность за тот же период

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.

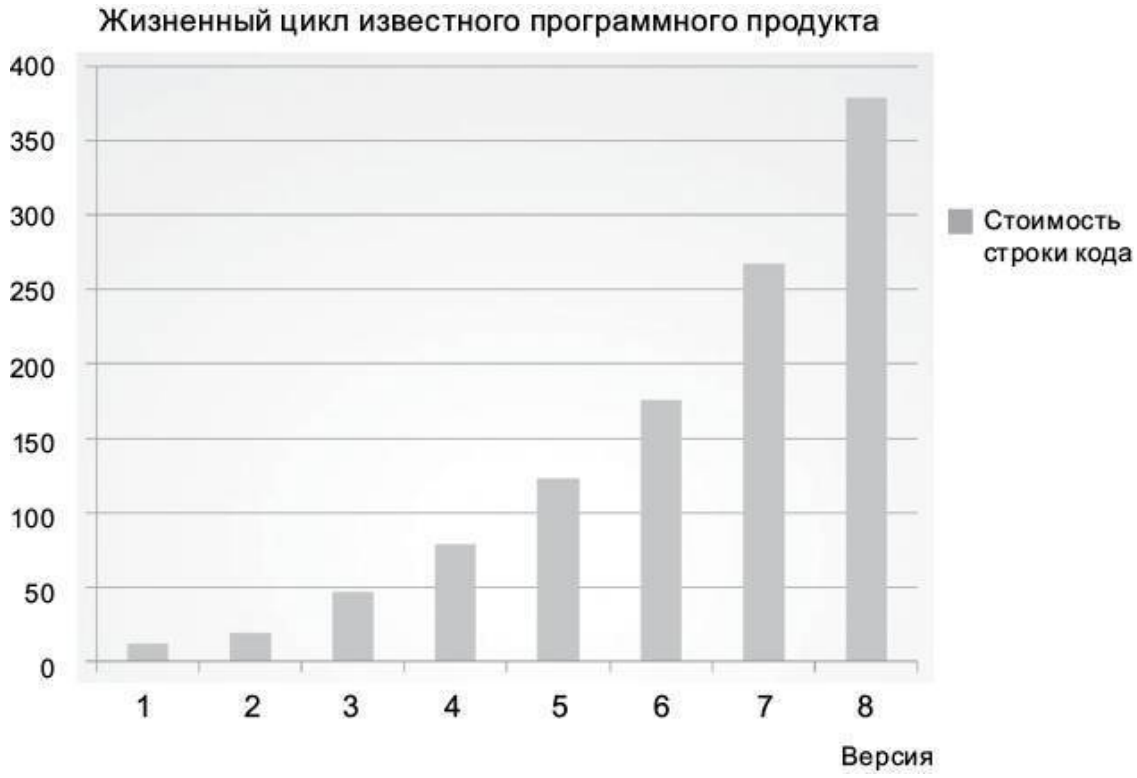


Рис. 1.3. Изменение стоимости строки кода с течением времени

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 показываю 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 удручающе, 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.

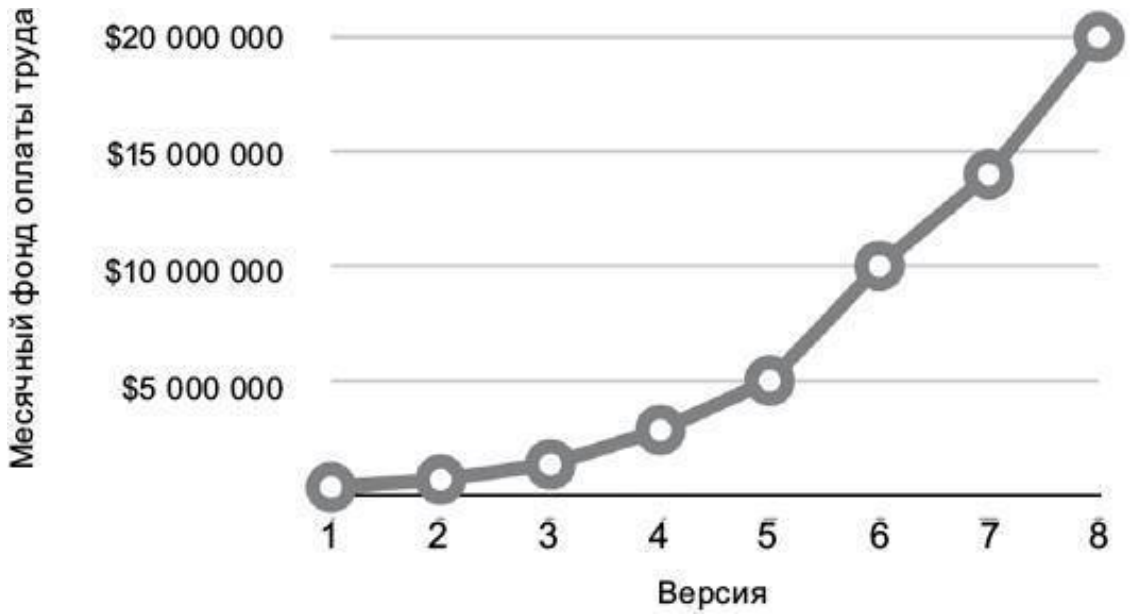


Рис. 1.5. Изменение фонда оплаты труда разработчиков с выпуском новых версий

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».

Притча 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 бездействует. Если впустить его, он уменьшит производительность до нуля за считаные 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

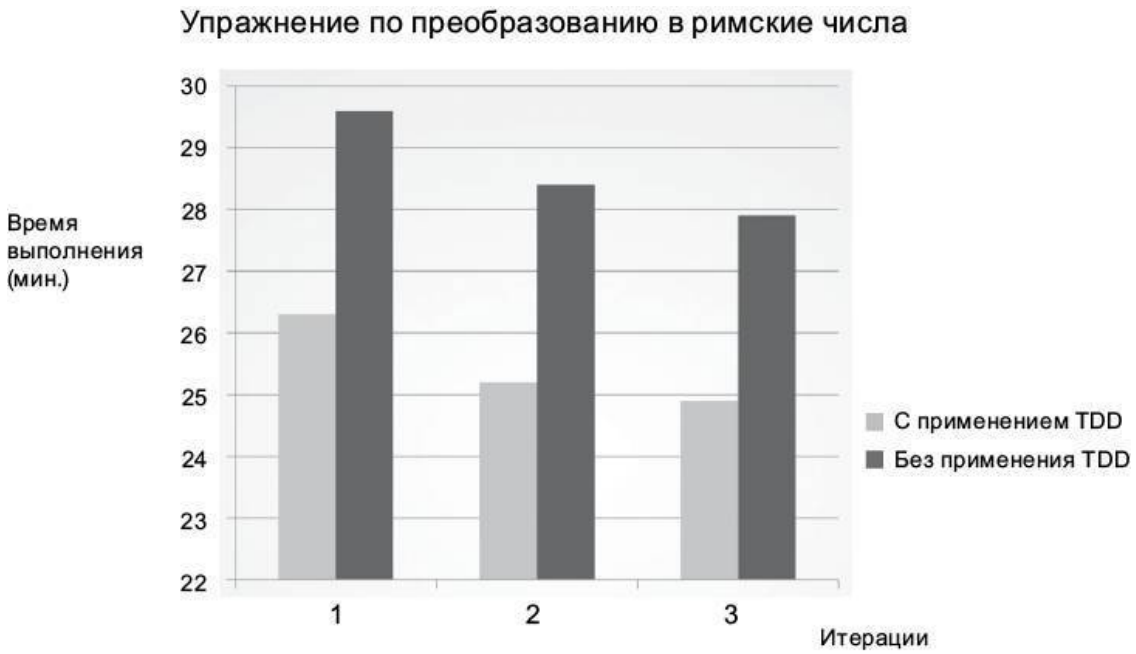


Рис. 1.6. Время на выполнение итерации с использованием и без использования методики TDD

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 такое хорошая архитектура. Чтобы создать систему, дизайн и архитектура которой способствуют уменьшению трудозатрат и увеличению продуктивности, 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 невоз-

можно, 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 Эйзенхауэр говорил так:

> *У меня есть два вида дел, срочные и важные. Срочные дела, как правило, не самые важные, а важные – не самые срочные*[9].

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,

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

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.

II. Начальные основы: парадигмы программирования

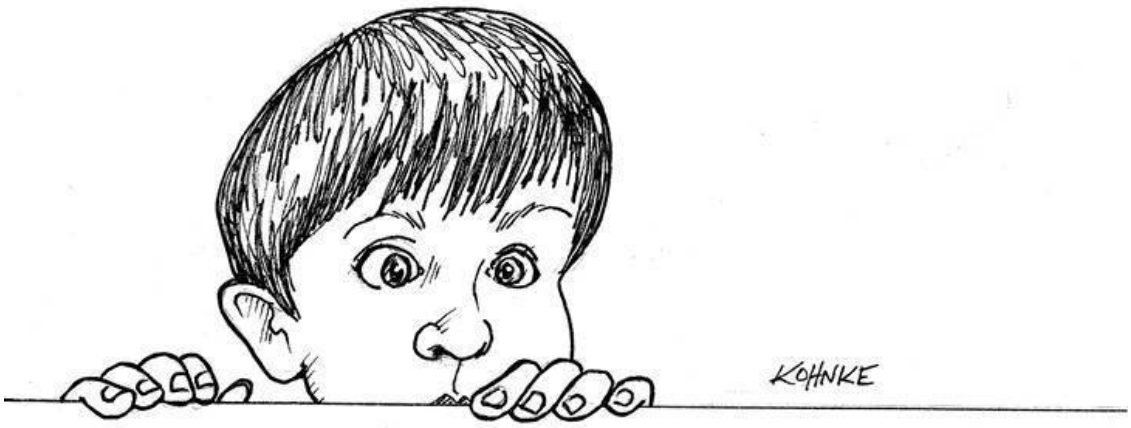
Архитектура программного обеспечения начинается с кода, поэтому начнем обсуждение архитектуры с рассказа о самом первом программном коде.

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?

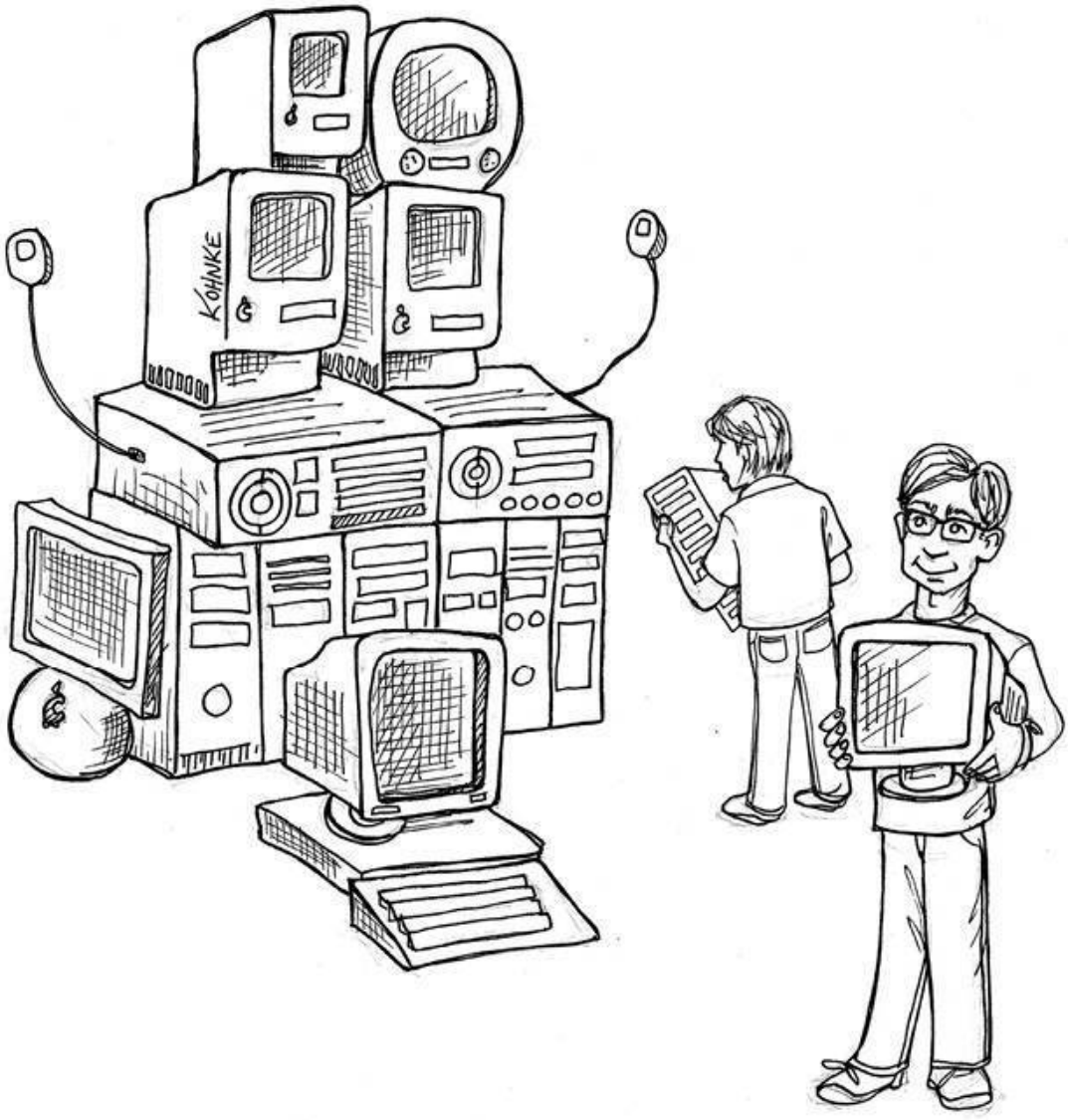
Вероятно, нет. Скорее всего, эти три парадигмы останутся единственными, которые мы увидим, – по крайней мере единственными, что-то отнимающими у нас. Доказательством отсутствия новых парадигм может служить тот факт, что все три известные парадигмы были открыты в течение десяти лет, между 1958 и 1968 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 вышесказанное 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 «доброкачественного» 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, Дейкстре 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 структур[11].

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`

[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.*

Конец ознакомительного фрагмента.