

Дэвид Фарли

Современная программная инженерия

**ПО в эпоху эджайла
и непрерывного развертывания**

Легендарный разработчик и создатель continuous delivery Дэйв Фарли о принципах, лежащих в основе эффективной разработки современного ПО.

MODERN SOFTWARE ENGINEERING

DOING WHAT WORKS TO BUILD
BETTER SOFTWARE FASTER

David Farley

Дэвид Фарли

Современная программная инженерия

ПО в эпоху эджайла
и непрерывного развертывания

Ещё больше книг в нашем телеграм канале:

<https://t.me/bookofgeek>



Оглавление

О книге «Современная программная инженерия»	11
Вступительное слово.....	13
Введение	17
Определение программной инженерии	19
Структура книги	20
Благодарности.....	21
Об авторе	23
От издательства	24

I

ЧТО ТАКОЕ ПРОГРАММНАЯ ИНЖЕНЕРИЯ

ГЛАВА 1. ВВЕДЕНИЕ.....	26
Инженерия — практическое применение теоретической науки.....	26
Что такое программная инженерия?	27
Переосмысление понятия программной инженерии.....	29
Зарождение программной инженерии.....	31
Меняя парадигму.....	33
Итоги.....	34
ГЛАВА 2. ЧТО ТАКОЕ ИНЖЕНЕРИЯ?	35
Производство не наша проблема	35
Инженерия проектная, а не производственная	36
Рабочее определение инженерии	42
Инженерия — это не код	43

Почему инженерия важна?	45
Ограничения ремесленного производства	46
Точность и масштабируемость.....	47
Управление сложностью.....	48
Повторяемость и точность измерений	50
Инженерия, креативность и ремесло	52
Почему то, чем мы занимаемся, — это не программная инженерия.....	54
Компромиссы	55
Иллюзия прогресса	55
От ремесла к инженерному делу	57
Ремесла недостаточно	58
Пришло ли время мыслить иначе?	58
Итоги.....	60
 ГЛАВА 3. ОСНОВЫ ИНЖЕНЕРНОГО ПОДХОДА	61
Разработка — это индустрия изменений?	61
Важность измерений	63
Использование метрик стабильности и пропускной способности	65
Основы программной инженерии	67
Экспертное познание	68
Экспертное управление сложностью	69
Итоги.....	71
 II	
ОПТИМИЗАЦИЯ ДЛЯ ОБУЧЕНИЯ	
 ГЛАВА 4. ИТЕРАТИВНЫЙ ПОДХОД	74
Практические преимущества итеративного подхода	76
Итерация как стратегия защитного проектирования	78
Сила планирования.....	80
Практическая ценность итеративного подхода.....	87
Итоги.....	89
 ГЛАВА 5. ОБРАТНАЯ СВЯЗЬ	90
Практический пример важности обратной связи.....	91

Обратная связь в кодировании	94
Обратная связь в интеграции	95
Обратная связь в дизайне	98
Обратная связь в архитектуре	101
Быстрая обратная связь предпочтительнее.....	103
Обратная связь в дизайне продукта.....	104
Обратная связь в организации и культуре.....	105
Итоги.....	108
 ГЛАВА 6. ИНКРЕМЕНТАЛИЗМ	109
Важность модульности	110
Инкрементализм в организации	112
Инструменты инкрементализма.....	114
Ограничение влияния изменений.....	116
Инкрементальный дизайн.....	118
Итоги.....	121
 ГЛАВА 7. ЭМПИРИЗМ.....	122
Основано на реальности	123
Отделяйте эмпиризм от эксперимента	123
«Я знаю ошибку!»	124
Избегайте самообмана.....	126
Изобретайте реальность, соответствующую аргументам	127
Опирайтесь на реальность	131
Итоги.....	132
 ГЛАВА 8. БЫТЬ ЭКСПЕРИМЕНТАТОРОМ	133
Что значит быть экспериментатором?.....	134
Обратная связь	135
Гипотеза	137
Измерения	138
Управление переменными.....	139
Автоматизированное тестирование как эксперимент	141
Помещая результаты тестирования в контекст.....	142
Объем эксперимента	145
Итоги.....	146

III

ОПТИМИЗАЦИЯ ДЛЯ УПРАВЛЕНИЯ СЛОЖНОСТЬЮ

ГЛАВА 9. МОДУЛЬНОСТЬ.....	148
Признаки модульности.....	150
Недооценка важности хорошего дизайна.....	150
Важность тестируемости.....	152
Тестируемость повышает модульность	154
Службы и модульность	161
Развертываемость и модульность.....	162
Модульность в разных масштабах.....	165
Модульность в системах, создаваемых человеком.....	166
Итоги.....	168
 ГЛАВА 10. СВЯЗНОСТЬ.....	169
Модульность и связность: основы дизайна	169
Базовое снижение связности.....	170
Контекст имеет значение	173
Высокопроизводительное программное обеспечение	177
Отсылка к связанности	178
Обеспечение высокой связности с помощью TDD	179
Как добиться связности	179
Цена плохой связности	182
Связность в человеческих системах.....	183
Итоги.....	183
 ГЛАВА 11. РАЗДЕЛЕНИЕ ОТВЕТСТВЕННОСТИ.....	185
Внедрение зависимости.....	189
Разделение необходимой и случайной сложности	190
Важность DDD	194
Тестируемость	196
Порты и адаптеры	196
Когда использовать порты и адаптеры	199

Что такое API?	201
Использование TDD для разделения ответственности	202
Итоги.....	203
 ГЛАВА 12. СОКРЫТИЕ ИНФОРМАЦИИ И АБСТРАКЦИЯ	204
Абстракция или сокрытие информации.....	204
Почему образуются большие комки грязи?.....	205
Организационные и культурные проблемы.....	205
Технические вопросы и вопросы проектирования	208
Страх чрезмерного усложнения.....	212
Повышение абстракции с помощью тестирования.....	215
Сила абстракции.....	216
Дырявые абстракции.....	218
Выбор подходящих абстракций	220
Абстракции из предметной области	222
Абстрактная случайная сложность	223
Изолируйте код от сторонних систем	227
Всегда скрывайте информацию, если это возможно.....	228
Итоги.....	229
 ГЛАВА 13. УПРАВЛЕНИЕ СВЯЗАННОСТЬЮ	230
Стоимость связанности	230
Масштабирование.....	231
Микросервисы.....	232
Снижение связанности может означать больше кода	235
Слабая связанность — не единственная важная деталь	237
Выбор в пользу слабой связанности	238
В чем отличие от разделения ответственности?.....	239
DRY — это слишком просто	240
Асинхронность как инструмент слабой связанности	242
Проектирование слабой связанности.....	244
Слабая связанность в организациях	245
Итоги.....	247

IV

Инструменты программной инженерии

Глава 14. ИНСТРУМЕНТЫ ИНЖЕНЕРНОЙ ДИСЦИПЛИНЫ.....	250
Что такое программная разработка	251
Тестируемость как инструмент	253
Точки измерения.....	256
Сложности с обеспечением тестируемости.....	257
Как улучшить тестируемость.....	261
Разворачиваемость.....	263
Скорость.....	265
Управление переменными.....	266
Непрерывная доставка	267
Общие инструменты для поддержки разработки	268
Итоги.....	269
Глава 15. СОВРЕМЕННЫЙ ИНЖЕНЕР-РАЗРАБОТЧИК.....	270
Инженерия как человеко-ориентированный процесс.....	272
Организации — лидеры в цифровой сфере	273
Результаты и механизмы	276
Устойчивость и широкая применимость.....	278
Основы инженерной дисциплины	282
Итоги.....	282

О КНИГЕ «СОВРЕМЕННАЯ ПРОГРАММНАЯ ИНЖЕНЕРИЯ»

«Современная программная инженерия» описывает реальные актуальные приемы, которые используются опытными инженерами для создания ПО. Техники, о которых рассказывается в этой книге, не являются раз и навсегда определенными, предписывающими или линейными. Они эмпирические, итеративные, основанные на обратной связи, экономичные и сосредоточенные на выполнении кода, то есть такие, какие и требуются для современной разработки.

*Гленн Вандербург (Glenn Vanderburg),
технический директор Nubank*

Отдельным техникам программной инженерии посвящено множество книг. Эта книга — иная. Дэйв рассматривает самую суть программной инженерии и то, чем она отличается от простой разработки. Он объясняет, почему высоко-классный инженер ПО должен быть экспертом в управлении сложностью, как освоить уже существующие техники и научиться оценивать потенциал всевозможных инженерных идей. Эта книга для тех, кто считает разработку ПО действительно инженерной дисциплиной, — неважно, новичок ли он в этой сфере или занимается разработкой уже не одно десятилетие.

*Дэйв Хаунслоу (Dave Hounslow),
инженер-разработчик*

Это очень важные темы, и здорово иметь под рукой руководство, где описаны все они, в комплексе.

*Майкл Найгард (Michael Nygard),
автор книги *Release IT*, программист, архитектор ПО*

Я прочитал книгу Дэйва Фарли и могу сказать: это действительно то, что нужно. Ее должен прочесть каждый, кто хочет стать инженером-разработчиком ПО или отточить свое мастерство в этой сфере. Она содержит полезные практические советы. Ее стоит включить в программу вузов и учебных центров.

*Брайан Финстер (Bryan Finster),
старший инженер и ведущий архитектор USAF Platform One*

*Эту книгу я посвящаю своей жене Кейт
и сыновьям Тому и Бену.*

*Многие годы Кейт неизменно поддерживает меня
в моем писательском труде; она мой верный соратник,
вдохновитель и лучший друг.*

*Том и Бен – парни, которыми я восхищаюсь и которых
я люблю. Работая над этой книгой, я был счастлив также
вести с ними несколько совместных проектов.*

Спасибо за вашу помощь и поддержку.

Вступительное слово

Я изучала компьютерные науки в университете, и, конечно, в учебный план входили дисциплины, в названии которых были слова «программная инженерия».

К моменту поступления в вуз я не была новичком в программировании и даже разработала систему учета для университетской библиотеки. Но я ощущала себя сбитой с толку. Мне казалось, что программная инженерия — это что-то исключительно про разработку и написание кода.

После выпуска, в начале 2000-х, я работала в ИТ-отделе крупного автомобильного концерна. Как и следовало ожидать, мы занимались программной инженерией. Именно там я впервые (и далеко не в последний раз) столкнулась с диаграммой Ганта и водопадной моделью разработки. Я увидела, что команды все основное время уделяют сбору требований и проектированию и в гораздо меньшей степени — написанию кода, что, само собой, оборачивалось затратами на тестирование, на которое оставалось не слишком много времени.

Казалось, что программная инженерия — это создание приложений, полезных нашим клиентам.

Как многих других разработчиков, меня это не совсем устраивало.

Я изучила экстремальное программирование и Scrum. Я хотела работать в команде, применяющей agile-подход, поэтому в поисках этого подхода сменила несколько компаний. Многие из них утверждали, что они «эджайл», но по факту все ограничивалось тем, что мы записывали требования или задачи на карточках и развесивали их на стене, называя неделю *спринтом*. Разработчики должны были в конце каждого спрингта закрывать определенное количество карточек, чтобы уложиться в произвольно заданные сроки. Очевидно, что отход от традиционного представления о программной инженерии тоже не работал.

Имея за плечами десятилетний опыт разработчика, я пришла на собеседование на Лондонскую финансовую биржу. Руководитель отдела разработки рассказал мне, что они используют методы экстремального программирования, разработку через тестирование (*test-driven development, TDD*), а также парное программирование. По его словам, они занимались чем-то вроде *непрерывной доставки* (*continuous delivery*), которая представляла собой непрерывную интеграцию продукта параллельно с его разработкой.

Я работала до этого в крупных инвестиционных банках, где развертывание занимало минимум 3 часа, а необходимые инструкции и списки выполняемых вручную команд для «автоматизации» занимали 12 страниц. Непрерывная доставка — это звучало здорово только в теории.

Руководителем отдела разработки был Дэйв Фарли, и когда я пришла в компанию, он трудился над книгой *«Continuous Delivery»*¹.

Мы проработали вместе 4 года, которые действительно изменили мою жизнь и карьеру. Мы на самом деле занимались парным программированием, TDD и непрерывной доставкой. Я узнала, что такое разработка через поведение, автоматизированное приемочное тестирование, предметно-ориентированное проектирование, разделение ответственности, уровни защиты от коррупции, что значит почувствовать машину и какие бывают уровни абстракции.

Я научилась создавать высокопроизводительные приложения с низкой задержкой на Java. Я наконец-то поняла, что означает нотация «О-большое» и как она применяется на практике. В общем, я реально использовала все те знания, которые получила в университете и которые почерпнула из книг.

Причем эти знания на самом деле помогли создать качественный высокопроизводительный продукт, который делал что-то новое. Ко всему прочему, мы были счастливы, и работа приносила удовлетворение. Мы не задерживались допоздна, у нас не было авралов перед релизами. А код не усложнялся со временем и его было легко поддерживать. Мы регулярно и последовательно внедряли новые функции и увеличивали ценность для бизнеса.

¹ На русском языке книга была издана под названием «Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ». Общепринятым при этом является термин «непрерывная доставка». — *Примеч. пер.*

Как нам все это удалось? Мы использовали методы, которые Дэйв описал в своей книге. Тогда они еще не были оформлены в том виде, в каком представлены здесь. Дэйв обобщил свой успешный опыт, полученный в различных организациях, и сформулировал концепции, которые можно применять в различных областях бизнеса при решении широкого спектра задач.

То, что работает для нескольких объединенных команд в высокопроизводительной среде биржи, вероятно, не будет в точности таким же при подготовке проекта для крупного промышленного предприятия или для быстрорастущего стартапа.

Как developer-адвокат, я общаюсь с сотнями разработчиков из разных компаний и сфер бизнеса, и они рассказывают мне о своих болевых точках (многие из которых даже сейчас не очень отличаются от тех, что я наблюдала 20 лет назад) и об историях успеха. Концепции, которые Дэйв сформулировал в книге «Современная программная инженерия», достаточно общие, чтобы их можно было применить во всех этих сферах, и достаточно конкретные, чтобы приносить практическую пользу.

По иронии судьбы называться *инженером-разработчиком* мне стало не-комфортно именно тогда, когда я покинула команду Дейва. Я не считала, что то, что мы делаем как разработчики, — это инженерия. Я не думала, что именно она — залог успеха нашей команды. Я считала инженерию слишком структурированной дисциплиной для построения сложных систем. Мне нравилось считать ее своего рода ремеслом — это понятие сочетает креативность и продуктивность, даже если командная работа, которая необходима для разработки ПО, при этом отходит на второй план. Но эта книга изменила мое мнение.

Дэйв наглядно объясняет, почему мы неверно понимаем, что на самом деле представляет собой инженерия. Он показывает, что инженерия — это научная дисциплина, но она необязательно должна быть застывшей. Он демонстрирует, как научные принципы и инженерные техники применяются в процессе разработки программных продуктов, и поясняет, почему техники, которые мы считали инженерными, на самом деле не годятся для разработки.

Больше всего мне нравится в этой книге то, что те концепции, которые казались абстрактными и сложными для использования в реальном коде, Дэйв представляет как прикладные инструменты, которые можно применять для решения конкретных задач.

Книга показывает, в какой сложной реальности приходится действовать разработчику или, не побоюсь сказать, программному инженеру — здесь нет ни одного правильного ответа. Мир меняется. То, что было верно когда-то, со временем может стать абсолютно ложным.

Практические советы первой половины книги — о том, как не только выжить в этой реальности, но и преуспеть в ней. Вторая половина книги, материал которой может кому-то показаться чересчур сложным или академичным, посвящена тому, как применять рассмотренные понятия, чтобы создавать лучший код, например, более устойчивый и легко поддающийся изменениям.

Говоря о создании, я совершенно не имею в виду страницы кода или конкретные диаграммы, а скорее то, как всесторонне продумать код, прежде чем его написать. Когда я работала вместе с Дэйвом, я заметила, как мало времени он тратит на то, чтобы ввести сам код. Оказывается, если как следует продумать код, прежде чем писать его, мы сэкономим огромное количество времени и сил.

Дэйв не пытается избежать возможных противоречий, которые случаются при комбинировании описанных методов, или ошибок, которые могут возникнуть, если использовать какой-то метод по отдельности, либо оправдать противоречия. Наоборот, он разбирает возможные области риска и необходимые компромиссы в работе. Благодаря этому я поняла, что в основе самых успешных решений лежит именно баланс используемых подходов.

Важно рассматривать описанные подходы как некое руководство, понять их преимущества и недостатки и использовать их, чтобы глубже рассмотреть свой код/дизайн/архитектуру и увидеть в них не только черное и белое, правильное или неправильное.

«Современная программная инженерия» помогла мне понять, почему мы чувствовали себя успешными и удовлетворенными все то время, пока работали с Дэйвом. Я надеюсь, что опыт и советы, которые вы получите, прочитав эту книгу, помогут вам так же, как если бы Дэйв Фарли трудился в вашей команде.

Удачи в разработке!

*Триша Джи (Trisha Gee),
developer-адвокат и Java Champion*

Введение

Основная цель книги — показать, что *разработка программного обеспечения* — это все-таки *инженерная* дисциплина. Я описываю практический подход к разработке, который подразумевает намеренно рациональный научный стиль мышления для решения задач. Эта идея сформировалась у меня на основе последовательного применения знаний о разработке за последние несколько десятилетий. Мне хочется убедить вас, что инженерия в действительности отличается от ваших представлений о ней и ее можно успешно и эффективно использовать при создании программных продуктов. Я расскажу об основах инженерного подхода к разработке и поясню, почему он работает.

Мы будем рассматривать не последние технологические веяния, а доказанные практические подходы, когда выводы о том, что работает, а что нет, основаны на данных.

Эффективная работа подразумевает итерации и деление процесса на ряд мелких этапов. Организуя работу в виде последовательности небольших неформальных экспериментов и анализируя данные, полученные посредством обратной связи, мы даем себе больше свободы для исследования задачи и возможных областей решений. Структурирование работы таким образом, когда каждая часть четкая, понятная и посвящена какой-то одной цели, позволяет двигаться вперед более уверенно, даже если в начале пути мы еще точно не знаем, к чему придем.

Такой подход ориентирует нас, на чем и когда следует остановиться, даже если мы не знаем ответов. Он повышает наши шансы на успех независимо от рода трудностей, которые мы преодолеваем.

В этой книге я определяю модель самоорганизации, необходимой, чтобы создавать лучший продукт, а также то, как с одинаковой эффективностью строить и простые, и гораздо более сложные системы.

Всегда будет кто-то, кто делает превосходный продукт. Мы движемся вперед благодаря первоходцам, которые показывают нам наши возможности. В последние годы, однако, мы научились лучше объяснять, что действительно работает. Теперь мы яснее понимаем, какие идеи являются более общими и могут применяться более широко. И теперь у нас есть данные, положенные в основу этого знания. Мы можем создавать продукты быстрее, лучше и увереннее — и у нас есть фундамент для этого. Мы решаем глобальные сложные задачи, полагаясь на опыт многих успешных проектов для многих компаний.

Наш подход включает ряд важных базовых идей и построен на результатах проделанной работы. Он не подразумевает никаких новых практик, однако объединяет важнейшие из них в единое целое и формулирует принципы, лежащие в основе программной инженерии.

Это не случайная коллекция разрозненных идей — идеи тесно связаны друг с другом, дополняют и усиливают друг друга. Если применять их последовательно к тому, как мы организуем и выполняем свою работу, они оказывают значительное влияние на ее эффективность и качество. Хотя каждая идея по отдельности нам знакома, вместе они образуют совершенно новый способ осмыслиения наших действий. Как руководящие принципы для принятия решений, эти идеи формируют новую парадигму разработки.

Мы знаем, что в действительности означает программная инженерия. И это не всегда то, что мы ожидаем.

Инженерия предполагает адаптацию научного рационалистического подхода к решению практических задач с учетом экономических условий, но это не значит, что подобный подход является теоретическим или бюрократическим. По определению инженерия pragmatична.

Все прежние определения программной инженерии были чересчур пре-скриптивными, они базировались на перечне конкретных инструментов и технологий. Инженерия — это больше, чем код, который мы пишем, и инструменты, которые мы используем. Это не технологическое проектирование, и технология — это не наша задача. Если, когда я произношу слово «инженерия», вы начинаете думать о чем-то бюрократическом, перечитайте эту книгу.

Программная инженерия — это не то же самое, что computer science, хотя мы часто их путаем. Нам нужны как инженеры ПО, так и специалисты

в области теории. Эта книга рассказывает о дисциплинах, процессах и идеях, которые нужно применять, чтобы постоянно создавать лучший продукт.

Чтобы наши ожидания оправдались, инженерия должна помогать лучше и эффективнее справляться со стоящими перед нами проблемами.

Инженерный подход также помогает решать задачи, о которых мы еще не задумывались, и использовать технологии, которые еще не изобретены. Концепции этой дисциплины должны быть общими, долговременными и устойчивыми.

Эта книга — попытка собрать воедино тесно связанные идеи. Я стремлюсь сформировать из них теорию, которую мы можем использовать при принятии любого решения как в индивидуальной, так и в командной разработке.

Программная инженерия как концепция призвана давать нам преимущества, а не только возможности адаптирования новых инструментов.

Все идеи разные. Какие-то лучше, какие-то хуже, так как нам их различить? Что нам делать, чтобы оценить новую идею, касающуюся разработки, и понять, хорошая она или нет?

Все, что можно с достаточной степенью уверенности определить как инженерный подход к проблемам разработки, можно положить в основу практических действий и применить в различных сферах. Книга — об этом. По каким критериям следует выбирать инструменты? Как организовывать свою работу? Как выстроить систему, над которой вы работаете, и код, который вы пишете, чтобы ваш продукт был успешен?

ОПРЕДЕЛЕНИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

В этой книге мы будем придерживаться следующего определения:

Программная инженерия — это эмпирический научный подход к поиску эффективных, экономичных решений практических задач при разработке ПО.

Моя цель амбициозна. Я хочу предложить каркас, структуру, подход, который мы могли бы считать истинной инженерной дисциплиной в сфере разработки ПО. В основе лежат три ключевые идеи.

- Теоретическая наука и инженерия как ее практическое применение — жизненно важные инструменты для достижения эффективного технического прогресса.
- Наша дисциплина подразумевает изучение и совершение открытий, поэтому для успешной работы мы должны стать **экспертами в познании**. Теория и практика инженерии включает понятие об эффективном обучении.
- Наконец, системы, которые мы создаем, обычно очень сложны, и их сложность со временем все возрастает. Поэтому чтобы успешно их разрабатывать, нам необходимо стать **экспертами в управлении сложностью**.

СТРУКТУРА КНИГИ

В первой части «Что такое программная инженерия» я разбираю, что действительно представляет собой инженерия в контексте программного обеспечения. Здесь описаны принципы и философия инженерии, а также то, как применять эти идеи в разработке ПО. Это технические основы разработки программного обеспечения.

Во второй части «Оптимизация для обучения» я расскажу, как организовать свою работу, чтобы она была успешной. Как определить, действительно ли мы движемся вперед или всего-навсего создаем то, что завтра уже устареет?

Третья часть «Оптимизация для управления сложностью» содержит описание принципов и методик, необходимых для управления сложностью. Здесь я разберу эти принципы более подробно и по существу, а также покажу, как применять их для создания высококачественных продуктов в любой сфере.

В заключительной, четвертой части «Инструменты программной инженерии» я описываю идеи и подходы к работе, используя которые гораздо проще обучаться и поступательно совершенствоваться, а также управлять сложностью систем по мере их роста и развития.

Кроме того, книга содержит ссылки к истории и философии программной инженерии, описывающие, как менялось и развивалось мышление. Эти врезки я задумал как соответствующий контекст для идей, описанных в книге.

Благодарности

Работа над книгой, подобной этой, отнимает много времени и сил и требует глубокого погружения в материал. Люди, которые оказывали мне поддержку, помогали по-разному: иногда соглашаясь со мной и моими убеждениями, а иногда споря и побуждая находить более убедительные аргументы или менять точку зрения.

В первую очередь я бы хотел поблагодарить свою жену Кейт за ее всемерную помощь и поддержку. Хотя она не профессионал в сфере разработки, она прочитала основную часть книги, исправляя грамматические ошибки и оттачивая слог.

Я благодарен моему родственнику Бернарду Маккарти (Bernard McCarty) за идеи в рамках научного подхода и за то, что помог мне задуматься, почему я стремлюсь уделять столько же внимания экспериментам и наблюдениям, сколько остальным вещам.

Я благодарен Трише Джи не только за отличное вступительное слово, но и за то, что она поддерживала во мне энтузиазм, когда я в этом нуждался.

Я благодарен Мартину Томпсону (Martin Thompson) за обмен мнениями в сфере computer science и за то, что он всегда мгновенно отзывался на мои зачастую спонтанные мысли.

Я благодарен Мартину Фаулеру (Martin Fowler), который, несмотря на чрезвычайную занятость своими проектами, дал мне несколько советов — они позволили заметно улучшить эту книгу.

Многие мои друзья: Дэйв Хаунслоу, Стив Смит (Steve Smith), Крис Смит (Chris Smith), Марк Прайс (Mark Price), Энди Стюарт (Andy Stewart), Марк Краузер (Mark Crowther), Майк Баркер (Mike Barker) и другие — на протяжении нескольких лет помогали мне формировать свой взгляд на эту тему и не только на нее.

Я благодарен команде издательства Pearson за их помощь и поддержку в публикации книги.

Я также благодарен множеству людей — и не всех из них я знаю лично, — кто поддерживал, спорил, заставлял задуматься. Несколько лет я черпал их идеи в Twitter и на своем канале YouTube, где мне удалось наладить несколько очень продуктивных диалогов. Спасибо вам!

Об авторе

Дэвид Фарли — пионер в области непрерывной доставки, лидер мнения и эксперт-практик в сфере непрерывной доставки, автоматизации технологических процессов, разработки через тестирование и общей разработки ПО.

Начав карьеру на заре эры современных компьютерных вычислений, Дэйв многие годы работал программистом, инженером ПО, системным архитектором, а также руководил работой успешных команд. Он использует фундаментальные принципы работы компьютеров и программного обеспечения и прорывные инновационные подходы, трансформирующие современную разработку. Он меняет традиционный подход к мышлению, и под его руководством команды создают продукты мирового класса.

Дэвид — соавтор книги «*Continuous Delivery*», получившей премию Jolt Award, постоянный участник конференций и автор успешного и популярного YouTube-канала Continuous Delivery, посвященного программной инженерии. Он — разработчик одной из самых быстрых в мире финансовых бирж, пионер разработки через поведение, автор Манифеста реактивных систем и лауреат премии Duke Award за создание открытого продукта с использованием LMAX Disruptor.

На своем канале, на курсах и консультациях Дэйв увлеченно делится опытом и дает советы командам разработчиков из разных стран мира, помогая им улучшать дизайн и качество и повышать надежность своих продуктов.

Twitter: @davefarley77

YouTube-канал: <https://bit.ly/CDonYT>

Блог: <http://www.davefarley.net>

Сайт компании: <https://www.continuous-delivery.co.uk>



ЧТО ТАКОЕ ПРОГРАММНАЯ ИНЖЕНЕРИЯ

ГЛАВА 1

ВВЕДЕНИЕ

ИНЖЕНЕРИЯ – ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ ТЕОРЕТИЧЕСКОЙ НАУКИ

Разработка программного обеспечения представляет собой процесс исследования и открытий, поэтому успешные разработчики должны быть экспертами **в познании**.

Человечество разработало самый эффективный способ познания — науку, поэтому к нашим задачам нужно применять научные техники и методы. Но это не значит, как зачастую ошибочно полагают, что следует становиться физиками и проводить всевозможные измерения, добиваясь чрезмерного с точки зрения разработки уровня точности. Инженерия более практична.

Говоря о том, что необходимо применять научные техники и стратегии, я имею в виду базовые, но тем не менее важнейшие идеи.

Принципы научного метода познания, которые большинство из вас изучали еще в школе, в Википедии описаны так.

- **Описание.** Понаблюдайте за текущим состоянием и опишите его.
- **Гипотеза.** Создайте теорию, которая может объяснить ваше наблюдение.
- **Предположение.** Сформулируйте предположение на основе гипотезы.
- **Эксперимент.** Протестируйте свое предположение.

Когда мы организуем свое мышление и развиваем идеи на основе небольших неформальных экспериментов, мы снижаем риск того, что наши выводы окажутся неверными, и в конце концов повышаем вероятность успеха.

Если мы в своих мысленных экспериментах начнем учитывать значение переменных, чтобы добиться большей надежности и последовательности результатов, мы скатимся к детерминантной организации системы и кода. Если мы начнем подвергать свои идеи сомнению и искать возможные способы того, как их опровергнуть, нам удастся быстрее обнаруживать и отсекать плохие идеи, а значит, быстрее добиваться успеха.

Эта книга посвящена практическому, прагматическому подходу к решению задач разработки, основанному на неформальном применении базовых научных принципов; другими словами, она описывает **инженерию**.

ЧТО ТАКОЕ ПРОГРАММНАЯ ИНЖЕНЕРИЯ?

Мое рабочее определение программной инженерии, которое соотносится с идеями книги, таково: *программная инженерия — это эмпирический научный подход к поиску эффективных, экономичных решений практических задач разработки ПО*.

Применение инженерного подхода к разработке важно по двум причинам. Во-первых, разработка — это всегда исследование и изучение, а во-вторых, если наша цель — эффективность и экономичность, наши способности к обучению должны быть устойчивыми.

Это означает, что мы должны управлять сложностью создаваемых систем таким образом, чтобы поддерживать способность изучать новое и приспособливаться к нему.

Поэтому нам необходимо стать **экспертами в познании и в управлении сложностью**.

Чтобы стать *экспертами в познании*, нам придется освоить пять техник:

- итерации;
- сбор и анализ обратной связи;
- инкрементализм;

- экспериментирование;
- эмпиризм.

Это слагаемые эволюционного подхода к созданию комплексных систем. Подобные системы не являются исключительно плодами нашего воображения, они — результат продвижения вперед небольшими шагами, когда мы пробуем идеи и реагируем на результат их реализации. Это те инструменты, с помощью которых мы осуществляем исследования и делаем открытия.

Такой способ организации работы подразумевает и некие ограничения. Нам необходимо максимально облегчить возможность исследования, лежащего в основе каждого проекта.

Поэтому хотя в основе наших действий лежит познание, мы должны научиться добиваться успеха, даже когда ответы, а иногда и направление движения нам неизвестны.

Поэтому нам необходимо стать **экспертами в управлении сложностью**. Независимо от природы задачи, которую мы решаем, или технологий, которые мы используем при ее решении, основное отличие плохих систем от хороших — сложность при решении задачи.

Чтобы стать *экспертами в управлении сложностью*, нам понадобится применять следующие подходы:

- модульность;
- связность (cohesion);
- разделение ответственности (separation of concerns);
- абстракцию;
- слабую связанность, или сцепление (loose coupling).

Наверняка все они вам уже знакомы. Цель данной книги — свести их в единый комплексный подход к разработке систем, чтобы воспользоваться всеми их преимуществами.

В книге рассказано, как преобразовать эти подходы в инструменты разработки. А теперь я перечислю несколько практических инструментов для повышения эффективности любого процесса разработки:

- тестируемость (testability);
- развертываемость (deployability);
- скорость;
- управление переменными;
- непрерывная доставка.

Если вы начнете использовать в работе подобные техники, подходы и инструменты, результаты вас впечатлят. Вы сможете создавать продукты превосходного качества — и гораздо быстрее. А разработчики из команд, применяющих эти принципы, утверждают, что работа приносит им больше удовлетворения, они испытывают меньше стресса и им удается соблюдать баланс между личной жизнью и работой¹.

И все это основано на данных.

ПЕРЕОСМЫСЛЕНИЕ ПОНЯТИЯ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Я буквально выстрадал название этой книги. Не потому, что я не знал, как ее озаглавить, а потому, что в нашей сфере понятие инженерии настолько трансформировалось, что приобрело почти противоположное значение.

В разработке этот термин часто употребляют как простой синоним кодирования либо, наоборот, для обозначения чрезмерной бюрократии или следования формальным процедурам. Но это не имеет ничего общего с настоящей инженерией.

В других сферах инженерия означает «что-то, что работает». Это те процессы и методы, которые вы используете, чтобы добиться нужного результата.

Если при использовании методов того, что мы называем программной инженерией, нам не удается разработать лучший продукт в более быстрые сроки, значит, это не инженерные методы и их следует изменить, — вот основная идея данной книги.

¹ На основании данных отчетов State of DevOps и отчетов Microsoft и Google.

Именно такой идеей я руководствовался при создании последовательной модели, которая способна объединить основные принципы и позволяет надеяться на успех разработки.

Конечно, это не гарантия успеха. Но применяя эти интеллектуальные инструменты и организующие принципы в своей работе, вы значительно повысите его вероятность.

Как добиться успеха

Разработка ПО сложна и многогранна. В некотором смысле это одно из самых сложных занятий. Наивно предполагать, что каждый разработчик или команда могут — и должны — каждый раз, приступая к новой задаче, заниматься разработкой с нуля.

Мы знаем и продолжаем узнавать инструменты, которые работают и которые не работают. Так как же нам — всей отрасли и отдельным командам — добиваться успеха, стоя на плечах гигантов, как сказал однажды Исаак Ньютона, если каждый может пренебречь всем, чем сочтет нужным? Для эффективной деятельности нам нужны определенные руководящие принципы.

Опасность этого подхода в том, что при неправильном применении он может породить стиль мышления и поведения, основанный на поклонении авторитетам.

Неэффективный способ организации работы — когда менеджеры и руководители считают своей непосредственной обязанностью указывать каждому, что и как делать.

Предписывающий, или директивный, стиль управления возникает, когда мы видим, что какие-то идеи ошибочны или неполны. А такие обязательно будут, поэтому нам необходимо понять, как отказываться от старых, но прочно укоренившихся плохих идей и как отыскивать инновационные проекты с большим потенциалом.

Существует прекрасный подход, позволяющий решать эту задачу. Он подразумевает интеллектуальную свободу отказа от догм и разграничение популярных и простых, но плохих решений и действительно стоящих. Такой подход позволяет нам заменять плохие идеи лучшими и улучшать хорошие. Наша основная потребность — это структура, которая позволит нам расти, развивать и совершенствовать подходы, стратегии, процессы, технологии и решения. Этот прекрасный подход называется *наукой*.

А использование такого стиля мышления для решения практических задач называется *инженерией*.

Моя книга — о том, что значит в разработке применять научный стиль мышления и таким образом достичь чего-то, что мы можем искренне и точно назвать *программной инженерией*.

ЗАРОЖДЕНИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Понятие **программной инженерии** сформировалось в конце 1960-х годов. Этот термин был впервые использован Маргарет Хэмилтон (Margaret Hamilton), которая позже стала руководителем подразделения программной инженерии в Инструментальной лаборатории Массачусетского технологического института. Маргарет возглавляла разработку программного обеспечения для контроля полетов в рамках космической программы «Аполлон».

В это же самое время в Гармиш-Партенкирхене, Германия, в попытках дать определение новому термину была создана конференция под эгидой Североатлантического альянса НАТО. Она стала первой конференцией по **программной инженерии**.

Первые компьютеры программировались переключением реле или даже жестким кодированием. Довольно быстро стало понятно, что это очень неудобно и небыстро. Так возникла идея хранимых в памяти программ и появилось четкое деление на программную и аппаратную части.

К концу 1960-х годов компьютерные программы достаточно усложнились. Их стали использовать для решения более сложных задач, и они быстро зарекомендовали себя как единственное возможное средство для решения целых классов задач.

Однако ощущался значительный разрыв между скоростью развития аппаратной и программной частей. Он получил название *кризиса программного обеспечения*.

Конференция под эгидой НАТО была посвящена в том числе и поиску решения этой проблемы.

На этой конференции был высказан ряд долгосрочных идей. Они выдержали испытание временем и сейчас так же актуальны, как и в 1968 году.

И этот факт нам интересен с точки зрения поиска неких фундаментальных основ нашей дисциплины.

Спустя несколько лет лауреат премии Тьюринга Фред Брукс (Fred Brooks) сравнил степень развития программных и технических средств:

Ни в одной технологии или управленческой технике не существует универсального метода, увеличивающего на порядок производительность, надежность и простоту¹.

Брукс исходил из знаменитого закона Мура² для долговременного развития аппаратных средств.

Это интересное наблюдение, которое, полагаю, удивит многих, но по сути своей оно абсолютно верно.

Брукс продолжает и утверждает, что это не проблема собственно разработки, а результат уникального, ошеломляющего развития производительности аппаратных средств:

Следует считать необычным не то, что прогресс в программировании идет так медленно, а то, что он так быстр в аппаратном обеспечении компьютеров. Ни одна другая технология за всю историю цивилизации не имела за 30 лет своего развития роста соотношения “производительность/цена” на шесть порядков.

Брукс написал эти слова в 1986 году, как мы сейчас скажем, на заре компьютерной эры. Развитие аппаратных средств шло своими темпами, и компьютеры, казавшиеся такими мощными во времена Брукса, по сравнению с современными системами выглядят игрушечными. И все же его наблюдения, касающиеся темпов развития в области разработки, остаются актуальными.

¹ Источник: Брукс Ф. Серебряной пули нет. 1986.

² В 1965 году Гордон Мур предсказал, что плотность размещения (не производительность) транзисторов будет удваиваться каждый год. Затем он пересмотрел свой прогноз до двух лет на следующее десятилетие, до 1975 года. Этот прогноз был взят за основу производителями полупроводников и значительно превзошел ожидания Мура, сбывшись десятилетиями позже. Некоторые исследователи считают, что мы приближаемся к завершению взрывного роста мощностей, поскольку современные подходы исчерпали себя, а также из-за влияния эффекта квантования. Но на то время развитие полупроводников шло в соответствии с законом Мура.

МЕНЯЯ ПАРАДИГМУ

Идея *смены парадигмы* была разработана физиком Томасом Куном (Thomas Kuhn).

Познание зачастую подобно приращению. Мы добавляем все новые уровни понимания и каждый последующий основываем на предыдущем.

Однако познание не всегда бывает таким. Иногда мы радикальным образом меняем свои взгляды на изучаемое явление. Это позволяет нам постигать новое. Но это также означает, что мы должны отринуть старое.

В XVIII столетии авторитетные биологи (тогда их так еще не называли) считали, что некоторые виды животных способны к самозарождению. В середине XIX века Ч. Дарвин описал процесс естественного отбора, и это окончательно поставило крест на теории самозарождения.

Такие изменения мышления привели к современному пониманию генетики и способности познавать жизнь на фундаментальном уровне, создавая технологии, которые позволяют управлять генами, разрабатывать вакцины от Covid-19 и генетические лекарства.

Подобным образом Кеплер, Коперник и Галилей отказались от привычного на тот момент представления о том, что Земля является центром Вселенной. Они предложили гелиоцентрическую модель Солнечной системы. Благодаря ей Ньютона сформулировал законы тяготения, Эйнштейн создал общую теорию относительности, а мы совершаем космические перелеты и разрабатываем такие технологии, как GPS.

Смена парадигмы предполагает отказ от тех идей, которые, как нам стало известно, не соответствуют истине.

Абсолютно верно рассматривать разработку как истинно инженерную дисциплину, основанную на философии научного метода познания и научного рационализма.

Это верно не только благодаря ее влиянию и эффективности, так красноречиво описанным в книге «Ускоряйся!»¹, но и благодаря насущной потребности отбрасывать идеи, которые не удовлетворяют подходу.

¹ Форсгрен Н., Хамбл Дж., Ким Дж. Ускоряйся! Наука DevOps: Как создавать и масштабировать высокопроизводительные цифровые организации.

Это позволяет нам более эффективно обучаться и отбирать только стоящие идеи.

Я убежден, что подход к разработке, который я описал в этой книге, делает возможной именно такую смену парадигм. Он дает возможность по-новому взглянуть на то, что мы делаем и как мы это делаем.

ИТОГИ

Инженерное мышление необязательно подразумевает чрезмерную сложность или поклонение авторитетам. Смена парадигмы, когда мы размышляем, что мы делаем и как мы это делаем, в процессе разработки позволит нам увидеть главное за мелочами и сделать разработку проще, надежнее и эффективнее.

Инженерия — это не чрезмерный бюрократизм, это более полная возможность создавать высококачественный продукт, устойчивый и надежный.

ГЛАВА 2

ЧТО ТАКОЕ ИНЖЕНЕРИЯ?

Я разговариваю с людьми о программной инженерии уже несколько лет. В результате я регулярно оказываюсь втянут в дискуссии о строительстве мостов. Обычно они начинаются с фразы: «Да, но программы не строят мосты», — как будто это какое-то откровение.

Конечно, разработка программного обеспечения — это не то же самое, что строительство мостов, но то, что большинство разработчиков ПО считают строительством моста, на самом деле им тоже не является. Подобные разговоры действительно возникают из-за смешения понятий производственной инженерии и инженерного проектирования.

Производственная инженерия — сложная сфера: вам нужно создавать материальные вещи с определенным уровнем точности и качества.

Вам нужно, чтобы созданный продукт был доставлен в определенное место, в определенное время, за определенные деньги и так далее. По мере того как вы адаптируете теоретические идеи к практической реальности, выясняется, что ваши модели и проекты нуждаются в доработке.

Цифровые активы — это совершенно иное. Хотя существуют некоторые аналоги этих проблем, для цифровых объектов таких проблем либо не существует, либо их можно легко упростить. Стоимость производства любых цифровых активов, по сути, равна нулю или по крайней мере должна быть равной нулю.

ПРОИЗВОДСТВО НЕ НАША ПРОБЛЕМА

В деятельности людей самое трудное — это производство вещей. Разработка автомобиля, авиаляйнера или мобильного телефона, конечно же,

требует усилий и изобретательности, но внедрение прототипа в массовое производство — чрезвычайно дорогая и сложная задача.

Это особенно верно, если мы хотим к тому же сделать это производство эффективным. В результате мы, порождение индустриального века и индустриального мышления, автоматически и почти бездумно в первую очередь беспокоимся именно о производственном аспекте любой крупной задачи.

В разработке ПО это вылилось в то, что мы довольно последовательно пытались применять производственный стиль мышления. Водопадные процессы¹ — это своего рода производственная линия, инструменты массового производства, а не инструменты для открытий, познания и экспериментов, то есть того, что должно лежать в основе нашей профессии.

Если только мы не ошиблись с ее выбором, производством для нас должен стать запуск сборки (билда)!

Это полностью автоматический, запускаемый одним нажатием кнопки и широко масштабируемый процесс, который настолько дешев, что его можно считать бесплатным. Да, мы все еще совершаляем ошибки, но эти ошибки понятны и эффективно решаются с помощью инструментов и технологий.

Нам не нужно беспокоиться о производстве, и это то, что делает нашу сферу непохожей на другие, а еще приводит к недопониманию иискаженному представлению о разработке, поскольку подобная простота производства сбивает с толку.

ИНЖЕНЕРИЯ ПРОЕКТНАЯ, А НЕ ПРОИЗВОДСТВЕННАЯ

Даже в реальном мире то, что большинство людей считает строительством мостов, будет отличаться, если вы начнете строить мост нового типа. В этом случае у вас возникнут две проблемы, одна из них будет иметь отношение к разработке, а вторая нет.

¹ Водопад применительно к разработке программного обеспечения — это поэтапный последовательный подход к организации работы, когда рабочий процесс разбивается на ряд последовательных этапов с четко заданными границами. Идея заключается в том, что вы работаете над каждым этапом по очереди, а не выполняете итерации.

Итак, проблема, которая не относится к разработке, такова: при строительстве нового типа мостов вы сталкиваетесь со сложностями производства и другими трудностями, которые я описал выше, поскольку мост — это физический объект. В разработке ПО таких проблем не возникает.

Вторая по-настоящему трудная задача, с которой вы столкнетесь при строительстве нового моста, это собственно его проектирование.

Это сложно, поскольку вы работаете с физическим объектом и не можете быстро его изменять.

Инженеры других отраслей также применяют методы моделирования. Они строят небольшие физические модели, а теперь еще и пользуются компьютерными симуляциями или различными математическими моделями.

А у разработчиков ПО есть огромное преимущество. Мостостроитель может создать компьютерную модель, но она будет лишь приблизительно соответствовать реальной. Она будет неточной. Модели же, которые создают разработчики, их компьютерные симуляции, и есть конечный продукт.

Нам не нужно беспокоиться, соответствуют ли наши модели реальности; наши модели — это реальность нашей системы, поэтому мы можем их проверить. Нам не нужно беспокоиться о стоимости их изменения. Это программы, поэтому их значительно легче изменять, по крайней мере по сравнению с мостом.

Наша дисциплина техническая. Нам нравится думать о себе именно так, и я уверен, что большинство людей, считающих себя профессиональными разработчиками, получили образование в computer science.

Несмотря на это, мало кто использует научно-рационалистический подход в своей работе. Отчасти так происходит потому, что в ходе развития отрасли было совершено несколько ошибок. Во-первых, сложилось мнение, что заниматься наукой трудно, дорого и по-настоящему невозможно в рамках стандартного графика разработки.

Во-вторых, мы ошибочно ожидаем некого уровня идеалистической точности, невозможного ни в одной области, не говоря уже о сфере разработки. Мы ошибаемся, стремясь к математической точности, а ведь она не синоним инженерии!

ИНЖЕНЕРИЯ КАК МАТЕМАТИКА

В конце 1980-х – начале 1990-х годов начались разговоры о более структурном подходе к программированию. Размышления о значении программной инженерии переросли в оценку способов, которыми мы создаем код. В частности, как организовать работу таким образом, чтобы более эффективно выявлять и устранять проблемы в наших моделях и реализациях?

Популярность приобрели формальные методы. В то время их преподавали в рамках большинства университетских программ. Формальный метод – это подход к созданию ПО, подразумевающий встроенную математическую валидацию написанного кода, то есть проверку кода на правильность.

Проблема в том, что сложно написать код для комплексной системы, но еще сложнее – тот, который определяет поведение подобной системы и одновременно подтверждает свою правильность.

Формальные методы – идея привлекательная, но они не получили широкого распространения на практике, поскольку с точки зрения производства они не упрощают, а наоборот, затрудняют процесс создания кода.

Более философский взгляд на проблему оперирует другими аргументами. Программы – вещь необычная. Программирование привлекает людей с математическим складом ума, поэтому применение математического подхода к программированию очевидно, но оно и задает некоторые ограничения.

Проведем аналогию с реальным миром. Современные инженеры используют все доступные им инструменты для создания новых систем. Они разрабатывают модели и симуляции, производят расчеты, чтобы определить, будет ли система работать. На стадии конструирования им не обойтись без математики, но затем они тестируют результаты в реальных условиях.

В других инженерных дисциплинах математика тоже, безусловно, важна, но она не заменяет тестирования и наблюдения за продуктом в реальных условиях. Эти условия слишком вариативны, чтобы их можно было полностью просчитать. Если бы одной математики было достаточно для разработки самолета, то авиастроители ею бы и ограничились, поскольку это гораздо дешевле, чем создавать рабочие прототипы. Но этого не происходит. Авиастроители используют математику для получения данных, а затем проверяют их, тестируя реальное устройство. Программное обеспечение – это не совсем то же самое, что самолет или ракета.

Программа – цифровой инструмент, который работает на детерминированных устройствах – компьютерах. Поэтому для некоторых узких контекстов, если проблема достаточно проста, ограничена, определена и имеет небольшую изменчивость, формальные методы могут быть эффективны.

Сложность здесь представляет определение степени детерминированности всей системы. Если система обладает параллелизмом, взаимодействует с реальным миром (людьми) или работает в чрезвычайно сложных областях, то ее теоретическая эффективность становится недоказуемой.

Поэтому, подобно нашим коллегам из аэрокосмической области, мы применяем математические модели там, где это возможно, и используем основанный на данных, практический, эмпирический, опытный подход к обучению, что позволяет нам адаптировать системы по мере их постепенного роста.

Пока я пишу эту книгу, компания SpaceX совершенствует ракету Starship¹. SpaceX разрабатывает надежные математические модели почти для всех узлов ракеты, двигателей, системы доставки топлива, инфраструктуры запуска и всего остального, а затем испытывает эти модели.

Такая кажущаяся простой вещь, как замена 4-миллиметровой нержавеющей стали 3-миллиметровой, может считаться в значительной степени контролируемым изменением. SpaceX оперирует подробными данными о пределе прочности металла на разрыв, собранными в результате экспериментов: они точно показывают степень прочности емкостей высокого давления, изготовленных из 4-миллиметровой стали.

И все же после того как SpaceX завершила все расчеты, она построила экспериментальные прототипы, чтобы оценить результат. Тестовые экземпляры подвергали воздействию высокого давления до самого разрушения, а затем оценивали точность предварительных расчетов. SpaceX собирала данные и тестировала модели, поскольку в каких-то труднопредсказуемых ситуациях эти модели могли быть ошибочными.

Замечательное преимущество разработки ПО перед остальными инженерными дисциплинами состоит в том, что все модели, которые мы создаем, являются конечным результатом нашей работы. Поэтому когда мы тестируем модель, мы на самом деле тестируем сам продукт, а не лучшее возможное представление о нем.

¹ На момент написания статьи компания SpaceX разрабатывает новый космический корабль многоразового использования. Компания намерена создать систему, которая позволит людям путешествовать на Марс и жить там, а также исследовать остальные части Солнечной системы. Компания применяет намеренно быстрый, итеративный стиль разработки, чтобы получить возможность оперативно создавать и оценивать серию прототипов. Такой подход представляет собой проектную инженерию на пределе знаний и наглядно показывает, какие усилия необходимы, чтобы создать что-то новое.

Если мы выделим часть системы, которая представляет для нас интерес, мы сможем оценить ее работу именно в той среде, в которой она будет в дальнейшем функционировать. Поэтому экспериментальное моделирование в разработке ПО гораздо точнее представляет реальные характеристики систем, чем в любой другой отрасли.

В своем потрясающем выступлении Real Software Engineering (Настоящая программная инженерия)¹ Гленн Вандербург говорит, что в других областях «инженерия — это то, что работает», а в разработке ПО почти все наоборот.

Вандербург исследует, почему это так. Он описывает академический подход к программной инженерии, который был настолько обременительным, что почти никто из тех, кто его практиковал, не рекомендовал его для будущих проектов.

Тяжеловесный подход не добавлял существенной ценности процессу разработки. Вандербург говорит так:

Академический подход работал только потому, что энтузиасты, которым не все равно, были готовы идти в обход процесса.

С какой стороны ни посмотри, это не инженерия.

Определение инженерии как «того, что работает», данное Вандербургом, очень важно. Если практики, которые мы определяем как инженерные, не позволяют нам быстрее создавать лучшее ПО, то инженерными их считать нельзя!

Разработка, в отличие от физических производственных процессов, целиком основана на открытии, изучении и проектировании. Проблемы разработки — это проблемы исследования, и поэтому мы даже больше, чем создатели космических кораблей, должны применять исследовательские методы, а не производственные технологии. Наша дисциплина относится исключительно к проектной сфере.

Итак, если наше понимание инженерии зачастую ошибочно, то что же оно все-таки означает?

¹ <https://youtu.be/RhdIBHHimeM>.

ПЕРВЫЙ ИНЖЕНЕР-ПРОГРАММИСТ

В то время когда Маргарет Гамильтон руководила разработкой системы управления полетом «Аполлона», правил игры еще не существовало. По ее словам, «мы разрабатывали свои инженерные правила на основе каждого открытия, которое мы делали, а высшее руководство НАСА от предоставления полной свободы перешло к чрезмерной бюрократии».

В то время у нас было очень мало опыта реализации таких сложных проектов, поэтому команда часто открывала совершенно новые горизонты. Задачи, стоявшие перед Гамильтон и ее командой, были очень сложными. В 1960-е годы никто не знал об исправлении ошибки переполнения стека.

Вот как Гамильтон описывает некоторые сложности:

Программное обеспечение космической миссии должно быть пригодным для использования человеком. Оно не только должно работать – но работать впервые. Помимо исключительной надежности, оно должно обнаруживать ошибки и выполнять восстановление в реальном времени. Языки, которые мы использовали, позволили нам смоделировать самые трудно обнаруживаемые ошибки. Мы сами придумывали правила разработки. Результаты, которые мы получили после обработки ошибок, зачастую оказывались неожиданными.

В то же время на разработку ПО в целом в сравнении с другими, более «взрослыми», формами инженерии смотрели свысока, как на какого-то бедного родственника. Гамильтон придумала термин «программная инженерия», чтобы коллеги из других дисциплин стали воспринимать разработку более серьезно.

Большое внимание Гамильтон уделяла тому, как совершаются ошибки, каким образом мы их допускаем.

Я с головой увлеклась ошибками. Все свое время я проводила, совершая конкретную ошибку или класс ошибок, изучая их и способы их предотвращения в будущем.

В основе изучения ошибок лежал научно-рациональный подход к решению проблем. Идея заключалась не в том, чтобы спланировать и сделать все правильно с первого раза, а в том, чтобы иметь возможность рассмотреть все варианты, когда что-то пошло не так. Время от времени реальность будет удивлять вас, но именно так работает инженерная эмпирика.

Другой инженерный принцип, сформулированный в ранних работах Гамильтон, – идея безопасного сбоя. Она предполагает, что мы никогда не сможем закодировать каждый сценарий. Тогда каким же образом мы составляем код так, что наша система справляется с неожиданными

сбоями и продолжает успешно работать? Известно, что именно благодаря тому, что Гамильтон реализовала эту необязательную идею, миссия «Аполлон-11» оказалась успешной и лунный модуль Eagle достиг поверхности Луны, хотя компьютер испытывал перегрузку во время посадки.

Во время спуска Нила Армстронга и Базза Олдрэна на поверхность Луны астронавты обменивались данными с центром управления. Когда модуль LEM приблизился к поверхности Луны, компьютер передал сигналы 1201 и 1202. Астронавты спросили, следует ли продолжать снижение или прервать миссию.

НАСА колебалось, пока один из инженеров не прокричал «Продолжаем!», потому что понял, что случилось.

На «Аполлоне 11» каждый раз, когда возникали сигналы тревоги 1201 и 1202, компьютер перезагружался и перезапускал важные системы – управление двигателем спускаемого аппарата или дисплей и клавиатуру, чтобы экипаж знал, что происходит. Но не перезагружал все ошибочные запланированные задачи радио сближения. Ребята из центра управления полетами НАСА знали, – поскольку институт проводил масштабное тестирование перезапуска, – что миссия может продолжаться¹.

Этот безопасный сбой был закодирован в системе без конкретного указания, когда и как он может быть полезен.

Итак, Гамильтон и ее команда внедрили две ключевые черты инженерного мышления: эмпирические познание и открытие и привычку прогнозировать возможность ошибки.

РАБОЧЕЕ ОПРЕДЕЛЕНИЕ ИНЖЕНЕРИИ

Большинство словарных определений слова «инженерия» включает одинаковые слова и фразы: «использование математики», «эмпирические доказательства», «научные рассуждения», «с учетом экономических ограничений».

Я предлагаю такое определение:

Инженерия – это эмпирический научный подход к поиску эффективных и экономичных решений практических задач.

¹ Источник: Peter Adler (<https://go.nasa.gov/1AKbDei>).

Все слова здесь важны. Инженерия — это прикладная наука. Она практически ориентирована. Слово «эмпирический» означает познание и глубокое понимание возможных решений проблемы.

Инженерные решения — это не что-то абстрактное. Они практичны и применимы к проблеме и контексту. Они эффективны и создаются с пониманием заданных экономических условий и их ограничений.

ИНЖЕНЕРИЯ – ЭТО НЕ КОД

Еще одно частое заблуждение: *инженерией* в разработке ПО считают исключительно результат — код или, возможно, его проект.

Это очень узкое толкование. Что инженерия означает для SpaceX? Это не ракеты. Ракеты — это продукт инженерии. Инженерия — это процесс их создания. Конечно, в ракетах есть инженерная составляющая, и безусловно, ракеты — это инженерные конструкции, но мы не считаем инженерией исключительно процесс сварки металла, если только, конечно, не смотрим на процесс слишком узко.

Если мое определение работает, инженерия — это применение рационального научного подхода к решению задач. Это не просто результат сам по себе, а именно решение проблемы. Оно подразумевает процессы, инструменты и техники. А также идеи, философию и подходы, которые все вместе составляют инженерную дисциплину.

Пока я писал эту книгу, мне довелось получить необычный опыт: на своем канале в YouTube я разместил видео о сбоях в играх, которое вышло в топ из всех моих роликов.

Самой частой негативной реакцией, которую я получал на свои слова: «Это была ошибка программной инженерии», — была реакция на то, что я виню в ней программистов, а не их руководителей. Я имел в виду, что ошибка возникает в самом подходе к разработке — планировании, культуре и коде (очевидно, содержащем множество багов).

Поэтому в этой книге, говоря об инженерии, я подразумеваю **все, что имеет отношение к разработке ПО**, если только прямо не уточняю иной смысл. Процессы, инструменты, культура — все это части единого целого.

РАЗВИТИЕ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Самые первые усилия программной инженерии были направлены на создание совершенных языков программирования. В первых компьютерах не было деления на программную и аппаратную части либо оно было ограниченным; программирование происходило путем подключения проводов к коммутационной панели или переключением реле.

Интересно, что эту работу часто поручали «вычислителям» – людям, которые до появления вычислительных машин сами проводили математические расчеты. Зачастую это были женщины.

Это название профессии, однако, преуменьшает ее роль. «Программа», заданная кем-то более важным в организации, в те времена часто выглядела как «нам нужно решить эту математическую задачу». Организация работы, а позже и особенности перевода задачи в соответствующую, воспринимаемую машинами форму оставались за упомянутыми «человеко-машинами». Именно они были настоящими пионерами нашей отрасли.

Сегодня для описания такой работы мы бы использовали другие формулировки. Описание, переданное человеку, выполняющему работу, мы бы назвали требованиями, разработку плана решения задачи – программированием, а «вычислителей» – первыми настоящими программистами этих первых электронных вычислительных систем.

Следующим значительным шагом вперед был переход к «хранимым программам» и их кодирование. Это была эпоха бумажных лент и перфокарт. Первые шаги по внедрению такого носителя информации были довольно сложными. Программы были написаны машинным кодом и хранились на ленте или на карте, а затем их загружали в машину.

Следующим крупным достижением стали высокоуровневые языки, способные выражать идеи на высоком уровне абстракции. Это позволило программистам продвигаться вперед гораздо более быстрыми темпами.

К началу 1980-х годов сформировались почти все основополагающие концепции проектирования языков. Это не значит, что дальше они не развивались, но все основные принципы уже были сформулированы. Тем не менее совершенствование разработки ПО с фокусом на языках программирования продолжалось.

На производительность программистов, безусловно, влияли многие факторы, но только один из них помог или почти помог добиться десятикратного увеличения производительности, о котором говорил Фред Брукс. Это переход от машинного кода к высокоуровневым языкам.

Среди других значительных факторов, повлиявших на развитие отрасли, можно назвать возникновение процедурного, объектно-ориентированного и функционального программирования, однако все они актуальны уже долгое время.

Одержанность нашей отрасли языками и инструментами вредит нашей профессии. Это не значит, что в проектировании языков не было достижений, но большинство работ, посвященных ему, делали акцент, например, на синтаксисе, а не на структурных особенностях, что неверно.

Вначале, безусловно, следовало изучать и исследовать все, что было возможно и что имело для нас значение. Однако огромный объем приложенных усилий дал относительно незначительный прогресс. После того как Фред Брукс заявил, что десятикратное увеличение производительности невозможно, он перешел к рассуждениям, как преодолеть это ограничение:

Первым шагом к лечению болезней стала замена представлений о демонах и "соках" в организме теорией бактерий. Сам этот шаг, обещавший надежду, опроверг все мечты о чудесном исцелении.

... сначала систему надо заставить выполнять, даже если при этом она не делает ничего полезного, кроме вызова некоторого числа фиктивных подпрограмм. Затем она понемногу обрастает "мясом", причем подпрограммы, в свою очередь, разрабатываются сначала как вызовы пустых заглушек, расположенных на уровень ниже.

Эти идеи были основаны на более глубоком и вдумчивом подходе, чем просто языковая реализация, они в основном касались философии нашей дисциплины и применения основополагающих принципов, которые работают для любой технологии.

ПОЧЕМУ ИНЖЕНЕРИЯ ВАЖНА?

Один из способов понять это — рассмотреть способы организации производственных процессов. На протяжении большей части человеческой истории все, что мы создавали, было продуктом ремесленного производства. Это эффективный подход к созданию вещей, но у него есть ограничения.

Ремесло очень хорошо годится для создания единичных предметов. При ремесленном производстве каждый предмет неизбежно будет уникальным. В чистом виде это справедливо для любой производственной системы, но для ремесленной это более верно, поскольку точность и повторяемость производственных процессов в ней очень низки.

Это означает, что индивидуально изготовленные предметы значительно отличаются между собой. Даже самые мастеровитые ремесленники способны создавать изделия лишь с точностью и погрешностью, доступными человеку, и это значительно влияет на способность ремесленных систем к надежному воспроизведению вещей. Грейс Хоппер (Grace Hopper) сказала:

Для меня программирование – это не просто важное прикладное искусство. Это еще и грандиозный фундамент знаний.

ОГРАНИЧЕНИЯ РЕМЕСЛЕННОГО ПРОИЗВОДСТВА

Мы часто эмоционально воспринимаем продукты ремесленного производства. Нам нравится разнообразие, нам приятно ощущение того, что созданная вручную вещь аккумулирует навыки, любовь и заботу своего создателя.

Однако, по сути, подобная продукция обычно низкого качества. Люди, безусловно, талантливы, но не настолько точны, как машины. Аппараты, которые мы создаем, обрабатывают отдельные атомы и даже субатомные частицы, но только выдающийся мастер способен добиться точности 0.1 мм¹, работая вручную.

Что подобная точность означает для разработки ПО? Представьте себе процесс выполнения программы. На восприятие какого-то изменения человеку требуется примерно 13 мс. Обработка изображений или реакция на действие занимает сотни миллисекунд².

На момент написания этой книги большинство современных потребительских компьютеров работает с тактовой частотой около 3 ГГц, это 3 млрд циклов в секунду. Большинство компьютеров содержат несколько ядер и обрабатывают несколько команд одновременно, поэтому за каждый цикл выполняется более одной команды. Давайте для простоты

¹ Атомы различаются размерами, но обычно измеряются десятками пикометров (1×10^{-12} м). Следовательно, лучшие образцы ручного человеческого труда уступают в точности качественным машинным продуктам в 10 млн раз.

² How Fast is Real-time? Human Perception and Technology, <https://bit.ly/2Lb7pL1>.

представим, что каждая команда, которая перемещает значение из одного регистра в другой, добавляет значение или ссылается на какой-то объем памяти в кэше, занимает один цикл.

Это 3 млрд операций в секунду. Если мы рассчитаем, сколько команд современный компьютер способен обрабатывать за абсолютный минимум времени, необходимый человеку, чтобы воспринять какое-либо событие, мы получим число 39 000 000.

Если мы ограничим качество своей работы уровнем человеческого восприятия и точности, мы в лучшем случае добьемся точности передачи, равной 1 : 39 000 000. Итак, каков риск что-то упустить?

ТОЧНОСТЬ И МАСШТАБИРУЕМОСТЬ

Разницу между ремесленным и инженерным способами производства подчеркивают два свойства инженерного подхода, важные для разработки программного обеспечения: точность и масштабируемость.

С точностью все понятно. Благодаря инженерным технологиям мы обрабатываем детали гораздо точнее, чем вручную. Масштабируемость менее очевидна, но более важна. Инженерный подход не ограничивается теми же факторами, что и ремесленный.

Ограничения каждого подхода, который опирается на человеческие способности, в конечном счете определяются непосредственно способностями. Если я захочу достичь чего-то выдающегося, я могу научиться рисовать линии, шлифовать металл или шить кожаные чехлы для автомобильных сидений с точностью до долей миллиметра, но как бы я ни старался, как бы ни была высока потенциальная награда, конечный результат ограничен возможностями моих мышц и органов чувств.

Инженер же строит машину, чтобы создать что-то гораздо более точное и меньших размеров. Мы способны создавать инструменты и машины для производства еще более мелких машин.

Такой подход позволяет масштабироваться от размеров квантовых частиц до космических тел. Не существует ограничений, по крайней мере в теории, которые не дадут нам с помощью инженерии работать с атомами и электронами (что мы уже делаем) или звездами и черными дырами (что мы, вероятно, сможем сделать в будущем).

В контексте программного обеспечения это означает, что после долгих и усердных тренировок мы, возможно, сможем вводить текст и нажимать на клавиши настолько быстро, чтобы успевать тестировать наше ПО ежеминутно.

Представим, например, что мы проводим один тест программы в минуту (теперь, который я, вероятно, не смогу выдерживать очень долго). При такой скорости наше отставание от компьютера составит сотни тысяч или, возможно, даже миллионов раз.

Я разрабатывал системы, выполняющие 30 000 тестов за 2 минуты. Можно и больше, но просто незачем. Google заявляет о 150 млн тестов в день, то есть 104 166 тестов в минуту.

Мы используем компьютеры не только чтобы выполнять операции со скользящей в десятки тысяч раз выше человеческой — мы сохраняем эту скорость, пока к компьютеру подается электропитание. И это масштабирование!

УПРАВЛЕНИЕ СЛОЖНОСТЬЮ

Есть еще один аспект, в котором инженерия обеспечивает масштабирование, в отличие от ремесленного производства. Инженерный способ мышления позволяет структурировать задачи. До Гражданской войны в США 1860-х годов, если вам требовалось оружие, вы шли к оружейнику. Оружейник был мастером и, как правило, мужчиной.

Он изготавливал для вас оружие целиком. Мастер разбирался во всех нюансах этого оружия и делал его специально для вас. Скорее всего, он даже предлагал вам отлить пули, потому что их изготавливали индивидуально для каждого ствола. Если бы у вашей винтовки был крепеж, то каждый винт отличался бы от остальных, поскольку их все создавали вручную.

Гражданская война в США стала уникальной для своего времени. В ходе нее впервые стало применяться оружие массового производства.

Один человек захотел продавать винтовки в северные штаты. Он был новатором и в какой-то степени шоуменом. Он отправился в Конгресс, чтобы получить контракт на поставку оружия в армию северных штатов.

Он взял с собой целый мешок нарезных деталей. Выступая в Конгрессе, он вытряхнул на пол его содержимое и предложил конгрессменам выбрать детали из этой кучи. Из них он собрал винтовку, получил контракт и внедрил массовое производство.

Это был первый случай стандартизации. Чтобы она стала возможной, должно было произойти много событий, например требовалось сконструировать машины и инструменты для создания повторяющихся идентичных деталей с определенной степенью допуска. Конструкция должна была быть модульной, чтобы удавалось собирать компоненты воедино, и так далее.

Результат оказался ошеломляющим. Гражданская война в США была, по сути, первой современной войной. Сотни тысяч людей погибли из-за массового производства вооружения. Такое оружие оказалось дешевле, проще в обслуживании и ремонте, а также точнее, чем прежнее.

Оружие создавалось с большей точностью и в большем количестве. Процесс производства теперь можно было разделить на этапы и масштабировать. Вместо опытных мастеров-ремесленников, производящих оружие поштучно, фабрики, оснащенные оборудованием, использовали менее квалифицированную рабочую силу, чтобы производить винтовки, не уступающие в точности тем, которые создавал мастер.

Позже, с развитием технологий производства и инженерного дела это оружие массового производства стало еще качественнее и производительнее, чем самые выдающиеся экземпляры, созданные вручную. А стоимость его была такой, что его мог купить каждый.

Упрощенно этот процесс можно представить как потребность в стандартизации или во внедрении массового производства ПО. Но такой подход исказит основу нашей задачи. Разработка ПО – это не производство, это проектирование.

Если мы спроектируем ружье, которое состоит из отдельных компонентов и модулей, подобно тому, как это было сделано во время Гражданской войны, то мы сможем независимо разрабатывать отдельные комплектующие этого оружия. С позиции разработки, в отличие от организации производства, мы таким образом улучшим управление сложностью создания оружия.

Прежде мастерам-оружейникам, желавшим изменить какую-то деталь, приходилось иметь дело со всей конструкцией ружья. Разделив его на

составляющие, производители времен Гражданской войны получили возможность внедрять изменения поэтапно, шаг за шагом повышая качество продукции. Как сказал Эдсгер Дейкстра (Edsger Dijkstra):

Искусство программирования — это искусство организации сложного.

ПОВТОРЯЕМОСТЬ И ТОЧНОСТЬ ИЗМЕРЕНИЙ

Еще одна черта инженерного подхода, которая часто обсуждается и иногда используется, чтобы показать, что понятие инженерии неприменимо к разработке ПО, — повторяемость.

Если мы создадим машину, которая точно и аккуратно воспроизводит болты и гайки, мы сможем выбрать любую гайку для любого болта из кучи деталей.

Подобная задача производства неприменима к разработке ПО. Здесь используется другая идея, более глубокая.

Чтобы создавать болты и гайки или любые другие вещи, которые должны подходить друг к другу, нам необходимо измерять предметы с определенным уровнем точности. Точность измерений — это основополагающий принцип инженерии в любой отрасли.

Давайте представим сложный программный комплекс. Допустим, через несколько недель работы в системе произошел сбой. Систему перезагрузили, но через две недели сбой повторился как по шаблону. Что в этом случае сделают те, кто мыслит как ремесленники, и что — кто мыслит как инженеры?

Первые, вероятно, решат, что необходимо более тщательно тестировать ПО. Они оперируют категориями ремесленного подхода, поэтому считают необходимым тщательно исследовать обстоятельство сбоя.

Это довольно разумно и имеет смысл в данной ситуации. Но как это осуществить? Чаще всего в таких случаях проводят так называемый тест на стабильность (soak test). Он проводится за более длительное время, чем обычный интервал между сбоями, допустим, в нашем примере это 3 недели. Иногда этот процесс ускоряют, чтобы симулировать проблему в более короткие сроки, но обычно так не делают.

Итак, проводится тестирование, в системе возникает сбой через 2 недели, ошибку обнаруживают и устраняют.

Существуют ли альтернативы этому способу? Да.

Тестирование стабильности позволяет обнаруживать утечки ресурсов в той или иной форме. Существует два способа: первый — ждать, пока утечка станет заметной, и второй — повысить точность измерений, чтобы обнаружить ее до того, как она станет критической.

Недавно у меня на кухне обнаружилась протечка в забетонированной трубе. Мы нашли течь, когда она стала достаточно заметной и уже натекла лужа воды. Это пример первого способа обнаружения.

Мы вызвали специалиста, чтобы он устранил течь. Он принес инженерное решение — инструмент, который представляет собой высокочувствительный микрофон для обнаружения течи по звуку льющейся воды.

С его помощью по шуму струи воды в бетоне мастер обнаружил течь с точностью, превышающей человеческую. Он выяснил местоположение протечки в пределах нескольких дюймов и выкопал небольшую траншею, чтобы добраться до поврежденного участка трубы.

Вернемся к нашему примеру. Специалисты, использующие инженерный подход, предпочтут точнее проводить измерения, а не ждать, пока случится сбой. Они оценят производительность своего продукта и обнаружат утечки до того, как те станут проблемой.

У такого подхода множество преимуществ. С одной стороны, он сводит вероятность критического сбоя к минимуму, а с другой — позволяет обнаружить индикаторы проблемы и ценные данные о состоянии системы гораздо быстрее. Вместо того чтобы неделями тестировать стабильность, инженерный подход позволяет выявлять утечки в ходе стандартного тестирования системы и получать результат за минуты. Дэвид Парнас (David Parnas) сказал:

Программная инженерия сейчас часто рассматривается как одно из направлений компьютерных наук, подобно тому как химическая инженерия считается частью науки химии. Нам нужны и химики, и химики-инженеры, и это разные специалисты.

ИНЖЕНЕРИЯ, КРЕАТИВНОСТЬ И РЕМЕСЛО

Размышляя об инженерии в целом и о программной инженерии в частности, я несколько лет изучал понятия, вынесенные в заголовок. Я обращался к этой теме в выступлениях на конференциях разработчиков и в статьях в своем блоге.

Иногда мне отвечали сторонники ремесленного подхода к разработке. В их ответах обычно звучала такая мысль: «Отмахиваясь от ремесленного подхода, вы упускаете кое-что важное».

Идеи мастерства очень важны. Они представляют собой важный шаг вперед от формализованного и сосредоточенного на конечном продукте подхода к разработке, применявшегося ранее. Я не утверждаю, что ремесленный подход ошибочен, он скорее недостаточен.

В частности, подобные возражения проистекают от неверных предпосылок, одну из которых я уже упоминал. Многие сторонники ремесленного подхода совершают распространенную ошибку, предполагая, что инженерия служит решению задач производства. Об этом мы уже говорили. Если наша задача — инженерное проектирование, то необходимо учитывать, что это гораздо более исследовательская и креативная сфера по сравнению с производственной инженерией.

Вдобавок мои оппоненты беспокоятся об опасности упустить из виду те преимущества, которые ремесленный подход привнес в разработку:

- навыки;
- креативность;
- свободу инноваций;
- наставничество.

Все они очень важны для эффективной профессиональной разработки. Однако они не ограничены только ремесленным подходом. Он позволил значительно усовершенствовать процесс разработки ПО, используя эти преимущества. Но они утратили свою важность в 1980–1990-х годах, когда популярность набрал руководящий подход к разработке с упором на конечный продукт. Это было неудачное решение, поскольку хотя водопадные процессы и оказались эффективны для понятных, повторяемых и прогнозируемых задач, подход в целом имел слабое отношение к самой сути разработки ПО, если вообще имел.

Ремесленный подход гораздо лучше соотносится с тем, что представляет собой разработка ПО. Но его проблема в том, что предлагаемые им решения не масштабируются в той степени, в какой это могут делать инженерные решения.

Продукты ремесленного производства качественные, но до определенной степени.

Инженерия повышает качество, сокращает затраты и предлагает надежные, устойчивые и гибкие решения практически во всех сферах деятельности.

Отождествлять такие понятия, как навыки, креативность и инновации, исключительно с ремесленным трудом — большая ошибка. Инженеры в целом и в особенности инженеры-проектировщики постоянно демонстрируют эти качества. Они — суть процесса проектирования.

Поэтому применение инженерного подхода к решению задач никоим образом не уменьшает важности навыков, креативности и способности к инновациям. Наоборот, этот подход подчеркивает необходимость этих качеств.

Мне всегда было интересно, считают ли сторонники ремесленного подхода, что молодому инженеру, недавнему выпускнику университета, сразу же можно поручить проектирование нового моста или космического шаттла. Конечно же нет.

В начале своей карьеры все инженеры работают с более опытными коллегами. Они учатся применять теорию на практике и, возможно, даже превзойдут своих учителей.

Я не вижу противоречий между ремеслом и инженерией. Если взглянуть с формальной точки зрения и рассмотреть цеха, учеников, подмастерьев и мастеров, то инженерия — это следующая ступень развития подобной структуры. По мере того как на смену идеям Просвещения XVII–XVIII веков пришел научный рационализм, инженерия добавила к ремеслу дополнительную точность измерений. Инженерия — это более масштабируемая и эффективная разновидность ремесленного подхода.

Если взять более привычное понятие о ремесле (представьте, например, ярмарку изделий народных промыслов), то в нем не предусмотрены стандарты качества или развития. Поэтому инженерия — это скорее прыжок вперед.

Инженерия, в особенности применение инженерного подхода к проектированию, действительно воплощает в себе разницу между нашей высокотехнологической цивилизацией и предшествующим ей аграрным обществом. Инженерия — это отрасль, позволяющая нам работать с чрезвычайно сложными задачами и находить для них изящные, эффективные решения. Применяя принципы инженерного подхода к разработке программного обеспечения, мы добиваемся огромного и измеримого повышения качества, продуктивности и применимости наших решений.

ПОЧЕМУ ТО, ЧЕМ МЫ ЗАНИМАЕМСЯ, – ЭТО НЕ ПРОГРАММНАЯ ИНЖЕНЕРИЯ

В 2019 году компания Илона Маска SpaceX приступила к решению масштабной задачи — созданию космического корабля, который позволил бы людям жить и работать на Марсе, а также исследовать другие части Солнечной системы. В 2019 году компания решила строить корабль Starship не из углеволокна, а из нержавеющей стали. Изначальный выбор углеволокна был достаточно смелым. В компании проделали огромную исследовательскую работу, включая строительство прототипов топливных баков из этого материала. Второй вариант — нержавеющая сталь — тоже был нестандартным. Ракеты в основном изготавливаются из алюминия, поскольку он легкий и прочный.

Выбор нержавеющей стали обусловили три фактора: килограмм этой стали стоит значительно дешевле, она более устойчива к высоким температурам и повторным температурным нагрузкам, чем алюминий, а также сохраняет больше полезных качеств при воздействии низких температур.

Углеволокно и алюминий намного хуже переносят воздействие низких и высоких температур.

Когда вы в последний раз слышали хотя бы приблизительно столь же рациональное обоснование некоего решения, касающегося разработки ПО?

Вот что представляют собой инженерные решения. Они основаны на рационализме, устойчивости к воздействию определенных температур и экономических показателях. При этом они экспериментальные, повторяющиеся и эмпирические.

Вы принимаете решение на основе того, что видите, и своих предположений о том, что это будет означать, а затем тестируете свои идеи на работоспособность. И это не всегда достаточно точно предсказуемый процесс. В SpaceX построили макеты, а затем подвергли их испытанию давлением — сперва воды, а затем жидкого азота, чтобы протестировать поведение материала (стали) при низких температурах. Инженерное проектирование — это в значительной степени исследовательский подход к получению знаний.

КОМПРОМИССЫ

Инженерия — это не что иное, как сочетание оптимизации и компромиссов. Мы стремимся успешно решить задачу и неизбежно сталкиваемся при этом с выбором. Одним из самых крупных компромиссов, на который пришлось пойти SpaceX при создании ракет, стал выбор между мощностью и весом. Это общая проблема всех летательных аппаратов и в целом средств передвижения.

Чтобы принимать инженерные решения, жизненно необходимо понимать, на какие компромиссы придется идти.

Повышение безопасности системы влечет за собой сложности в ее использовании. Если система будет более рассредоточенной, на объединение данных потребуется больше времени, чем на их сбор. Увеличение штата разработчиков для ускорения процесса усложнит коммуникацию и взаимодействие, что в целом замедлит процесс.

Один из ключевых компромиссов разработки программного обеспечения на любом уровне детализации функций системы — это связанность, или сцепление. Мы рассмотрим ее более подробно в главе 13.

ИЛЛЮЗИЯ ПРОГРЕССА

Изменения, которые произошли в нашей отрасли, впечатляют. Но я считаю, что большинство из них в действительности не играют большой роли.

Я пишу эти строки на конференции, посвященной бессерверным вычислениям¹. Переход на бессерверные системы — интересный шаг. Однако различия между наборами инструментов AWS, Azure, Google и других поставщиков весьма незначительны.

Решение использовать бессерверные вычисления повлияет на дизайн системы. Где хранить состояния? Где их обрабатывать? Как разделять функции системы? Как организовать сложную систему и перемещение по ней, когда единицей проектирования является функция?

Эти вопросы представляют гораздо больший интерес и большую важность для успеха вашего проекта, каким бы он ни был, чем способ обозначения функции или использования хранилищ либо средств безопасности платформы. Тем не менее почти все материалы по данной теме посвящены в основном инструментам, а не дизайну систем.

Представьте, что плотнику рассказали о различиях винтов с прямым и крестообразным шлицем, но не сказали, для чего использовать каждый из них, а также когда использовать винты, а когда — гвозди.

Бессерверные вычисления — это новый этап развития модели вычислений. Но я не буду здесь рассказывать о них. Эта книга о том, как определить, что важно, а что нет.

Бессерверные вычисления важны по нескольким причинам, но в основном потому, что они способствуют **модульному подходу** к проектированию с более четким **разделением ответственности**, особенно в отношении данных.

Бессерверные вычисления меняют экономику системы, переводя вычисления от схемы «стоимость на байт» к схеме «стоимость на такт процессора». Это означает, или должно означать, что нам необходимо учитывать различные способы оптимизации.

Вместо того чтобы оптимизировать системы для сокращения объема хранилищ, нормализуя массивы данных, нам, вероятно, следует внедрить более распределенную модель вычислений с использованием ненормализованных массивов и шаблонов конечной согласованности. Это важно, поскольку влияет на модульность создаваемых систем.

¹ Бессерверные вычисления — это облачная стратегия реализации подхода «функция как услуга». Функции образуют единый вычислительный блок, а выполнение кода для вызова функции инициируется по запросу.

Инструменты имеют значение только в той степени, в какой они являются «двигателями прогресса» более фундаментальных понятий.

ОТ РЕМЕСЛА К ИНЖЕНЕРНОМУ ДЕЛУ

Не следует недооценивать значение мастерства. Тщательность и внимание к деталям необходимы, чтобы создавать продукты высокого качества. Не нужно также недооценивать значение инженерии для повышения качества и эффективности продуктов ремесленного производства.

Братья Райт первыми создали управляемый летательный аппарат, который был тяжелее воздуха. Они были превосходными мастерами и не менее превосходными инженерами. Большую часть своей работы они основывали на эмпирических открытиях, но проводили и реальные испытания эффективности своей конструкции. Будучи первыми конструкторами летательного аппарата, они также первыми создали аэродинамическую трубу, в которой оценивали эффективность конструкции крыла.

Крыло самолета — это удивительное сооружение. Конструкция братьев Райт была очень красивой, хотя по современным стандартам довольно примитивной. Каркас состоял из дерева и проволоки, покрытых плотной тканью, пропитанной банановым маслом в целях защиты от ветра.

Такую конструкцию наряду с аэробумбой Райты использовали для изучения теории аэродинамики на основе работ предшественников. Однако изначально летательный аппарат, в частности его крылья, братья строили путем проб и ошибок, а не на основе теории.

С современной точки зрения этот аппарат — продукт ремесленного, а не инженерного производства. Это действительно так, хотя не во всем. Многие люди пытались смастерить самолет, но им это не удавалось. Братья Райт добились успеха во многом потому, что они применили инженерный подход. Они совершали расчеты и использовали инструменты для измерения и исследований. Они применили управление переменными, поэтому смогли глубже понять и улучшить свою модель. Они создавали тестовые модели летательных аппаратов и конструкции аэробумбы и затем изучали их. Полученные ими знания были неидеальны, но они основывались не только на практических опытах, но и на теории.

К тому времени как братья Райт осуществили управляемый полет на аппарате тяжелее воздуха, их знания аэродинамики позволили им создавать машины с относительной дальностью планирования 8,3:1¹.

По сравнению с современными конструкциями крыло самолета братьев Райт имело недостаточную кривизну (аэродинамический профиль для медленного подъема) и было довольно тяжелым, хотя для своего времени конструкция считалась легкой. Для ее создания использовались простые натуральные материалы, и это помогло достичь соотношения 8,3:1.

Применение инженерного подхода, опытного обнаружения, экспериментов, а также знания в материаловедении, аэродинамической теории, компьютерном моделировании и так далее позволили построить современное крыло. Оно более удлиненное и изготавливается из углеволокна. Сочетание легкости и подъемной силы оптимально для набора высоты, а относительная дальность планирования превышает соотношение 70 : 1 — это почти в 9 раз выше, чем у аппарата братьев Райт.

РЕМЕСЛА НЕДОСТАТОЧНО

Ремесленный подход очень важен, особенно если при этом вы действительно подразумеваете креативность. Разработка программного обеспечения — очень креативная сфера, но и инженерия тоже. Я считаю, что инженерия на самом деле является вершиной человеческой креативности и мастерства. Нам просто необходим инженерный подход, если мы хотим создавать действительно крутые программные продукты.

ПРИШЛО ЛИ ВРЕМЯ МЫСЛИТЬ ИНАЧЕ?

Развитие **программной инженерии** как дисциплины не совсем соответствует ожиданиям людей. Программы изменили — и меняют — мир. Мы видели появление действительно выдающихся результатов и потрясающих

¹ Относительная дальность планирования — это показатель эффективности летательного аппарата, соотношение между пройденным расстоянием и потерей высоты. Например, каждый фут (метр) снижения при (безмоторном) планировании соответствует пройденному расстоянию 8,3 фута (2,53 метра).

инновационных систем. Но многие команды, компании и отдельные разработчики не всегда понимают, как добиваться таких результатов.

Наша сфера изобилует философиями, практиками, процессами и технологиями. Специалисты ведут настоящие войны за лучший язык программирования, или подход к архитектуре, или процесс разработки и инструмент. Часто кажется, что за всем этим теряется понимание действительных целей и задач разработки.

Современные команды работают под давлением сроков, требований к качеству и к удобству сопровождения своих систем. Они повсеместно идут на поводу у пользователя и не находят времени, чтобы изучить предметную область, технологию и возможности создать что-то действительно выдающееся.

Компании очень часто не получают желаемого от своих разработок. Они сетуют на недостаточное качество и эффективность работы команд. Однако у них складывается неверное представление о том, на что те способны, чтобы справиться с этими сложностями.

Тем не менее эксперты, мнение которых я ценю, сходятся в определении некоторых основных принципов, которые применяют достаточно редко или по крайней мере недостаточно четко.

Возможно, настало время пересмотреть некоторые из этих основополагающих принципов. Каковы они для нашей отрасли? Какие из них останутся актуальными спустя десятилетия, а не только при нынешнем поколении технических средств?

Разработка ПО – это сложная и неоднородная задача. Однако есть и общие практики, способы изучения, управления, организации и осуществления разработки, которые оказывают значительное, даже колоссальное, влияние на все стороны нашей отрасли.

Далее я расскажу о некоторых из этих общих практик и перечислю основные принципы для разработки ПО в любой предметной области, независимо от используемых инструментов и требований к качеству и бюджету.

Как мне кажется, они раскрывают глубинную природу нашей сферы.

Если мы будем следовать этим принципам, как поступают многие команды, мы добьемся большей продуктивности, снизим уровень стресса и выгорания у разработчиков, повысим качество и устойчивость создаваемых систем.

Эти системы лучше служат пользователям. Мы наблюдаем значительно меньшее количество ошибок, и команды, следующие этим принципам, отмечают, что по мере накопления знаний им становится значительно проще изменять любой компонент системы. Экономический результат — повышение прибыльности разрабатываемого продукта. Все это отличительные признаки **инженерного** подхода.

Инженерия усиливает наши способности творить, приносить пользу, создавать качественный продукт, которому доверяют. Благодаря инженерии мы способны проводить более глубокие исследования и повышать наш творческий потенциал, чтобы создавать еще более масштабные и еще более сложные системы.

У нас есть шанс создать уникальную программно-инженерную дисциплину. Если мы используем эту возможность, то сможем изменить саму суть ведения, организации и изучения разработки ПО. Возможно, это задача для нескольких поколений, но ее решение имеет настолько большую ценность для компаний, да и для всего мира, что мы просто обязаны попытаться. Что, если мы сможем создавать программы гораздо быстрее и гораздо дешевле? Что, если наши продукты станут еще более качественными, адаптивными, устойчивыми, простыми в обслуживании и лучше отвечающими потребностям пользователей?

ИТОГИ

В сфере разработки программного обеспечения понятие инженерии несколько переопределено. Иногда мы рассматриваем инженерию как что-то ненужное, обременительное и ограничивающее, мешающее «настоящей разработке». Инженерия в других сферах не имеет ничего общего с нашей. Другие инженеры быстрее, а не медленнее получают результат. Они создают продукты более высокого качества, а не хуже.

Когда мы начнем применять практический, рациональный, легкий, научный подход к разработке ПО, мы увидим аналогичный эффект. Программная инженерия будет специфичной в нашей области, но она поможет нам быстрее создавать лучшие программные продукты.

ГЛАВА 3

ОСНОВЫ ИНЖЕНЕРНОГО ПОДХОДА

Инженерия имеет свои особенности в каждой сфере. Строительство мостов и самолетов — это разные отрасли, так же как электроинженерия и химическая инженерия. Но инженерия во всех этих отраслях имеет кое-что общее. В основе всегда лежит научный рационализм и практический эмпирический подход к достижению результатов.

Если мы хотим сформировать набор мыслей, идей практик и моделей поведения, составляющих *программную инженерию*, нам необходимо учесть, что все эти понятия должны быть основополагающими для всей разработки и быть устойчивыми к изменениям.

РАЗРАБОТКА – ЭТО ИНДУСТРИЯ ИЗМЕНЕНИЙ?

Мы много говорили об изменениях в сфере разработки. Нас восхищают новые технологии и новые продукты. Но действительно ли эти изменения ведут к прогрессу отрасли? Многие из них на поверку не несут чего-то принципиально нового, как мы рассчитывали.

Мое любимое доказательство этому факту представила Кристин Горман (Christin Gorman) в своей очаровательной презентации на одной из конференций¹. В ней Кристин показала, что при использовании некогда популярной библиотеки Hibernate с открытым исходным кодом, предназначенней для решения задач объектно-реляционного отображения,

¹ Источник: Кристин Горман, Gordon Ramsay Doesn't Use Cake Mixes.

в действительности приходилось писать больше кода, чем при использовании SQL, по крайней мере субъективно; к тому же SQL более понятен. Далее Кристин сравнивает разработку с выпеканием торты. Когда вы печете торт, вы берете готовую смесь для выпечки или же готовите тесто сами?

Большинство изменений в нашей сфере — кажущиеся и не способствуют улучшениям. А некоторые, как в примере с Hibernate, даже делают хуже.

Мне кажется, что наша отрасль испытывает довольно серьезные проблемы с обучением и достижением результатов. Их относительный недостаток маскируется невероятным прогрессом оборудования, на котором выполняется наш код.

Я не утверждаю, что в программном обеспечении прогресс вовсе отсутствует, — это далеко не так, но я считаю, что он идет гораздо медленнее, чем многие из нас полагают. Задумайтесь на минуту: какие события в вашей отрасли оказали значительное влияние на то, как вы занимаетесь разработкой? Что именно повлекло за собой изменение качества, масштаба или сложности задач, которые вы решаете?

Этот список короче, чем мы обычно считаем.

К примеру, я за свою карьеру использовал порядка 15–20 языков программирования. И хотя у меня есть предпочтения, только два перехода от одного языка к другому действительно коренным образом изменили мои представления о программах и разработке.

Это был переход от ассемблера к Си и от процедурных языков к объектно-ориентированным. Отдельные языки гораздо менее важны, чем общая методология. Эти два шага ознаменовали собой значительные изменения в уровне абстракции при написании кода и, как следствие, в сложности систем, которые мы могли создавать.

Когда Фред Брукс писал, что увеличение производительности на порядок невозможно, он кое-что упустил. Увеличение на порядок, вероятно, невозможно, но снижение на порядок возможно совершенно точно.

Я видел компании, загнавшие себя в угол из-за использования неверного подхода к разработке, — иногда в области технологий, но чаще в области процессов. Однажды я консультировал крупную компанию, которая более пяти лет не могла выпустить ни одного релиза.

Изучать новое действительно сложно; нам почти невозможно отказаться от старого, каким бы несостоятельным оно ни было.

ВАЖНОСТЬ ИЗМЕРЕНИЙ

Одна из причин, по которой нам сложно отказаться от плохих идей, в том, что в действительности мы не очень эффективно измеряем свою производительность.

Большинство метрик, используемых в разработке, нам либо не подходят (скорость), либо вообще вредят (количество строк кода или тестовое покрытие).

В agile-разработке бытует устоявшееся мнение, что измерить производительность проекта или команды невозможно. Об этом, в частности, писал Мартин Фаулер в своем популярном блоге Bliki в 2003 году¹.

Его точка зрения верна; у нас нет надежного способа измерить продуктивность, но это не значит, что мы вообще не можем измерить что-то полезное.

Опубликованные Николь Фосгрен, Джезом Хамблом и Джином Кимом отчет State of DevOps² и книга «Ускоряйся! Наука DevOps: Как создавать и масштабировать высокопроизводительные цифровые организации»³ посвящены способности принимать лучшие решения, основанные на доказательствах. Авторы представили интересную и убедительную модель измерения производительности работы команд программистов — очень важный шаг вперед для отрасли в целом.

Что интересно, они не старались определить продуктивность, скорее они оценивали эффективность команд по двум главным показателям. Измерения, которые затем и использовались для составления прогнозной модели, не доказывают, что показатели имеют причинно-следственную связь с производительностью команд, но демонстрируют их статистическое соответствие.

Эти показатели — **стабильность и пропускная способность**. Стабильные команды с высокой пропускной способностью определяются как высокопроизводительные, а команды с низкими значениями этих двух показателей — как непроизводительные.

¹ Источник: Мартин Фаулер, Cannot Measure Productivity, <https://bit.ly/3mDO2fB>.

² Источник: Николь Фосгрен, Джез Хамбл, Джин Ким, <https://bit.ly/2PWujw7>.

³ В книге «Ускоряйся!» приведены данные о том, что команды, применяющие более дисциплинарный подход к разработке, тратят на новые задачи на 44 % больше времени, чем команды, которые его не применяют.

Интересно, что если вы проанализируете работу этих двух типов команд, то обнаружите некую закономерность. Высокопроизводительные команды работают схожим образом. Мы можем предсказать производительность команды, руководствуясь моделью ее работы. Для прогнозирования производительности достаточно наблюдать за действиями команды.

Например, если команда использует автоматизацию тестирования, магистральную разработку (Trunk Based Development – TBD) и еще около 10 похожих практик, она, скорее всего, осуществляет **непрерывную доставку**. В таком случае, согласно прогнозной модели, команду можно считать высокопроизводительный в том, что касается доставки и организации.

И наоборот, если мы посмотрим на компании, которые считаются высокопроизводительными, то заметим, что они все используют непрерывную доставку и работу в небольших командах.

Измерение стабильности и пропускной способности позволяет разработать модель для оценки результата команды.

Как стабильность, так и пропускная способность оценивается по двум метрикам.

Метрики стабильности:

- **Change Failure Rate (частота отказов)** — доля случаев, когда изменение приводит к ошибке в определенной точке процесса;
- **Recovery Failure Time (время восстановления)** — сколько времени занимает восстановление после сбоя в определенной точке процесса.

Измерение стабильности важно, поскольку в действительности это измерение качества проделанной работы. Оно не скажет вам, делает ли команда действительно то, что нужно, но покажет, насколько эффективно команда доставляет продукт с измеряемым качеством.

Метрики пропускной способности:

- **Lead Time (время на реализацию)** — оценка эффективности процесса разработки, то есть сколько времени занимает путь от идеи до рабочего продукта;
- **Frequency (частота)** — оценка скорости, то есть как часто осуществляются развертывания.

Пропускная способность оценивает, насколько эффективно команда генерирует идеи, которые приводят к рабочему программному продукту.

Сколько времени занимает процесс доставки изменения пользователю и как часто это происходит? Помимо всего прочего, это показатель возможности обучения команды. Команда не обязательно этим воспользуется, но при отсутствии высоких показателей пропускной способности вероятность обучения для любой команды снижается.

Это технические метрики в рамках подхода к разработке. Они отвечают на вопросы, каково качество работы и насколько эффективно команда выполняет работу подобного качества.

Ответы важные, но неполные. Они не показывают, делаем ли мы правильные вещи, а лишь — делаем ли мы их правильно; но они не становятся менее полезными только потому, что не идеальны.

Интересно, что описанная выше корреляционная модель охватывает гораздо больше аспектов, чем просто расчет количества членов команды и необходимость постоянного развертывания. Авторы книги «Ускоряйся!» приводят данные, свидетельствующие о значимой взаимосвязи намного более важных характеристик.

К примеру, компании, в которых работают высокопроизводительные команды, согласно данной модели более прибыльны, чем те, в которых таких команд нет. Существуют данные, свидетельствующие о наличии взаимосвязи между подходом, используемым в разработке, и финансово-выми результатами компаний.

Авторы также развенчивают популярное убеждение в том, что можно обеспечить либо скорость, либо качество, но не оба этих показателя одновременно. Это просто неправда. Скорость и качество, как показывает данное исследование, тесно взаимосвязаны. Залог скорости — высокое качество продукта, залог высокого качества продукта — скорость обратной связи, а залог и скорости и качества — грамотная инженерия.

ИСПОЛЬЗОВАНИЕ МЕТРИК СТАБИЛЬНОСТИ И ПРОПУСКНОЙ СПОСОБНОСТИ

Взаимосвязь между высокими значениями рассмотренных метрик и высоким качеством работы очень важна. Она позволяет применять эти метрики для оценки изменения процессов, организации, культуры и технологий.

Представим, что мы хотим повысить качество нашего продукта. Как это сделать? Вероятно, внести изменения в процессы. Для этого необходим совет по утверждению изменений (change approval board, CAB).

Очевидно, дополнительные этапы проверки и утверждения повлияют на пропускную способность и неизбежно замедлят процессы. Однако повысится ли стабильность?

В нашем примере ответ известен. Удивительно, но CAB не повышает стабильность. Однако она уменьшается при замедлении процессов.

Мы обнаружили, что внешние процессы утверждения отрицательно коррелировали со временем выполнения, частотой развертывания и временем восстановления и не имели корреляции с частотой возникновения ошибок. Копроче говоря, утверждение внешним органом (менеджером или CAB) просто не работает на повышение стабильности производственных систем, измеряемой временем восстановления и частотой возникновения ошибок. Однако это, безусловно, замедляет работу, что на самом деле очень плохо. Лучше вообще не утверждать изменения¹.

В действительности моя цель — не глумиться над советами по утверждению изменений, а показать, что принимать решения следует на основе доказательств, а не предположений.

То, что идея создания CAB плоха, понять довольно сложно, поскольку она звучит разумно, и многие компании, если не сказать большинство, стремятся обеспечивать качество именно таким способом. Проблема в том, что это не работает.

Но без эффективной системы измерений мы не можем это утверждать, мы можем только предполагать.

Если вы начнете применять рациональный, основанный на доказательствах подход к принятию решений, то вам не следует принимать на веру мои слова, или слова Фосгрен и ее коллег, или кого-то еще.

Надо просто провести измерения в своей команде. Оценить ее стабильность и пропускную способность. Внедрить изменение. Способствует ли такое изменение повышению каких-либо из показателей?

¹ Форстгрен Н., Хамбл Дж., Ким Дж. Ускоряйся! Наука DevOps: Как создавать и масштабировать высокопроизводительные цифровые организации.

Вы можете узнать больше об этой модели корреляции из замечательной книги «Ускоряйся!». В ней описан подход к организации измерений и соответствующая модель, основанная на проводимых исследованиях. Я не буду пересказывать книгу, лишь отмечу значительное, а возможно, и основополагающее влияние этого подхода на нашу сферу деятельности. **Наконец у нас появилось надежное мерило.**

Данную модель можно использовать для оценки эффективности любого изменения.

Мы можем наблюдать, как изменения влияют на организацию, процессы, культуру и технологии. «Если я использую этот новый язык, повысится ли стабильность или пропускная способность моей команды?»

Эти показатели годятся для анализа этапов процесса. «Если я провожу довольно большой объем ручного тестирования, то трачу больше времени, чем если бы оно было автоматическим, но повышается ли при этом стабильность?»

Нам необходимо тщательно обдумывать свои действия и учитывать последствия. Что произойдет, если пропускная способность понизится, а стабильность повысится?

Тем не менее наличие значимых критериев измерения важно и даже жизненно необходимо для принятия решений на основе данных.

ОСНОВЫ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Итак, каковы же эти основные принципы, которые останутся актуальны через столетие и которые применимы к любой задаче и технологии?

Существуют две категории: процесс, или, возможно, даже его философия, и технология, или проектирование.

Говоря проще, наша отрасль должна опираться на две ключевые компетенции.

Нам необходимо стать **экспертами в познании**, усвоить и принять тот факт, что наша отрасль проектная и креативная, не связанная с производственной инженерией и делающая упор на мастерстве исследования, открытиях и изучении. Здесь на практике применяется научный стиль мышления.

Нам также необходимо совершенствовать навыки управления сложностью. Мы создаем системы, которые не дано охватить одному человеку. Эти системы масштабны, и над ними работает множество людей. В подобных условиях нам необходимо стать **экспертами в управлении сложностью** как на техническом, так и на организационном уровне.

ЭКСПЕРТНОЕ ПОЗНАНИЕ

Наука – лучшая методика решения задач, которую создало человечество. Чтобы стать экспертами в познании, необходимо освоить практический научноинформированный подход к решению задач, который составляет суть инженерных дисциплин.

Его необходимо адаптировать для задач разработки ПО. Программная инженерия отличается от других видов инженерии так же, как авиационная инженерия отличается от химической. Для решения задач разработки необходим практический, легкий и всеобъемлющий подход.

Лидеры мнений в нашей отрасли выражают единую точку зрения на эту проблему. Хотя их идеи широко известны, их редко используют для организации разработки.

Этот подход определяют пять практик:

- работать итеративно;
- оперативно предоставлять качественную обратную связь;
- быть последовательными;
- экспериментировать;
- придерживаться знаний, полученных из опыта.

На первый взгляд такие практики могут показаться абстрактными и далекими от повседневных задач разработчика, не говоря уже об инженере.

Разработка ПО – это применение навыков исследования и открытия. Мы всегда стремимся узнать больше о потребностях пользователей, о том, как эффективнее решить задачу и применить доступные инструменты и техники.

Мы узнаем, что что-то упустили, и значит, это необходимо исправить. Мы обдумываем, что следует предпринять, чтобы работать более эффективно, и учимся глубже исследовать задачу, над которой работаем.

Познание — краеугольный камень всего, что мы делаем. Его практики лежат в основе эффективной разработки и помогают отбросить менее эффективные подходы.

К примеру, водопадная разработка не позволяет реализовать эти практики. Тем не менее она десятилетиями ассоциируется с высокой производительностью команд и служит маркером успеха в отрасли.

В части II мы рассмотрим эту тему с более практической точки зрения. Я расскажу, как стать экспертом в познании и как использовать это качество в повседневной работе.

ЭКСПЕРТНОЕ УПРАВЛЕНИЕ СЛОЖНОСТЬЮ

Как разработчик, я рассматриваю окружающий мир с точки зрения своей области знаний. Вследствие этого я воспринимаю ошибки в разработке главным образом с позиции двух принципов теории информации: параллелизма и связанности.

Эти принципы довольно сложны для понимания в целом, а не только в сфере разработки. Они обусловлены структурой создаваемых систем и влияют на то, как работают компании-разработчики.

Их можно объяснить, опираясь на такие понятия, как закон Конвея¹, но он является скорее независимым свойством этих глубоких принципов.

Более эффективно рассматривать их с технической точки зрения. Любое объединение людей — это такая же информационная система, как и вычислительная система. Оно почти всегда очень сложное, но основано на тех же принципах — параллелизме и связанности.

Если мы беремся за создание систем, превосходящих по сложности учебные задачи по программированию, необходимо учитывать эти принципы.

¹ В 1967 г. Мервин Конвей (Mervin Conway) сформулировал идею о том, что «любая организация, которая разрабатывает какую-либо систему (в широком смысле), неизбежно создаст такую модель, которая будет повторять коммуникационную структуру самой организации». См. <https://bit.ly/3s2KZP2>.

Нам придется управлять сложностью этих систем по мере их создания, и, если мы собираемся масштабировать их за пределы зоны ответственности одной небольшой команды, нам следует управлять сложностью информационных систем организаций, а также технических информационных систем.

Я считаю, что наша отрасль уделяет слишком мало внимания этим принципам, и результаты известны всем, кто хоть немного знаком с разработкой: системы с нераспознаваемой архитектурой («большие комки грязи»), неуправляемый технический долг, огромное количество ошибок и боязнь вносить изменения.

Все это признаки утраты контроля над сложностью создаваемых систем.

Если вы работаете над простой и не очень значимой системой, качество проектирования не столь важно. Если же вы создаете что-то более сложное, необходимо разделить задачу на составные части и работать с каждой по отдельности, чтобы контролировать ее сложность.

Границы частей зависят от ряда факторов: от области задачи, которую вы решаете, технологии, которую вы применяете, и даже в какой-то степени, вероятно, от уровня вашей эрудиции; но вам просто необходимо уметь ими управлять, если вы собираетесь решать сложные задачи.

Как только вы это осознаете, вы поймете, что эти факторы имеют очень большое влияние на дизайн и архитектуру создаваемых систем. Я немного опасался говорить об эрудиции в предыдущем абзаце, но речь именно о ней. Проблема в том, что большинство из нас переоценивает свою способность разобраться в коде.

Один из самых полезных уроков, который нам дает неформальный подход к науке, заключается в том, что лучше всего изначально подвергнуть свою идею критическому осмыслинию и допускать возможность, что она неверна. Поэтому во время разработки следует пристальнее отслеживать возможный взрывной рост сложности системы и не допускать этого.

При управлении сложностью важно учитывать пять принципов, которые тесно связаны друг с другом и с моделями поведения экспертов в познании (особенно если необходимо структурированное управление сложностью любой информационной системы):

- модульность;
- связность;

- разделение ответственности;
- сокрытие данных/абстракция;
- связанность, или сцепление.

Мы подробнее разберем каждый из этих принципов в части III.

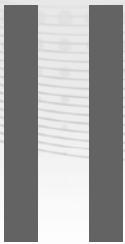
ИТОГИ

Нашиими рабочими инструментами зачастую оказываются не те, которые мы таковыми считаем. Для разных проектов мы можем использовать разные языки, программные средства и фреймворки, реальные же наши рабочие инструменты — это те принципы, которые помогают нам учиться и работать со сложными системами. Соблюдая их, мы сможем выбрать более подходящий язык, лучше освоить рабочее средство и использовать подходящий фреймворк, чтобы эффективнее решать поставленные задачи.

Наличие эталона для измерения показателей работы — огромное преимущество, если мы хотим принимать решения на основе данных и доказательств, а не предположений. Принимая решение, задавайте себе вопрос: «Повысит ли решение качество создаваемого продукта?» — если вы измеряете **стабильность**, и «Повысит ли оно эффективность, с которой мы создаем продукт такого качества?» — если вы измеряете **пропускную способность**. Стоит внедрять решение, если оно не ухудшит ни один из этих показателей. Иначе зачем его вообще внедрять?

Ещё больше книг в нашем телеграм канале:
<https://t.me/bookofgeek>





ОПТИМИЗАЦИЯ ДЛЯ ОБУЧЕНИЯ

ГЛАВА 4

ИТЕРАТИВНЫЙ ПОДХОД

Итерация — это «способ организации работы, при котором успешное повторение последовательности операций приводит к желаемому результату»¹.

Если говорить в общем, то итерация способствует познанию. Она помогает изучать и адаптировать изученное, а также взаимодействовать с ним. Без итераций и сбора обратной связи невозможно постоянно обучаться. Итерация позволяет нам совершать ошибки и исправлять их и двигаться вперед.

Это определение также напоминает нам, что итерация — возможность приближаться к цели, даже когда в действительности мы не знаем, как ее достичь. Поскольку мы способны так или иначе понять, приближаемся мы к цели или отдаляемся от нее, мы можем совершать даже случайные итерации и все же прийти к цели. В нашей воле избегать действий, которые отдаляют нас от цели, и совершать те, которые делают цель ближе. Так работает эволюция. И так работает современное машинное обучение (МО).

AGILE-РЕВОЛЮЦИЯ

Разработчики используют итеративный, основанный на обратной связи подход как минимум с 1960-х годов. Идея, предполагающая внедрение более гибких, основанных на обучении стратегий разработки вместо распространенных в то время тяжеловесных процессов, была сформулирована в манифесте гибкой разработки по итогам встречи ведущих практиков и лидеров мнений на горнолыжном курорте в Колорадо.

¹ Источник: Толковый словарь Merriam Webster Dictionary, <https://www.merriam-webster.com/dictionary/iteration>.

Agile-манифест¹ довольно прост. Он состоит из 9 строк и включает 12 принципов, но значение его огромно.

До его принятия бытовало общее мнение (за редкими исключениями), что если вы разрабатываете что-то «серьезное», вы должны использовать водопадные техники, ориентированные на конечный продукт.

Методология agile оказалась прорывной, и сейчас именно она, а не водопадная модель является доминирующей, по крайней мере в мышлении.

Однако многие компании до сих пор тяготеют к водопадной модели – если не в техническом плане, то в организационном.

Тем не менее agile-методология имеет более прочные основы, чем ее предшественники. Их можно сформулировать как «инспекция и адаптация» (Inspect and Adapt, I&A).

Подобное изменение взглядов было значительным, но недостаточным. Почему значительным? Потому что оно ознаменовало стремление трактовать разработку как задачу познания, а не производства. Водопадная модель может быть эффективной для некоторых задач, но она проигрывает там, где требуется исследование.

Хотя повышение производительности на порядок, о чём говорил Фред Брукс, и кажется невозможным с точки зрения технологий, инструментов и процессов, некоторые подходы настолько неэффективны, что усовершенствовать их в разы очень даже реально. Водопадная разработка – именно такой случай.

Водопадный способ мышления основан на предположении, что «если мы хорошо подумаем/поработаем, то сразу же сделаем то, что нужно».

Agile переворачивает подобные представления. Он утверждает, что мы обязательно сделаем все не так. «Мы не понимаем, чего хочет пользователь», «мы не создадим нужный дизайн с первой попытки», «мы не знаем, исправили ли мы все баги в коде», и так далее и т. п. Команды, работающие по принципам agile, исходят из того, что они совершают ошибки, и тем самым снижают стоимость этих ошибок.

Agile разделяет научные принципы. Именно научному подходу свойственно критическое рассмотрение идей и стремление доказать их ошибочность, а не верность (опровергнуть).

Эти два способа организации мышления, основанные на предсказуемости и исследовании, формируют абсолютно разные, несовместимые подходы к управлению проектами и работой команд.

¹ Agile-манифест, <https://agilemanifesto.org/>.

Согласно принципам agile, мы организуем работу команд, процессы и технологии так, чтобы ошибки не имели серьезных последствий, их было легко выявлять, исправлять и в идеале избегать в дальнейшем.

Споры о преимуществах Scrum перед экстремальным программированием, или непрерывной интеграции перед ветвлением функционала, или разработки через тестирование перед тщательным обдумыванием кода не имеют смысла. Любой действительно гибкий процесс – это эмпирическое управление процессом.

Такой подход к разработке гораздо эффективнее, чем предшествующий ему, основанный на конечном продукте прогнозный метод водопадной разработки.

Итеративный подход принципиально отличается от последовательной работы по плану. Он значительно более эффективен.

Многим читателям это покажется очевидным, но это не так. История разработки показывает, что раньше итерации считались необязательными, а основой успешной работы было тщательное предварительное планирование всех ее этапов.

Итерация лежит в основе всей исследовательской деятельности и является необходимым условием приобретения действительных знаний.

ПРАКТИЧЕСКИЕ ПРЕИМУЩЕСТВА ИТЕРАТИВНОГО ПОДХОДА

Если мы рассматриваем разработку как познание и открытие, то в ее центре должна находиться итерация. Однако преимущества итеративного подхода не всегда очевидны с первого взгляда.

Возможно, самое важное — то, что переход к итеративному способу организации работы автоматически сужает наше поле зрения и побуждает думать категорийно, уделяя внимание модульности и разделению ответственности. Эти понятия возникают как прямое следствие итеративного подхода, но в конце концов становятся частью логического механизма повышения качества нашей работы.

Одна из идей, общих для Scrum и экстремального программирования, — это необходимость деления рабочего процесса на небольшие этапы. Логика

agile такова: прогресс в разработке трудно измерить, но мы можем измерить завершенный функционал, поэтому следует работать с отдельными функциями, чтобы видеть результат.

Подобное ограничение стало значительным шагом вперед. Однако процесс усложняется, если вы хотите понять, сколько осталось до «завершения». Такой итеративный подход к разработке отличается от традиционных методов. Например, при непрерывной доставке мы готовим незначительные изменения к релизу по несколько раз в день. Они должны быть настолько завершенными, чтобы релиз продукта оставался надежным и безопасным в любой момент. Так что же в действительности означает «завершенный» в этом контексте?

Каждое изменение завершено, поскольку оно готово к релизу, поэтому единственной осозаемой метрикой окончания изменения является ценность, которую это изменение создает для пользователя. Это очень субъективно. Как узнать, сколько корректировок понадобится, чтобы создать ценность для пользователя? Большинство компаний-разработчиков стремятся предоставить набор функций, которые в совокупности представляют «ценность», но если изменения возможны в любое время, то эта концепция становится довольно размытой.

Определить, какие именно изменения создают «ценность», сложно, поскольку это предполагает, что, приступая к работе, вы уже знаете, какие функции вам понадобятся, и способны определить, насколько вы прошли к цели, то есть к «завершенности». Это очень упрощенное изложение того, что имели в виду пионеры agile-методологии, но идея, которой руководствуются большинство традиционных компаний при переходе к agile-планированию, именно такова.

Одно из самых неочевидных преимуществ итеративного подхода в том, что у нас есть выбор. Мы можем совершать итерации, работая над продуктом, и управлять им на основе обратной связи от пользователей, чтобы повысить его ценность. Это один из наиболее значимых аспектов этого подхода, и его часто упускают из виду при внедрении такого способа работы.

Тем не менее, независимо от намерений или результатов, пакетно-ориентированный подход помог нашей отрасли уменьшить объемы и сложность функций, которые мы создаем, и это действительно важный шаг вперед.

Agile-планирование во многом основано на разбиении рабочего процесса на небольшие этапы, что позволяет завершить работу над одной функцией в рамках одного спринта, или итерации. Изначально это был способ измерения прогресса, но он оказался очень полезен с точки зрения получения точной и постоянной обратной связи о качестве и целесообразности нашей работы. В результате повышается скорость нашего обучения. Работает ли этот дизайн? Нравится ли эта функция пользователям? Достаточно ли быстро функционирует система? Все ли ошибки мы исправили? Грамотно ли написан код? И так далее.

Итерации небольших, четко определенных и готовящих продукт к релизу шагов помогают получить ценную обратную связь.

ИТЕРАЦИЯ КАК СТРАТЕГИЯ ЗАЩИТНОГО ПРОЕКТИРОВАНИЯ

Итерации способствуют внедрению защитного проектирования (defensive design). Мы рассмотрим этот подход подробнее в части III.

Интересную точку зрения на принципы agile впервые высказал мне мой друг Дэн Норт. Он представил различие между водопадным программированием и agile с точки зрения экономики. Водопадная разработка исходит из того, что с течением времени стоимость изменения растет. Это классическая интерпретация модели «Стоимость изменений» (рис. 4.1).

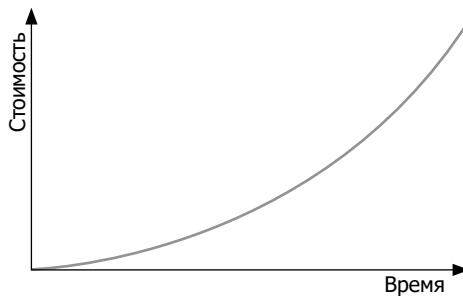


Рис. 4.1. Классическое представление стоимости изменений

Такое представление сопряжено со сложностями: если модель верна, то единственный разумный выход — принимать все важные решения

в начале работы над проектом. Но проблема в том, что на первоначальном этапе наши знания минимальны. Поэтому, как бы мы ни старались, мы принимаем ключевые решения, основываясь на ничем не подкрепленных предположениях.

Какой бы предварительный анализ мы ни провели, проект никогда не начнется с «...мы поняли, что нам делать, от и до». И учитывая, что мы никогда не будем иметь на старте «четко определенный набор исходных данных», вне зависимости от тщательности составления планов, регламентированная процессная модель, или водопадная разработка, даст сбой при первой же сложности. Невозможно вписать разработку ПО в столь неподходящий для нее шаблон.

Неожиданности, недопонимания и ошибки — обычное дело в разработке, потому что это часть процесса исследования и открытий, а значит, нам необходимо уделять больше внимания обучению, чтобы избегать ошибок, которые обязательно встретятся на нашем пути.

Вернемся к мысли Дэна Норта: если классическая модель стоимости изменений неэффективна, то что тогда? Что, если мы сделаем кривую этой модели более плоской (рис. 4.2)?

Предположим, мы изменим подход так, чтобы предлагать новые идеи, выявлять и исправлять ошибки на одном уровне стоимости независимо от того, на каком этапе это будет происходить. Кривая стоимости станет плоской.

Это даст нам свободу открытий и преимущества от использования открытий. Мы сможем непрерывно совершенствовать знания и код, а также опыт взаимодействия пользователя с нашим продуктом.

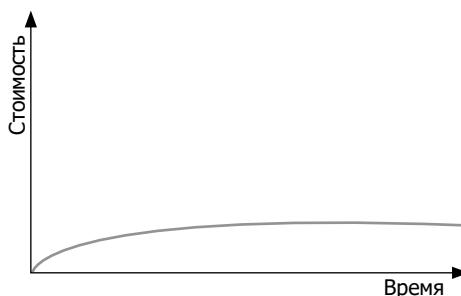


Рис. 4.2. Представление стоимости изменений в модели agile

Итак, что нужно, чтобы выровнять эту кривую?

Нам не удастся уделять больше времени исключительно анализу и проектированию, без реального процесса создания, поскольку это подразумевает меньше времени на изучение. Поэтому нам необходимо уплотниться, надо работать итеративно. Анализ, проектирование, код, тестирование и релизы требуются ровно в той степени, чтобы доставить продукт пользователю и увидеть, что реально работает. Затем следует сделать выводы и на основе новых знаний определить дальнейшие действия по улучшению продукта.

Это один из основных принципов непрерывной доставки (рис. 4.3).

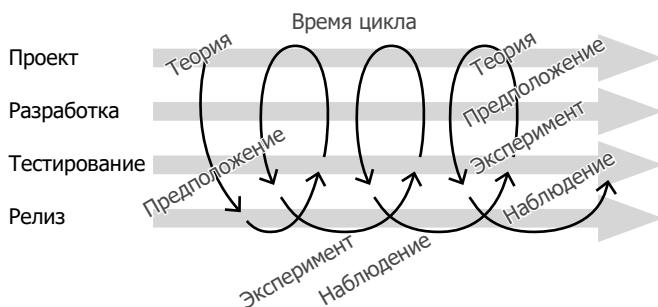


Рис. 4.3. Итерации непрерывной доставки

СИЛА ПЛАНИРОВАНИЯ

У сторонников водопадной разработки были благие намерения. Они считали, что это оптимальная стратегия успеха. В итоге мы десятилетиями пытались его добиться, но не смогли.

Сложность в том, что принципы водопада кажутся довольно разумными: «Тщательно обдумать, прежде чем начинать работу», «Грамотно спланировать действия и прилежно следовать этому плану». Исходя из опыта нашей отрасли, все это имеет смысл. Если ваш процесс строго регламентирован, подобный стиль даст отличные результаты.

При решении конкретных задач проблемы производственной инженерии и масштабирования обычно перевешивают проблемы дизайна. Однако сейчас это не так даже в промышленном производстве. По мере того как оно становится все более гибким и многие предприятия меняют

направленность, трансформируются даже самые жесткие производственные процессы. Производственный подход, тем не менее, преобладал в большинстве компаний на протяжении как минимум столетия, и мы сейчас остаемся его заложниками.

Чтобы осознать, что парадигма, в которой вы работаете, неверна, требуется огромное интеллектуальное напряжение. И оно кратно увеличивается, если осознать ошибку должен целый мир.

БИТВА ПРОЦЕССОВ

Если увеличение на порядок невозможно осуществить с помощью языка, формализации или построения диаграмм, то где искать выход?

И здесь чрезвычайно перспективно выглядит поиск по типичным для нашей отрасли способам самоорганизации и подходам к навыкам и техникам познания и открытий.

На заре программирования разработчики обычно имели математическое, естественно-научное или инженерное образование. Они действовали в одиночку или в небольших группах. Как большинство исследователей, они опирались на свой опыт и свои установки. Разработка на ранних этапах во многом напоминала математику.

Когда произошла компьютерная революция, спрос на разработку быстро превысил предложение. Требовалось как можно больше программных продуктов и как можно быстрее. Поэтому мы обратились к опыту других отраслей, чтобы научиться работать эффективно в масштабируемой среде.

И вот тут мы совершили ужасную ошибку, неверно истолковав саму природу разработки программных продуктов и заимствовав производственные техники. Мы привлекли целые армии программистов и попытались наладить аналог конвейерного производства товаров массового потребления.

Люди, которые совершили эту ошибку, не были глупыми, они просто заблуждались. Проблема была разноплановой. Программный продукт – комплексная вещь, и процесс ее создания не похож на привычную задачу производства, каким его видят большинство людей.

Первые попытки индустриализации разработки ПО были всеобъемлющими, болезненными и нанесли большой вред. Они привели к созданию множества некачественных продуктов. Решения оказались медленными, неэффективными, несвоевременными, они не соответствовали потребностям пользователей, и их было очень сложно обслуживать. В 1980-1990-е годы случился бум разработки и так же резко возросла сложность ее процессов во многих организациях.

Эта ошибка возникла, невзирая на тот факт, что в целом лидеры отрасли знали о существовании проблемы.

В книге Фреда Брукса «Мифический человеко-месяц», изданной в 1970 году, как раз описана эта проблема и то, как ее избежать. Если вы еще не читали этот знаковый для нашей отрасли труд, вы, несомненно, удивитесь, как точно в нем сформулированы проблемы, с которыми вы сталкиваетесь практически ежедневно. И это несмотря на то, что Брукс описывает только свой опыт разработки ОС для суперкомпьютера IBM 360 в начале 1960-х годов, когда в его распоряжении имелись довольно примитивные технологии и ограниченный инструментарий. Тем не менее книга затрагивает гораздо более важные и ключевые понятия, чем язык, инструменты и технологии.

В то время многие команды создавали превосходные продукты, зачастую полностью игнорируя общепринятые способы планирования и управления проектами. Такие команды имели общие черты. Они были небольшими. Разработчики взаимодействовали с пользователями их продукта. Они внедряли идеи быстро и меняли курс, если эти идеи не работали, как ожидалось. Это был революционный подход – настолько революционный, что многие из этих команд, по сути, работали украдкой, поскольку их компании внедряли тяжеловесные процессы, которые тормозили их работу.

К концу 1990-х годов в ответ на эти тяжеловесные процессы многие разработчики начали искать более эффективные стратегии. Стали популярными несколько конкурирующих подходов к разработке – Crystal, Scrum, экстремальное программирование и другие. Их общий подход нашел отражение в agile-манифесте.

Революция, которую произвела agile-методология, коренным образом изменила принятую в разработке норму, но даже сегодня многие компании, если не сказать большинство, все еще ориентированы на планирование/водопадный подход.

Вдобавок ко всей сложности выявления проблемы такие организации иногда выдают желаемое за действительное. Очень хорошо, если организация способна:

- выявить действительные потребности пользователей;
- правильно определить, какую выгоду принесет организации удовлетворение этих потребностей;
- точно оценить затраты на удовлетворение этих потребностей;

- решить на основе рациональной оценки, превысит ли выгода эти затраты;
- составить грамотный план действий;
- четко придерживаться этого плана;
- посчитать затраты по окончании работы.

Проблема в том, что это не дает эффекта ни на уровне бизнеса, ни на уровне технологии. Реальный мир и разработка ПО как его составная часть так не работают.

Данные свидетельствуют, что две трети всех идей самых продвинутых мировых компаний-разработчиков на самом деле не приносят им прибыли либо приносят убыток¹. Мы очень плохо понимаем, чего хотят пользователи. Даже если мы спросим их самих, они не ответят, потому что сами не знают этого. Самое эффективное здесь — итерация. Необходимо допустить, что некоторые, возможно, даже большинство наших идей — плохие, и испытывать их на практике как можно более быстрым, дешевым и эффективным способом.

Оценить ценность идеи для бизнеса тоже очень сложно, и это известный факт. Все знают слова главы IBM Томаса Дж. Уотсона (Thomas J. Watson), который предсказал, что мировая потребность в компьютерах составит целых пять штук!

Это не проблема технологий, это проблема человеческих ограничений. Чтобы двигаться вперед, следует использовать возможности, делать предположения, быть готовыми идти на риск. Но мы плохие прогнозисты. Поэтому, чтобы добиваться успеха, нам нужен такой способ самоорганизации, чтобы наши предположения не причинили нам вреда. Нам надо работать более осторожно, продвигаясь небольшими шагами и сужая радиус действия наших предположений, а также непрестанно обучаться. Следует работать итеративно!

Если у нас есть идея для работы, нам требуется способ определить, как завершить эту работу. Как остановить разработку плохой идеи? Если мы решили испытать решение, как сократить его масштаб так, чтобы не потерять вообще все, если оно окажется плохим? Нам необходимо уметь как можно быстрее выявлять плохие идеи. Здорово, если мы исключим

¹ Источник: Online Controlled Experiments at Large Scale, <https://stanford.io/2LdjvmC>.

их на этапе обдумывания. Но не все плохое так легко распознать. Успех — понятие скользкое. Идея может быть изначально хорошей, но несвоевременной, а ее исполнение — плохим.

Нам нужен быстрый и минимально затратный способ проверки предложений. Опрос, касающийся проектов в сфере разработки, проведенный в 2012 году McKinsey Group совместно с Оксфордским университетом, показал, что 17% крупных проектов (с бюджетом более 15 млн долларов США) были настолько плохи, что поставили под угрозу само существование компаний, которые их запустили. Как распознать такие плохие проекты? Если выполнять небольшие этапы, отслеживать прогресс и постоянно проверять и оценивать решение, мы скорее и с минимальными затратами заметим, если что-то пойдет не так, как задумывалось. При итеративном подходе стоимость каждого неверного шага несравнимо ниже, как и уровень риска.

В книге «Начало бесконечности» Дэвид Дойч описывает ключевое различие между идеями, применение которых ограничено, и теми, для которых таких ограничений нет. Разница между плановым, водопадным, ориентированным на процесс подходом и итеративным, исследовательским и экспериментальным подобна разнице этих двух фундаментальных видов идей. Модели управления регламентированными процессами¹ требуют наличия регламентированного процесса. Согласно определению, этот процесс ограничен. Ограничителем служит в какой-то степени способность человеческого мозга удерживать подробности всего процесса. Наш интеллект позволяет применять абстракцию и модульность, чтобы скрыть некоторые ненужные подробности, но необходимость представить некий план всего процесса вынуждает нас удерживать в уме всю его последовательность. Это изначально ограниченный подход к решению задач. Применяя его, мы способны решать только те задачи, которые понятны заранее.

Итеративный подход заметно отличается. Мы можем приступить к работе, даже если почти ничего не знаем о задаче, и все равно добиться успеха. Мы

¹ Кен Швабер (Ken Schwaber) описывал водопадную модель как модель управления регламентированным процессом, которой он дал такое определение: «Модель управления регламентированным процессом требует полного понимания каждого этапа работы. Регламентированный процесс начинается и выполняется до своего окончания с одинаковым результатом». Швабер сравнивает эту модель с моделью управления эмпирическим процессом, которую использует agile-методология. См. <https://bit.ly/2UiiaZdS>.

можем начать с той части системы, которая проста и понятна. Взять ее за основу исследования, пробовать кажущиеся подходящими архитектуру и технологии и так далее. Ничто здесь не является строго определенным. Мы получим результат, даже если поймем, что выбрали неподходящую технологию или архитектуру. Мы получим новые знания. Это изначально открытый, бесконечный процесс. Имея в своем распоряжении нечто вроде фитнес-функции, которая подсказывает нам, приближаемся ли мы к цели или отдаляемся от нее, мы можем продолжать действовать бесконечно, уточняя, расширяя и улучшая свои представления, идеи, навыки и продукты. Мы даже можем изменить эту фитнес-функцию, если поймем, что способны достичь лучших целей.

НАЧАЛО БЕСКОНЕЧНОСТИ

В необычайно познавательной книге «Начало бесконечности» физик Дэвид Дойч определяет науку и просвещение как поиск хороших объяснений и показывает, как на протяжении истории развития человечества разные идеи формируют «начало бесконечности», благодаря которому мы находим понятное и подходящее применение этим хорошим объяснениям.

Отличный тому пример – различие между алфавитным и пиктографическим письмом.

Первой формой письма была пиктография, и она до сих пор используется в китайской и японской письменности. Она очень красива, но имеет серьезный недостаток. Если вы услышите незнакомое слово, вы не сможете его записать, пока кто-то не научит вас. Пиктографическое письмо не инкрементно. Вам придется выучить символ для каждого слова. (В китайском языке примерно 50 000 иероглифов.)

Алфавит – совершенно иная система. Знаки алфавита обозначают звуки, а не слова. Вы сможете записать слово, возможно, с ошибками, но так, что любой поймет, что вы написали.

Вы сделаете это, даже если никогда не слышали этого слова или не видели, как оно пишется.

Точно так же вы сможете прочесть незнакомое слово. Вы сможете читать слова, не зная, как они произносятся или что означают. Это исключено при использовании пиктограмм. Таким образом, алфавитное письмо бесконечно, а пиктографическое ограниченно. Первое можно масштабировать для представления различных идей, второе – нет.

Такая бесконечность применения справедлива для agile-методологии и неверна для водопадной модели.

Водопадная разработка последовательна. Прежде чем перейти на следующий этап, вам необходимо решить вопросы текущего. Это означает, что каким бы интеллектом мы ни обладали, существует предел, когда сложность совокупной системы превосходит рамки человеческого понимания.

Возможности человеческого разума ограничены, но способность человека понимать – не обязательно. Физиологические ограничения удается преодолеть с помощью технологий, которые мы развиваем и совершенствуем. Мы способны мыслить абстракциями и категориями (модульно) и таким образом расширять степень понимания.

Agile-методология построена на работе с небольшими фрагментами. Легко начать действовать, не имея ответов на все вопросы. Этот подход предполагает достижение успеха, возможно, не самыми лучшими и иногда даже неподходящими способами, но тем не менее благодаря ему на каждом этапе мы узнаем что-то новое.

Мы уточняем свою мысль, определяем следующий этап и затем реализуем его. Agile-разработка безгранична и бесконечна, поскольку мы каждый раз работаем с небольшой областью и двигаемся вперед с уже изученной и понятной позиции. Это более органичный и эволюционный подход к решению задачи.

Именно поэтому agile представляет собой важный шаг вперед как способ решения сложных задач.

Это не значит, что методология agile идеальна или дает все ответы. Но она очень важна для достижения лучшей производительности.

Сила планирования — кажущаяся. Оно не помогает добиться большей точности, управляемости и профессионализма. Это скорее ограничивающий подход, основанный на предположениях и работающий только в небольших, понятных системах с четкой структурой.

Из этого следуют очень важные выводы. Мы должны, говоря словами Кента Бека, которые он использовал как подзаголовок к своей классической книге «*Экстремальное программирование*», «*объять изменения!*»!

Нам необходимо иметь уверенность, когда мы приступаем к работе, не зная ответов и не имея точного понятия об объемах этой работы. Это внушает опасение многим компаниям и специалистам, но на самом деле люди почти всегда ведут себя именно таким образом. Когда кто-то открывает новый бизнес, он не может знать точно, когда к нему придет успех, и придет ли вообще. Ему неизвестно, скольким людям придется по душе новый продукт и будут ли они готовы заплатить за него.

Даже совершая такое обычное действие, как поездка на машине, вы не можете быть уверены, сколько времени точно она займет и действительно ли выбранный маршрут оптимальен. Сегодня у нас есть замечательные инструменты — спутниковые навигаторы, которые не только планируют маршрут, но и итеративно предоставляют обновляемую информацию о трафике, помогая нам оценивать меняющиеся условия поездки и подстраиваться под них (типичный пример «inspect&adapt»).

Итеративный подход к планированию и работе помогает нам всегда иметь самую актуальную картину происходящего, а не некую предположительную и теоретическую не очень точную версию. Благодаря ему мы изучаем изменения, реагируем на них и приспосабливаемся к ним. Итеративный подход — единственная эффективная стратегия работы в меняющейся среде.

ПРАКТИЧЕСКАЯ ЦЕННОСТЬ ИТЕРАТИВНОГО ПОДХОДА

Итак, что необходимо, чтобы применять итеративный подход? Прежде всего, разделить задачу на небольшие этапы. Необходимо сократить область влияния каждого изменения и делать изменения небольшими порциями — чем они меньше, тем лучше. Это позволит чаще испытывать применяемые техники, идеи и технологии.

Поэтапная работа также означает, что мы ограничиваем время активности наших предположений. У мироздания остается меньше времени, чтобы вмешаться в нашу работу, поэтому вероятность возникновения критических последствий гораздо меньше. Наконец, даже если окажется, что очередной наш шаг был ошибочен из-за изменения условий или непонимания с нашей стороны, мы потеряем меньше. Итак, продвигаться небольшими шагами действительно очень важно.

В agile-методологии эта идея очевидным образом воплотилась в итерации, или спринты. Agile подразумевает создание законченного, готового к поставке кода за небольшой фиксированный интервал времени. Полезные следствия, описанные в этой главе, — это только один очевидный и значительный результат итеративного подхода.

Практики непрерывной интеграции (continuous integration, CI) и разработки через тестирование (test-driven development, TDD) также можно представить как итеративные процессы. И это уже совершенно другой масштаб.

В CI изменения фиксируются часто, несколько раз в день. Это означает, что каждое изменение должно быть атомарным, даже если функция, для которой оно предназначено, еще не завершена. Это меняет подход к работе, но дает нам больше возможностей учиться и понимать, по-прежнему ли работает наш код в сочетании с другим кодом.

TDD часто описывается приемами, которые используются: красный, зеленый и рефакторинг.

- Красный: напишите тест, запустите его и убедитесь, что он провалился.
- Зеленый: напишите достаточно кода, чтобы тест прошел, запустите его и убедитесь, что он прошел.
- Рефакторинг: модифицируйте код и тест, чтобы сделать его точным, ясным, выразительным и более общим. Запускайте тест после каждого, даже небольшого, изменения, чтобы убедиться, что он прошел.

Это глубоко детализированный, итеративный подход. Он способствует значительно более итеративному отношению к технической стороне написания кода.

Например, когда я пишу код, я почти всегда совершаю много последовательных крошечных шагов рефакторинга, по пути добавляя новые классы, переменные, функции и параметры и при этом часто проводя тестирование, чтобы удостовериться, что код работает.

Это итеративная работа с очень хорошим разрешением. Это означает, что мой код верный, а значит, каждый шаг безопаснее.

На каждом этапе процесса я могу легко пересмотреть и изменить свое представление о дизайне и коде. Я открыт для изменений!

Именно благодаря этим свойствам итеративный подход так ценен и важен для всей программной инженерии.

ИТОГИ

Итерация — важная техника и основа нашей способности применять более контролируемый подход к обучению, открытиям и разработке лучших программных продуктов. Однако, как всегда, у всего есть цена. Если мы хотим работать итеративно, нам придется во многом изменить привычную организацию своей работы.

Итеративный подход влияет на дизайн систем, которые мы создаем, на то, как мы строим свою работу, и то, как мы структурируем организации, в которых работаем. Идеей итерации пронизаны мысли, стоящие за этой книгой, и модель разработки, которую я в ней описываю. Все идеи глубоко взаимосвязаны, и иногда сложно понять, где заканчивается итерация и начинается обратная связь.

ГЛАВА 5

ОБРАТНАЯ СВЯЗЬ

Обратная связь — это «передача оценочной или корректирующей информации о действии, событии или процессе исходному, или управляющему, источнику»¹.

Без обратной связи нет обучения. Мы можем только строить предположения, но не принимать решения, основываясь на реальных данных. Несмотря на это, удивительно, как мало внимания люди и организации уделяют обратной связи.

Например, многие компании разрабатывают «экономическое обоснование» для нового программного обеспечения. Но сколько организаций затем отслеживают стоимость разработки и оценивают ее наряду с реальными преимуществами, предоставляемыми клиентам, чтобы убедиться, что плановые экономические показатели достигнуты?

Если мы не знаем и не понимаем результатов нашего выбора и действий, мы не можем определить, есть ли прогресс.

Это кажется настолько очевидным, что не стоит даже упоминания, но на практике для принятия решений в большинстве организаций используются догадки, традиции и иерархия.

Обратная связь позволяет установить источник доказательств для принятия решений. Как только у нас появляется такой источник, качество наших решений неизбежно повышается. Благодаря этому мы начинаем отделять мифы от реальности.

¹ Источник: словарь Merriam Webster Dictionary. <https://www.merriam-webster.com/dictionary/feedback>.

ПРАКТИЧЕСКИЙ ПРИМЕР ВАЖНОСТИ ОБРАТНОЙ СВЯЗИ

Иногда трудно понять абстрактные идеи. Приведем простой пример того, насколько важны скорость и качество обратной связи.

Представьте, что вы пытаетесь сбалансировать метлу.

Можно тщательно проанализировать конструкцию метлы, определить ее центр тяжести, внимательно изучить устройство ручки и точно рассчитать точку, где метла будет идеально сбалансирована. Затем надо очень осторожно положить метлу так, как мы рассчитали, и если все будет выполнено правильно, убедиться, что остаточный импульс, из-за которого метла потеряет равновесие, отсутствует.

Этот первый подход соответствует модели водопадной разработки. Наверное, он сработает, но шанс на успех невероятно мал. Результат непредсказуем. Чтобы все получилось, наши расчеты должны быть очень точны, и даже при малейшем отклонении метла упадет.

Второй вариант — положить метлу на руку и корректировать положение руки в зависимости от наклона метлы.

Второй подход основан на обратной связи. Его быстрее внедрить, а его успех будет зависеть от скорости и качества обратной связи. Если мы двигаем рукой слишком медленно, нам придется внести большие корректиры. Если мы будем слишком медленно определять направление наклона метлы, нам придется делать большие поправки, иначе метла упадет. Если обратная связь будет быстрой и эффективной, достаточно небольших поправок, чтобы положение метлы оставалось стабильным. На самом деле, даже если какое-то внешнее воздействие повлияет на равновесие метлы или на наше состояние, нам удастся быстро среагировать и исправить ситуацию.

Этот второй подход настолько успешен, что именно он используется при балансировке тяги двигателей космических ракет. Он так хорошо работает, что если бы я хотя бы немного его освоил, то, наверное, смог удержать метлу в равновесии, даже если бы меня неожиданно толкнули.

Этот второй подход кажется более спонтанным; в некотором смысле он менее строгий, но гораздо более эффективный.

Вы сейчас, наверное, думаете: «Что курит автор? Какое отношение метла имеет к разработке ПО?» Я лишь хочу показать, что в основе работы любых процессов лежит нечто глубокое и важное.

Первый подход — это плановый подход, основанный на предположениях. Он хорошо работает до тех пор, пока вы полностью понимаете все вводные и пока не происходит ничего, что меняет ваши представления или планы. На самом деле это основа любого детального, разработанного процесса. Если у вас есть подробный план, он подразумевает только одно правильное решение, поэтому либо проблема должна быть настолько простой, чтобы это решение стало возможным, либо вы должны быть выдающимся предсказателем будущего.

Второй, альтернативный подход по-прежнему основан на плане «Я собираюсь сбалансировать метлу», но этот план касается только результатов и ничего не говорит о способах их достижения. Вы просто начинаете работать и делаете все возможное, чтобы достичь цели. Если для этого придется, отрабатывая обратную связь, быстро двигать рукой на несколько миллиметров — хорошо. Если для этого нужно сделать несколько широких движений рукой вперед и в сторону, потому что произошло что-то неожиданное, это тоже нормально, если это приведет к успеху.

Второй подход, хотя он и кажется более ситуативным, больше похожим на действия по наитию, на самом деле гораздо более эффективен и стабилен с точки зрения результата. В первом подходе есть только одно правильное решение. Во втором их много, поэтому мы с большей вероятностью достигнем одного из них.

Обратная связь — важная составляющая любой системы, работающей в изменяющейся среде. Создание программного обеспечения — это всегда обучение, и среда, в которой оно происходит, всегда меняется; поэтому обратная связь очень важна для эффективного процесса разработки.

КОНФЕРЕНЦИЯ НАТО¹

К концу 1960-х годов стало очевидно, что программирование — сложная задача. Увеличивались размеры, сложность и важность создаваемых систем. Количество программистов быстро росло. По мере того как специалисты осознавали это увеличение сложности, они начали задумываться, как сделать процесс создания ПО более эффективным и менее подверженным ошибкам.

¹ Источник: отчет NATO Conference on Software Engineering 1968, <https://bit.ly/2rOtYvM>.

В результате в 1968 году была созвана конференция, целью которой стала попытка определить, что такое программная инженерия, а также обсудить методы разработки программного обеспечения.

Конференция проводилась в закрытом режиме, на нее были приглашены мировые эксперты, чтобы обсудить широкий круг идей, касающихся разработки ПО. Учитывая значительный рост вычислительных мощностей за последние 50 лет, некоторые идеи, которые высказывались на той конференции, уже давно устарели, например:

Х. Дж. Хелмс:

Только в Европе установлено около 10 000 компьютеров, и это число увеличивается со скоростью от 25 до 50% в год. Качество программного обеспечения для этих компьютеров скоро затронет более четверти миллиона аналитиков и программистов.

Но другие идеи остаются актуальными:

А. Дж. Перлис:

Ситуация, о которой говорит Селиг, требует петли обратной связи для мониторинга системы. Необходимо собирать данные о производительности системы и использовать их в будущих версиях.

Хотя слова Перлиса касаются устаревших понятий, они подходят для описания современного подхода DevOps к разработке, а не только чего-то написанного на Алголе¹!

Многие другие выступления были столь же пророческими:

Ф. Селиг:

Внешние спецификации на любом уровне описывают программный продукт с точки зрения элементов, контролируемых пользователем и доступных пользователю. Внутренний дизайн показывает программный продукт с точки зрения программных структур, реализующих внешние спецификации. Следует понимать, что обратная связь между дизайном внешних и внутренних спецификаций является неотъемлемой частью реалистичного и эффективного процесса внедрения.

Это очень похоже на современные истории² в agile-разработке, поскольку указывает на важность отделения «что» от «как» в процессе определения требований.

¹ Источник: отчет NATO Conference on Software Engineering 1968, <https://bit.ly/2rOtYvM>.

² Пользовательская история (*user story*) — это неформальное описание функции системы с точки зрения пользователя этой системы. Это одно из понятий экстремального программирования.

На конференции звучали и выступления, в которых видны зачатки со времененных реалий нашей профессии:

д'Агапеев:

Программирование по-прежнему в большой степени художественное занятие. Нам нужна более предметная основа для практического обучения и контроля в отношении:

- (i) структуры программ и последовательности их выполнения;
- (ii) формирования модулей и среды для их тестирования;
- (iii) моделирования условий времени выполнения.

Такие идеи, как «формирование модулей и сред [для облегчения] тестирования» и «моделирование условий выполнения», звучат вполне современно и формируют основы принципа непрерывной доставки.

Изучая материалы конференции, можно заметить, что множество высказанных идей актуальны и по сей день. Они выдержали испытание временем и сегодня так же верны, как и в 1968 году.

Например, «создайте циклы обратной связи» или «предположим, что вы все делаете неправильно» весьма современны, в отличие от, скажем, «используйте язык X» или «для обоснования проекта постройте диаграмму Y».

ОБРАТНАЯ СВЯЗЬ В КОДИРОВАНИИ

Как на практике эта потребность в быстрой и качественной обратной связи влияет на способ организации нашей работы?

Если мы серьезно относимся к обратной связи, мы хотим, чтобы ее было много. Недостаточно просто писать код и полагаться на отчет тестировщиков, который они предоставляют через шесть недель.

Мой собственный подход к написанию кода значительно изменился со временем. Теперь я постоянно использую обратную связь на нескольких уровнях. Я вношу изменения постепенно и в небольшом объеме.

Обычно при создании кода я полагаюсь на тестирование. Если я хочу добавить в систему новое поведение, я сначала пишу тест.

И я хочу знать, верен ли он. Мне нужна обратная связь, подтверждающая это. Поэтому я пишу тест и запускаю его, чтобы убедиться, что он прошел. Анализируя сбой, я получаю обратную связь, которая помогает мне понять, верен ли тест.

Если тест пройден, прежде чем я напишу для этого код, значит, с тестом что-то не так, и мне нужно исправить его, прежде чем двигаться дальше. Все это и есть применение точной обратной связи, позволяющей быстро обучаться.

Как я отмечал в предыдущей главе, я вношу изменения в код последовательно и небольшими порциями. Здесь работают как минимум два уровня обратной связи. Первый — использование инструментов рефакторинга в интегрированной среде, но кроме того, я собираю обратную связь о том, работает ли мой код и, что более субъективно, нравится ли мне результат по мере продвижения, на каждом этапе. В результате мне гораздо проще замечать ошибки и определять неверные шаги.

Этот второй уровень обратной связи возможен потому, что каждый раз, когда я вношу изменения, я могу повторно запустить тест, с которым работаю. Это позволяет мне очень быстро убедиться, что код продолжает работать после изменения.

Такие циклы обратной связи очень короткие или должны быть короткими. Большинство циклов, о которых я здесь говорил, занимают не более нескольких секунд. Некоторые из них, как модульный тест работоспособности системы, делятся миллисекунды.

Короткие, быстрые циклы обратной связи очень важны благодаря скорости и тесной связи с вашим кодом.

Организуя работу в виде последовательности небольших шагов, мы можем лучше оценить скорость продвижения вперед и скорректировать проект так, чтобы добиться лучших результатов.

ОБРАТНАЯ СВЯЗЬ В ИНТЕГРАЦИИ

Когда я фиксирую свой код, он запускает систему непрерывной интеграции и оценивает мои изменения в контексте остальных изменений. В этот момент я получаю обратную связь нового, более глубокого уровня.

Теперь я могу посмотреть, нет ли «утечки» в коде, вызвавшей сбой в другой части системы.

Если на этом этапе все тесты выполнены, я получаю сигнал, что могу приступать к работе над следующим этапом.

Это жизненно важный уровень обратной связи, который позволяет реализовать непрерывную интеграцию.

К сожалению, непрерывная интеграция остается непонятой и недостаточно используется на практике. Если мы применяем интеллектуальный, строгий подход к разработке, инженерный подход важен, чтобы беспристрастно оценить плюсы и минусы рабочих идей. В нашей отрасли это зачастую трудно выполнить. Многие идеи получили широкое распространение, потому что они кажутся лучшими, а не потому, что они действительно лучше.

Хороший тому пример — споры между сторонниками **непрерывной интеграции** (continuous integration, CI) и **ветвлений** (feature branching, FB).

Давайте разберем плюсы и минусы этих подходов с рациональной точки зрения.

Непрерывная интеграция заключается в том, чтобы оценивать каждое изменение в системе вместе с любым другим изменением в ней настолько часто, стремясь к «непрерывному оцениванию», насколько это практически возможно.

Определение CI звучит так:

«(CI) — это практика слияния всех работающих копий в общую основную ветвь разработки несколько раз в день».

Большинство экспертов по CI вместо условия «несколько раз в день» считают возможным, хотя и нежелательным, условие «по крайней мере один раз в день».

Таким образом, по определению CI подразумевает последовательное внедрение небольших изменений для оценки хотя бы один раз в день.

Ветвление любого рода также по определению связано с изоляцией изменений:

«Ветви позволяют участникам изолировать изменения».

В основных, определяющих терминах, CI и FB на самом деле несовместимы друг с другом. Первый метод заточен на то, чтобы применить изменения как можно раньше, второй — чтобы отложить это применение.

FB выглядит просто, и это его свойство привлекает последователей, так как это упрощает жизнь. «Я программирую независимо от товарищей по команде». Проблема возникает в момент объединения изменений. CI придумали, чтобы избежать «адского слияния».

В старые недобрые времена, а в некоторых организациях и по сей день команды и отдельные разработчики писали фрагменты кода до «окончательного завершения», прежде чем объединять их.

На этапе слияния при этом появлялись всевозможные неожиданные проблемы, поэтому процесс был сложным и отнимал массу времени.

Чтобы решить проблему, были разработаны два подхода, первый из которых — CI. Второй — повышение качества инструментов слияния.

Распространенный аргумент приверженцев FB заключается в том, что инструменты слияния теперь настолько хороши, что проблемы почти не возникают. Однако всегда остается риск написать код, который инструменты слияния не обработают; код слияния — это не обязательно то же, что поведение слияния.

Допустим, мы с вами работаем в одной кодовой базе и у нас есть функция, которая выполняет несколько действий для преобразования значения. Мы оба независимо решаем, что нужно увеличить значение этой функции на единицу, но каждый из нас реализует это увеличение в разных частях функции. Вполне возможно, что слияние не опознает связь таких изменений, потому что они находятся в разных частях кода, и в результате будут применены оба. В итоге значение увеличится на два.

Применяя непрерывную интеграцию в соответствии с определением, мы получаем регулярную, частую обратную связь. Таким образом мы получаем представление о состоянии кода и поведении системы в течение рабочего дня, но за это приходится платить.

Чтобы CI действовал, для получения обратной связи приходится вносить изменения достаточно часто. Это означает, что придется действовать совсем по-другому.

Теперь не требуется доводить функцию до состояния, пока она не будет «закончена» или «готова к выпуску». Непрерывная интеграция и ее старшая сестра — непрерывная доставка требуют последовательного внедрения небольших изменений, что ведет к появлению готовых к использованию фрагментов на каждом шаге. Это значительно меняет наши представления о дизайне системы.

В этом случае процесс разработки кода больше похож на управляемую эволюцию, где каждый маленький шаг дает обратную связь, но не обязательно завершается созданием целой функции. Такое очень сложное изменение парадигмы действия ведет к свободе мышления, что положительно влияет на качество проектов.

Это значит, что наш продукт всегда готов к выпуску и мы получаем частую подробную обратную связь о качестве и применимости своей работы, а именно это и побуждает придерживаться такого подхода на протяжении всей разработки.

ОБРАТНАЯ СВЯЗЬ В ДИЗАЙНЕ

Одна из причин, по которой я так высоко ценю разработку через тестирование (test-driven development, TDD) на практике, — это обратная связь о качестве продукта. Если мои тесты сложно писать, это много говорит о качестве кода.

Моя способность создавать простые и эффективные тесты и эффективность дизайна связаны признаками качества, отличающими хороший код. Можно долго спорить о том, что такое хорошее качество кода, но не думаю, что мне стоит это делать, чтобы доказать свою точку зрения. Следующие признаки, на мой взгляд, со всей уверенностью можно считать признаками качества кода; они могут быть не единственными, но уверен, вы согласитесь, что они важны:

- модульность;
- разделение функций;
- высокая связность;

- сокрытие данных (абстракция);
- надлежащая связанность, или сцепление.

Я подозреваю, что вам уже знакомы эти характеристики. Помимо того что они показывают качество кода, они являются инструментами для управления сложностью. И это не совпадение!

Так как же эти свойства помогут сделать код качественным? Без TDD все зависит исключительно от опыта, стремлений и навыков разработчика.

Применяя TDD, мы по определению сначала пишем тест. Если этого не сделать, то это уже не разработка через тестирование.

Если мы собираемся сначала написать тест, то, должно быть, выглядим странными и туповатыми, поскольку сами готовы усложнять себе жизнь. Поэтому постараемся жизнь себе облегчить.

Крайне маловероятно, что тест мы напишем так, чтобы было невозможно получить результаты из тестируемого кода. Поскольку тест создается до кода, это означает, что мы разрабатываем интерфейс для кода. Мы определяем, как внешние пользователи кода будут с ним взаимодействовать.

Поскольку нам требуются результаты для теста, мы сделаем так, чтобы их было легко получить. Это означает, что применение стратегии TDD предписывает создавать код более пригодным для тестирования. Как он будет выглядеть?

Пригодный для тестирования код:

- является модульным;
- предполагает надлежащее разделение ответственности;
- демонстрирует высокую связность;
- использует сокрытие информации (абстракцию);
- является достаточно связанным.

ФУНДАМЕНТАЛЬНАЯ РОЛЬ ТЕСТИРОВАНИЯ

В классических подходах к разработке тестирование иногда оставляли на завершающий этап проекта, иногда на усмотрение заказчика, а иногда настолько сжимали из-за нехватки времени, что, можно считать, почти не проводили.

Это настолько расширило петлю обратной связи, что она стала практически бесполезной. Ошибки в коде или дизайне часто обнаруживались уже после завершения разработки и передачи проекта группе поддержки для обслуживания.

Экстремальное программирование (Extreme Programming, XP) и применение в его рамках стратегий TDD и CI коренным образом изменили эту традицию, выдвинув тестирование на первое место в процессе разработки. Это сократило цикл обратной связи до нескольких секунд, так что ошибки стали выявляться почти мгновенно, что, в свою очередь, помогало, при правильном подходе, устраниć целые классы ошибок, которые ранее, без TDD, часто попадали в продакшен.

Таким образом, тестирование определяло процесс разработки и, что еще более важно, сам дизайн ПО. Программы, написанные с использованием TDD, выглядели иначе, чем программы, написанные без него. Чтобы сделать продукт пригодным для тестирования, важно убедиться, что ожидаемое поведение можно оценить.

Это подтолкнуло дизайн в определенном направлении. Программное обеспечение, которое можно тестировать, отличалось модульностью и слабой связанностью, демонстрировало высокую связность, имело хорошее разделение ответственности и реализовывало скрытие информации. Эти свойства также считаются признанными маркерами качества ПО. Значит, TDD не только оценивала поведение программного обеспечения, но и повышала качество его проектирования.

Тестирование ПО чрезвычайно важно. Немногие продукты столь же чувствительны, как программные. Малейший дефект – запятая не на своем месте – может привести к критической ошибке.

Программное обеспечение также намного сложнее, чем большинство человеческих творений. Современный пассажирский самолет состоит примерно из 4 миллионов деталей. ПО современного грузовика Volvo – это примерно 80 миллионов строк кода, каждая из которых содержит множество инструкций и переменных.

TDD не была новинкой, когда Кент Бек представил ее в своей книге в конце 1990-х годов. Аллан Перлис на конференции НАТО по программной инженерии в 1968 году говорил о чем-то подобном, но именно Бек сформулировал концепцию и описал ее подробно, поэтому она и стала широко известной.

Многие продолжают считать TDD спорным подходом, но данные подтверждают ее эффективность. Этот подход помогает значительно сократить количество ошибок в системе и положительно влияет на качество дизайна.

TDD побуждает создавать код, который объективно имеет более высокое качество независимо от таланта или опыта разработчика. TDD не делает плохих разработчиков ПО великими, но делает плохих разработчиков лучше, а отличных разработчиков — еще лучше.

TDD и другие виды разработки, основанные на тестировании, оказывают существенное влияние на качество создаваемого кода. Это эффект оптимизации для лучшей обратной связи, но этим он не ограничивается.

ОБРАТНАЯ СВЯЗЬ В АРХИТЕКТУРЕ

Менее заметный эффект от подхода, основанного на обратной связи, проявляется при применении его к программной архитектуре систем, которые мы создаем, а также к детальным проектным решениям на уровне кода.

Непрерывная доставка — это высокопроизводительный подход к разработке, основанный на обратной связи. Один из ее краевальных камней — идея, что мы должны создавать продукт, всегда готовый к релизу. Этот стандарт требует частой и качественной обратной связи.

Чтобы этого достичь, организациям приходится значительно менять свой подход к разработке. Два свойства, которые выходят на первый план, можно считать архитектурными качествами систем. Это — **тестируемость** (testability) и **развертываемость** (deployability) систем.

Компаниям, с которыми я работаю, я советую поставлять «продукт, готовый к релизу» не реже одного раза в час. Это означает, что мы должны иметь возможность запускать десятки, а то и сотни тысяч тестов ежечасно.

Если бы мы располагали неограниченным бюджетом и вычислительными мощностями, то могли бы запускать тесты параллельно, чтобы получать быструю обратную связь, но существуют ограничения. Мы можем выполнять тесты независимо от остальных и параллельно с остальными.

Некоторые тесты подразумевают проверку развертываемости и конфигурации системы, поэтому ограничения по времени для обратной связи зависят от длительности развертывания и запуска системы, а также от времени запуска самого медленного теста.

Если выполнение отдельного теста или развертывание ПО занимает больше часа, вам не удастся проводить тесты быстро, сколько бы денег вы ни потратили на оборудование.

Таким образом, тестируемость и развертываемость системы налагают ограничения на способность собирать обратную связь. В наших силах проектировать системы таким образом, чтобы их было легче тестировать и развертывать, а это позволит эффективнее получать обратную связь за более короткие интервалы времени.

Мы бы предпочли, чтобы тесты выполнялись за секунды или миллисекунды, а развертывание — за несколько минут или, лучше, за несколько секунд.

Такие параметры производительности в развертывании и тестируемости требуют командных усилий и концентрации, желания компании внедрить непрерывную доставку, а также продвинутого архитектурного мышления.

Есть два эффективных пути: создавать монолитные системы и оптимизировать их для развертываемости и тестируемости либо разделить систему на отдельные независимые единицы развертывания — микросервисы.

Микросервисы позволяют командам разрабатывать, тестиировать и развертывать сервисы независимо друг от друга; они также разделяют их организационно, что обеспечивает более эффективный и результативный рост компаний.

Независимость микросервисов представляет собой большое преимущество и одновременно большую проблему. Микросервисы по определению являются независимо развертываемыми единицами кода. Это означает, что мы не можем тестиировать их вместе.

Применение непрерывной доставки в монолитных системах имеет эффект, но по-прежнему требует вносить небольшие изменения и оценивать их несколько раз в день. При разработке более крупных систем нам по-прежнему придется работать совместно в одной кодовой базе, поэтому нам требуется защита, которую обеспечит хороший дизайн и непрерывная интеграция.

Решим ли мы разбивать систему на более мелкие и независимые модули (микросервисы) или создавать более эффективные, но и тесно связанные кодовые базы (монолиты), любое решение значительно влияет на архитектуру системы.

Применение техники непрерывной доставки в обоих подходах, монолитном и микросервисном, отдает предпочтение модульным, слабо связанным вариантам дизайна с большей степенью абстракции, потому что только их можно достаточно эффективно развертывать и тестировать в рамках этой концепции.

Это означает, что оценка и приоритизация обратной связи в разработке помогает создавать более разумные и эффективные архитектурные решения.

Это глубокая и важная мысль: приняв некоторые общие принципы, мы сможем добиться значительного и измеримого влияния на качество создаваемых систем. Внедряя процессы, технологии, практику и культуру, основанные на эффективной высококачественной обратной связи, мы будем создавать более качественные программные продукты и делать это значительно эффективнее.

БЫСТРАЯ ОБРАТНАЯ СВЯЗЬ ПРЕДПОЧТИТЕЛЬНЕЕ

Правильный подход — пытаться получить конкретную обратную связь как можно раньше. Когда я пишу код, я использую инструменты разработки, чтобы подсвечивать ошибки по мере печатания кода. Это самый быстрый, дешевый цикл обратной связи и один из самых ценных. Я использую такие средства, как системы проверки ввода, чтобы быстро получить обратную связь о качестве работы.

Я запускаю тест (или тесты) прямо в среде разработки и очень быстро получаю обратную связь — обычно менее чем за несколько секунд.

Мои автоматические юнит-тесты, созданные благодаря TDD, обеспечивают обратную связь второго уровня, когда я регулярно запускаю их в своей локальной среде разработки.

Мой набор юнит- и других тестов будет запущен после коммита. Это позволяет более тщательно, но гораздо дольше проверять, совместим ли мой код с кодом других разработчиков.

Приемочные тесты, тесты производительности, тесты безопасности и все остальное, что мы считаем важным для понимания значимости изменений,

дают дополнительную уверенность в качестве и применимости нашей работы, но за счет увеличения времени на получение результатов.

Таким образом, стремление находить ошибки сначала на этапе компиляции (в вашей среде разработки), затем в юнит-тестах и — только после того как эти проверки будут пройдены успешно — в других высокоуровневых тестах помогает выявить сбой как можно раньше и получить максимально качественную и эффективную обратную связь.

Разработчики, практикующие непрерывную доставку и DevOps, иногда называют этот процесс выявления ранних сбоев *сдвигом влево*, хотя я предпочитаю менее абстрактное «Ошибайся быстро!».

ОБРАТНАЯ СВЯЗЬ В ДИЗАЙНЕ ПРОДУКТА

Серьезное отношение к обратной связи для определения качества систем, которые мы создаем, очень важно и лежит в основе успеха, но в конечном счете разработчикам не платят за то, чтобы их продукт легко тестиировался и имел оптимальный дизайн. Нам платят за то, чтобы мы создавали ценность для компаний, которые нас нанимают.

Это одно из противоречий, которое в большинстве традиционных организаций часто возникает между бизнесменами и техническими специалистами.

Проблему удается решить за счет *непрерывной доставки полезных идей в производство*.

Откуда нам знать, что идеи, которые у нас есть, и продукты, которые мы создаем, хороши?

Мы этого не знаем, пока не получим обратную связь от потребителей (пользователей или клиентов).

Замкнутый цикл обратной связи от идеи продукта до ввода в производство — вот реальная ценность непрерывной доставки. Именно поэтому доставка, а не более узкие (хотя и важные) технические подходы стала так популярна везде в мире.

Применение принципов реализации для получения быстрой и качественной обратной связи позволяет компаниям учиться быстрее; определять,

какие идеи работают, а какие нет; и адаптировать продукты для лучшего удовлетворения потребностей клиента.

Добавление в системы телеметрии, которая собирает данные о том, какие функции и каким образом используются, теперь обычное дело. Сбор информации (обратной связи) от выпускаемых систем помогает не только диагностировать проблемы, но и более эффективно разрабатывать продукты и услуги следующего поколения, превращает организации из «бизнеса и ИТ» в «цифровой бизнес». Способы сбора данных во многих областях настолько усложнились, что данные часто более ценные, чем предоставляемые услуги, поскольку выявляют такие желания, потребности и поведение клиентов, о которых не подозревают даже сами клиенты.

ОБРАТНАЯ СВЯЗЬ В ОРГАНИЗАЦИИ И КУЛЬТУРЕ

Измеримость разработки ПО уже давно стала проблемой. Как мы измеряем успех и улучшения? Как определить, эффективны ли вносимые нами изменения?

На протяжении большей части истории разработки ПО мы либо измеряли то, что было легко измерить (например, строки кода, дни разработки или покрытие тестами), либо строили предположения и принимали субъективные решения на основе интуиции. Проблема в том, что ничего из этого не имеет отношения к успеху, что бы он ни значил.

Больше строк кода не означает лучший код; более того, это скорее означает худший код. Покрытие тестами бессмысленно, если тесты не проверяют что-то полезное. Количество усилий, которые мы вкладываем в разработку, не связано с конечной ценностью продукта. Таким образом, догадки и субъективизм вполне могут оказаться столь же хороши, как и упомянутые метрики.

Так как нам работать лучше? Как наладить получение полезной обратной связи без измерения успеха?

Есть два подхода. Первый сформировался в кругу agile-разработчиков. Мы признаем, что суждения в некоторой степени субъективны, но пытаемся принять некоторый разумный порядок, чтобы нивелировать

субъективность. Успех этого подхода неизбежно связан с людьми. «Люди и взаимодействия важнее процессов и инструментов»¹.

Исторически эта стратегия сыграла важную роль в отходе от шаблонных, формальных подходов к разработке и остается важным принципом.

Agile-подход к разработке предполагает вовлечение команды людей в работу, в петлю обратной связи, чтобы они могли наблюдать результаты своих действий, анализировать их и корректировать действия, чтобы добиться улучшений. Этот субъективный подход, основанный на обратной связи, оказался основополагающим для самой фундаментальной идеи agile — инспекции и адаптации, I&A.

Я бы хотел внести небольшое уточнение в этот субъективный подход к обратной связи, чтобы улучшить ее качество, — уточнить ее природу.

Например, если ваша команда стремится к улучшениям, возьмите лист бумаги и четко обозначьте, где, по вашему мнению, вы находитесь сейчас (текущее состояние) и где бы вы хотели оказаться в будущем (целевое состояние). Опишите шаг, который, как вы считаете, ведет вас в нужном направлении. Решите, как вы будете определять, приближаетесь вы к цели или удаляйтесь от нее. Сделайте шаг и проверьте, туда ли вы движетесь; повторяйте эти действия, пока не добьетесь успеха².

Это простое, легкое и очевидное применение научного метода. Такая практика должна стать «нашим всем», но тем не менее это не то, что внедряется в большинстве организаций. Когда люди применяют этот подход, они добиваются гораздо лучших результатов. Например, эта идея лежит в основе бережливого мышления³, и в частности «пути Toyota», бережливого подхода к производству, который произвел революцию в автомобильной промышленности и других отраслях.

Многие годы я считал, что все, что мы действительно можем сделать, — применять субъективные, но более организованные подходы к решению

¹ «Люди и взаимодействия важнее процессов и инструментов» — один из принципов Agile-манифеста; см <https://agilemanifesto.org/>.

² Майк Ротер подробно описал этот подход в своей книге «Toyota Kata» («Тойота Ката. Лидерство, менеджмент и развитие сотрудников для достижения выдающихся результатов»). СПб, Издательство «Питер»). Хотя, по сути, это просто развитие научного метода.

³ Бережливое мышление — общий термин для идей, связанных с бережливым производством и бережливыми процессами.

проблем. В последние годы мое мнение изменилось благодаря отличным результатам группы Google DORA¹. Теперь я считаю, что их усилия выявили более конкретные, менее субъективные меры, применимые для оценки изменений как в организации и культуре, так и в технической сфере.

Это не означает, что нужно отказаться от прежнего подхода. Необходимо использовать креативность; решения, принятые на основе данных, тоже могут быть ошибочными, но мы в силах подкреплять субъективную оценку данными и добавить количественные характеристики в оценки успеха.

Важны метрики стабильности и пропускной способности, о которых я рассказывал в главе 3. Они не идеальны, и модель, в рамках которой они действуют, является корреляционной, а не причинной. У нас нет доказательств того, что «X вызывает Y»; модель сложнее, чем это утверждение. Существует также множество вопросов, на которые мы хотели бы ответить, оперируя количественными показателями, но не знаем, как это сделать. **Стабильность и пропускная способность** важны, потому что это наиболее понятные показатели, а не потому, что они идеальны.

Тем не менее это огромный шаг вперед. Теперь мы можем использовать метрики эффективности и качества, которые определяют разумность и полезность результатов, для оценки практически любых изменений. Если моя команда решает реорганизовать рабочее место, чтобы улучшить коммуникацию, мы сможем отследить стабильность и пропускную способность и посмотреть, изменятся ли они. Если мы хотим попробовать какую-то новую технологию, поможет ли она создавать программные продукты быстрее, улучшить показатели пропускной способности или повысить качество, чтобы добиться лучшей стабильности?

Эта обратная связь бесцenna как фитнес-функция, которая направляет наши усилия на достижение лучших результатов, предсказанных в модели DORA. Отслеживая показатели стабильности и пропускной способности, когда мы совершенствуем процессы, технологии, организацию и культуру, мы можем быть уверены, что вносимые нами изменения действительно полезны. Мы превращаемся из жертв моды или предположений в инженеров.

¹ Группа DORA разработала научно обоснованный подход к сбору и анализу данных, лежащий в основе «Отчета о состоянии DevOps», который публикуется ежегодно с 2014 года. Их подход и результаты подробно описаны в книге «Ускоряйся! Наука DevOps: как создавать и масштабировать высокопроизводительные цифровые организации».

Эти изменения по-прежнему отражают реальную ценность создаваемых программных продуктов. Ценность проявляется в том влиянии, которое наши изменения оказывают на пользователей. Однако изменения измеряют важные свойства нашей работы и не поддаются манипуляциям. Если у вас хорошие показатели стабильности и пропускной способности, ваша техническая реализация тоже будет хороша. Если вы не добились успеха при хорошей стабильности и пропускной способности, виной тому слабая идея продукта или бизнес-стратегия.

ИТОГИ

Обратная связь необходима, чтобы мы могли учиться. Без быстрой и эффективной обратной связи мы только строим догадки. И скорость, и качество обратной связи одинаково важны. Если обратная связь запаздывает, она бесполезна. Если она вводит в заблуждение или неверна, то решения, которые мы принимаем на ее основе, тоже будут неправильными. Мы часто не задумываемся о том, какая обратная связь нам требуется для обоснования выбора и насколько важны сроки ее получения.

Непрерывная доставка и непрерывная интеграция — это идеи, лежащие в основе оптимизации процесса разработки для максимизации качества и скорости получаемой обратной связи.

ГЛАВА 6

ИНКРЕМЕНТАЛИЗМ

Определение звучит так: «Инкрементальный дизайн — разновидность модульного дизайна, когда элементы можно свободно заменять при их улучшении, чтобы обеспечить более высокую производительность».

Инкрементализм — это постепенное создание ценности. Проще говоря, речь идет о преимуществах модульности, или компонентности, систем.

Если итеративный подход заключается в уточнении и улучшении продукта в течение серии итераций, то инкрементальный — в построении системы и в ее надлежащей поставке по частям. Это прекрасно иллюстрирует рис. 6.1, который я взял из книги Джека Паттона (Jeff Patton) «User Story Mapping»¹.

Итеративный подход



Инкрементальный подход



Рис. 6.1. Итеративный и инкрементальный подходы

¹ Паттон Дж. Пользовательские истории. Искусство гибкой разработки ПО. СПб, Издательство «Питер».

Для создания сложных систем нам нужны оба подхода. Инкрементальный подход позволяет разбивать рабочий процесс на этапы и создавать ценность шаг за шагом (*инкрементно*), увеличивая скорость ее создания, упрощая и сокращая соответствующие шаги.

ВАЖНОСТЬ МОДУЛЬНОСТИ

Модульность — важная идея. Она имеет большое значение для развития технологий и относится не только к ИТ. Когда мастера каменного века делали кремневые топоры с деревянной рукоятью, это была модульная система. Если вы сломали рукоять, вы могли сохранить рабочую часть топора и приделать новую рукоять. Если вы сломали рабочую часть, вы могли привязать новую к старой, надежной рукояти.

По мере того как машины становились все более сложными, важность и ценность модульности тоже возрастили. На протяжении всего XX века, кроме последних нескольких лет, когда авиаконструкторы создавали что-то новое, они делили работу на два основных этапа: создание силовой установки (двигателя) и корпуса летательного аппарата. Авиация в основном развивалась в форме своеобразной технической эстафеты. Если вы хотели испытать новый двигатель, вы сначала испытывали его в проверенном корпусе. Если вы хотели испытать новый корпус, вы использовали проверенный двигатель.

Когда в 1960-х годах для отправки людей на Луну была запущена программа «Аполлон», одним из первых шагов стало создание профиля миссии, которую называли *сближением на лунной орбите* (lunar orbit rendezvous, LOR). LOR подразумевала, что космический корабль будет состоять из модулей, предназначенных для определенной части задачи. Задача «Сатурна-5» заключалась в том, чтобы доставить все остальные модули на орбиту Земли, а затем, на заключительном этапе, другой специальный модуль должен был доставить остальные компоненты космического корабля «Аполлон» с земной орбиты на Луну.

Итак, что делали четыре основных модуля.

- Служебный модуль доставлял все остальные модули с Земли на Луну и обратно.
- Командный модуль служил основным местом пребывания космонавтов; его главной задачей было вернуть астронавтов с орбиты на Землю.

- *Лунный модуль LEM* (Lunar Excursion Module) состоял из двух ступеней: посадочной и взлетной. Посадочная ступень доставляла астронавтов с лунной орбиты на поверхность Луны.
- Взлетная ступень возвращала астронавтов на лунную орбиту, где они встретились и состыковались с командным и служебным модулями перед возвращением на Землю.

Такая модульность имела много преимуществ. Каждый элемент предназначался для работы с частью задачи, и при его разработке требовалось меньше компромиссных решений. Над каждым модулем практически независимо друг от друга могли работать разные команды, а в данном случае совершенно разные компании. После того как команды договорились о совместимости модулей, они могли работать над своими модулями без ограничений. Вес каждого модуля становился меньше, потому что, например, лунный модуль не должен был переносить на поверхность Луны средства для возвращения на Землю.

Хотя сложно назвать космический корабль «Аполлон» простым, каждый модуль оказался проще, чем если бы он был предназначен для решения большей части общей задачи.

Я надеюсь, что вы проведете аналогию с разработкой ПО. Хотя ни одно из этих космических устройств не было простым, они были минималистичны с точки зрения соответствия решаемым задачам.

Это в чистом виде пример проектирования, основанного на компонентах, таких как микросервисы или любой сервис-ориентированный дизайн.

Разделите задачу на части, каждая из которых направлена на решение одной проблемы. У этого подхода масса преимуществ. Каждый элемент системы проще и ориентирован на быстрое решение поставленной задачи. Каждый легче тестировать, быстрее развертывать, а иногда его даже можно развертывать независимо от других. Если вы сможете все это осуществить, — но не раньше, — значит, вы действительно имеете дело с микросервисами.

Однако микросервисы не единственный способ добиться модульности программной системы и извлечь из этого преимущества. В действительности это вопрос серьезного отношения к дизайну.

Используя модульный подход, необходимо тщательно задавать границы между модулями системы и не нарушать их. Эти границы важны;

они представляют собой одно из ключевых проявлений связанности в системе. Если вы уделите достаточно внимания протоколам обмена информацией между модулями, то изолировать работу отдельных модулей и добиться гибкости будет проще.

Я более подробно разберу, что это значит, в следующих главах.

ИНКРЕМЕНТАЛИЗМ В ОРГАНИЗАЦИИ

Одним из огромных преимуществ модульности является изоляция; внутренние детали одного модуля скрыты от других модулей и не имеют к ним отношения. Это важно с технической стороны и еще более важно с организационной.

Модульный подход позволяет командам работать более независимо. Каждая из них небольшими шагами продвигается вперед без необходимости координации или по крайней мере с минимальной координацией между командами. Такая свобода позволяет компаниям, которые ею пользуются, внедрять инновации с беспрецедентной скоростью.

Помимо возможности постепенно вносить технические изменения, этот подход позволяет поэтапно внедрять в компаниях культурные и организационные изменения.

Многие компании изо всех сил пытаются добиться эффективных изменений в организации рабочих процессов. Это очень непросто. Основным препятствием для таких изменений всегда является то, как вы внедряете решение в компании. Известно, что распространение изменений затрудняют два барьера. Первый — как объяснить людям необходимость изменений и мотивировать их на принятие этих изменений, а второй — как преодолеть организационные или процедурные сложности.

Наиболее распространенный подход к внедрению изменений, очевидно, заключается в стандартизации процессов. Картирование процессов и трансформация бизнеса — две крупные сферы применения управлеченческого консалтинга. Проблема в том, что успешность всех компаний, особенно тех, которые что-либо создают, зависит от креативности сотрудников. Если бы нам удалось стандартизировать процесс в виде последовательности шагов, мы могли бы автоматизировать его и исключить из него людей — дорогостоящих, склонных к ошибкам.

Сколько раз вы использовали автоматическую систему распределения телефонных звонков и попадали в меню, в котором не было пункта, соответствовавшего вашему запросу, или которое просто сбрасывало вызов? Это происходило потому, что некоторые процессы непросто разбить на простые модули, и это подтвердит каждый, кто хоть раз писал компьютерную программу.

О том, чтобы исключить креатив из разработки, и речи быть не может. Следовательно, чтобы стимулировать креативность, нам нужно выделить пространство для творческой свободы в процессе и политике, которые структурируют работу. Одним из определяющих признаков высокоэффективных команд является их способность добиваться прогресса и менять направление, не спрашивая разрешения у тех, кто не входит в их малую группу.

Давайте разберемся подробнее. Начнем с малых групп. Хотя теперь у нас больше данных, подтверждающих это утверждение¹, уже давно известно, что небольшие команды превосходят по эффективности крупные коллективы. В своей книге «Мифический человеко-месяц» Фред Брукс писал:

Вывод прост: если над проектом работают 200 человек, включая 25 менеджеров, являющихся наиболее знающими и опытными программистами, увольте 175 бойцов, и пусть менеджеры снова займутся программированием.

В наши дни большинство agile-практиков полагают, что 25 человек — это много. Считается, что оптимальный размер команды — восемь человек или меньше.

Небольшие команды имеют целый ряд преимуществ, но прежде всего именно в малых группах можно добиваться прогресса постепенно, небольшими шагами. Чтобы внедрить организационные изменения, целесообразно создать ряд небольших независимых команд и предоставить им свободу действий. Этот процесс может и должен быть структурирован. Его можно в некоторой степени ограничить, чтобы отдельные, независимые команды двигались примерно в одном направлении развития компании в целом. Но это гораздо более распределенный вид организационной структуры, чем традиционный, принятый в большинстве крупных фирм.

¹ В книге «Ускоряйся! Наука DevOps: Как создавать и масштабировать высокопроизводительные цифровые организации» Николь Форсгрен, Джез Хамбл и Джин Ким описывают признаки высокоэффективных команд.

Таким образом, главная трансформация бизнеса должна заключаться в предоставлении большей автономии отдельным сотрудникам и командам, чтобы они могли выполнять творческую работу с высоким качеством. Ключ к этому — распределенные инкрементальные изменения.

Модульные организации более гибкие, более масштабируемые и более эффективные для разработки ПО, чем традиционные компании.

ИНСТРУМЕНТЫ ИНКРЕМЕНТАЛИЗМА

Пять принципов обучения и пять принципов управления сложностью, которым я следую, тесно взаимосвязаны.

Трудно говорить о каком-то из них, не упоминая другие.

Наиболее действенные инструменты для внедрения инкрементализма — **обратная связь** и **экспериментирование**, но также не нужно забывать о **модульности** и **разделении ответственности** (separation of concerns, SoC).

Но помимо этих глубоких принципов, какие еще идеи помогут реализовать более поэтапный подход к изменениям? Что необходимо предпринять, чтобы начать работать инкрементально?

Инкрементализм и модульность тесно связаны. Если мы хотим вносить изменения инкрементально, мы должны уметь ограничивать их влияние на другие области. Полезно улучшить модульность системы — но как это сделать?

Если мой код представляет собой большой комок грязи и я вношу изменение в одном месте, это может случайно повлиять на другие фрагменты кода. Существуют три способа того, как сделать изменение более безопасным.

Можно спроектировать систему так, чтобы ограничить масштаб изменений. Разрабатывая модульные системы с надлежащим разделением ответственности, вы ограничиваете влияние изменений пределами некой заданной области кода.

Еще один способ — применять практики и методы, которые позволяют корректировать код с меньшим риском. Главный среди таких способов — **рефакторинг**. Это возможность вносить изменения небольшими,

простыми, контролируемыми порциями, тем самым улучшая или по крайней мере изменяя код безопасно.

Разработчики часто недооценивают навыки рефакторинга, упуская из виду их значимость. Если мы будем вносить изменения часто и по чуть-чуть, мы будем гораздо более уверены в стабильности изменений.

Если я использую инструменты рефакторинга в среде разработки, чтобы, скажем, извлечь метод или ввести параметр, то я уверен, что изменение будет безопасным; в противном случае я приобрету лучшие средства разработки.

Крошечные изменения также легко отменить, если я решу, что мне не нравятся результаты. Я могу работать итеративно и инкрементно. Сочетая мелкомодульный инкрементализм с жестким **контролем версий**, я всегда нахожусь рядом с «безопасным местом» и могу откатиться к прежнему состоянию.

Наконец, **тестирование**. Оно, в частности автоматизированное тестирование, дает возможность постепенно двигаться вперед, причем уверенно.

Тонкости эффективной работы с высокими уровнями автоматизированного тестирования мы рассмотрим в следующих главах, но оно очень важно для быстрого и уверенного внесения изменений.

Есть еще один аспект автоматизированного тестирования, который часто упускают из виду те, кто на самом деле редко им пользуется при решении повседневных задач. Это влияние, которое тестирование оказывает на дизайн, и в частности на модульность и разделение ответственности в проектах.

Автоматизированное тестирование требует создания малых исполняемых спецификаций для изменений, которые мы вносим в системы. Каждая из этих небольших спецификаций описывает необходимые условия для начала теста, выполнение тестируемого поведения и результаты.

Учитывая необходимый объем работы, мы сойдем с ума, если не сделаем тесты как можно более простыми, чтобы облегчить себе жизнь. А для этого следует спроектировать систему в виде тестируемого кода.

Поскольку такой код — модульный, с хорошим разделением ответственности, в ходе автоматизированного тестирования создается положительный цикл обратной связи, что помогает проектировать более

совершенные системы, сужает радиус действия ошибок и делает изменения более безопасными. В конечном счете сочетание этих трех методов позволяет значительно усовершенствовать способность вносить изменения инкрементно.

ОГРАНИЧЕНИЕ ВЛИЯНИЯ ИЗМЕНЕНИЙ

Наша цель — управлять сложностью с помощью описываемых методов, поэтому мы разрабатываем системы инкрементно. Добиваться прогресса всегда лучше, делая много маленьких шагов, а не несколько больших и рискованных.

Как мы уже выяснили, если в компании работает несколько небольших групп разработчиков, то ее эффективность будет максимальной, когда эти группы работают независимо друг от друга.

Существуют только две плодотворные стратегии, и обе они носят инкрементный характер.

Мы можем разбить системы на независимые части, как я уже рассказывал в этой главе, или улучшить скорость и качество обратной связи, которую мы собираем в процессе непрерывной интеграции изменений.

Чтобы сделать части системы более независимыми, можно использовать архитектурный шаблон **портов и адаптеров**¹ (Ports & Adapters).

В любой точке интерфейса между двумя компонентами системы, которые мы хотим разделить, — эта точка называется **порт** — мы задаем отдельный фрагмент кода для трансляции входов и выходов — **адаптер**. Это позволяет более свободно изменять код адаптеров без принудительного изменения других компонентов, которые взаимодействуют с ним через порт.

Этот код является ядром логики, поэтому возможность изменить его без координации с другими командами или людьми — большой успех.

¹ Порты и адаптеры — это архитектурный шаблон, цель которого — создание более слабо связанных компонентов приложения; он также известен как гексагональная архитектура.

В результате постепенно удается решить задачи в этой части кода, а затем заняться значительно более сложными и дорогостоящими изменениями в согласованных протоколах обмена информацией между компонентами. Такого рода изменения в идеале должны происходить весьма редко, поэтому командам не придется часто ломать код друг друга.

Необходимо относиться к точкам интеграции, то есть портам, с большей осторожностью, чем к другим частям систем, поскольку внесение в них изменений может вызвать проблемы. Применение архитектуры портов и адаптеров дает возможность внедрить эту «большую осторожность» в наш код.

Обратите внимание, что этот архитектурный шаблон никак не зависит от используемой технологии. Порты и адаптеры не менее — возможно, даже более — полезны для двоичной информации, отправляемой через сокет, чем для структурированного текста, отправляемого через вызов REST API.

Другой важный инструмент управления влиянием изменений, который часто упускают из виду, — скорость обратной связи. Если я напишу код, который сломает ваш код, то масштаб вреда будет зависеть от того, когда именно мы узнаем, что я его сломал.

Если мы обнаружим это через несколько месяцев, последствия могут быть серьезными. Если мы увидим проблему, когда код уже выпущен в продакшен, последствия могут быть очень серьезными.

С другой стороны, если мы выявим сбой в течение нескольких минут после внесения изменений, то большого вреда он не принесет. Я могу решить проблему, которую я создал, до того, как вы ее заметите. Это возможно благодаря механизмам **непрерывной интеграции** и **непрерывной доставки**.

Допустимо использовать любую из этих стратегий или обе, чтобы ограничить влияние изменений. Мы можем спроектировать систему так, чтобы корректировать код самостоятельно, не заставляя других это делать, и оптимизировать методы работы, чтобы вносить эти изменения поэтапно, небольшими порциями. Внесение небольших изменений в общую систему оценки, а затем оптимизация этой системы оценки обеспечивают достаточно быструю обратную связь, чтобы оперативно реагировать на нее и решать любые проблемы, связанные с изменениями.

ИНКРЕМЕНТАЛЬНЫЙ ДИЗАЙН

Я давно являюсь сторонником agile-методологии в разработке ПО. Отчасти это связано с тем, что я рассматриваю agile как важный шаг «начала бесконечности», — об этом я уже говорил в предыдущей главе. Это важно, потому что теперь мы можем начинать работу до того, как получим все ответы. Мы учимся по мере постепенного продвижения вперед, и эту идею я доказываю на протяжении всей книги.

Такой метод бросает вызов предубеждениям многих разработчиков. Многие специалисты, с которыми я разговаривал, отвергали возможность писать код до того, как они будут иметь подробное представление о конечном дизайне.

Еще больше программистов считают инкрементальное проектирование сложной системы почти немыслимым, но оба этих метода составляют основу любого грамотного инженерного подхода.

Сложные системы, полностью готовые к работе, не порождение какого-то гениального творца; они являются плодами работы над ошибками, углубления понимания и исследования идей и потенциальных решений, что иногда очень непростое дело.

Отчасти это сложно, потому что требуется что-то вроде щелчка тумблера в голове, а еще — определенный уровень уверенности, что нам удастся решить проблемы, о которых еще ничего не известно.

Мои рассуждения о том, что такое инженерия и что представляет собой разработка программного обеспечения, призваны помочь щелкнуть этим тумблером, если вы еще этого не сделали.

Уверенность в том, что можно добиться прогресса в ситуации неведения, — это проблема другого рода. В некотором смысле у нее несколько практических решений.

Прежде всего, нужно принять тот факт, что изменения, ошибки и последствия неожиданностей по мере углубления наших знаний просто неизбежны, признаете вы их или нет. Это просто очевидное свойство любой сложной конструкции, особенно в контексте разработки, — это ее дьявольская натура.

Один из признаков — жалобы на то, что «там» всегда неправильно понимают требования. Да, в начале работы никто не знает, что делать. Если «там» говорят, что знают, в действительности «там» не знают ничего.

Признание того, что мы не знаем, сомнения в том, что мы знаем, и стремление быстро учиться — это переход от убеждений к инженерии.

Мы используем новые знания и открытия постепенно и на каждом этапе пытаемся вычислить следующий шаг в неизвестное, основываясь на том, что, как мы полагаем, мы знаем. Это более научное, рациональное мировоззрение. Как однажды сказал физик Ричард Фейнман, «наука — это убедительная философия невежества»:

Ученый обладает большим опытом работы с неведением, сомнением и неуверенностью, и я считаю, что этот опыт чрезвычайно важен.

Методы управления сложностью важны по нескольким причинам, но в контексте разработки как акта открытия они жизненно важны, потому что позволяют ограничить радиус взрыва, когда наш шаг вперед оказывается ошибочным. Можно назвать эти методы защитным дизайном или защитным программированием, но на самом деле это **инкрементальный дизайн**.

Мы можем писать код в виде простой последовательности шагов, организованной или скорее неорганизованной, как большой ком грязи, с плохим разделением на части. Либо мы можем писать код так, чтобы осознавать его сложность и управлять ею по мере ее нарастания.

Если мы выбираем первый путь, то чем больше код связан, чем менее он модульный и менее связный, тем труднее его изменить. Вот почему важны свойства, которые позволяют управлять сложностью кода, — и это я постоянно повторяю. Если мы будем внедрять такие идеи на каждом уровне детализации, мы оставим больше возможностей для будущих изменений, даже неожиданных. Это не означает чрезмерного усложнения и написания универсального кода, который предусматривает любые неожиданности. Это означает написание такого кода, который **облегчит внесение изменений**, а не реализует все возможные задумки.

Предположим, я создаю систему, которая делает что-то полезное и требует, чтобы я где-то сохранял результаты. Я могу сделать то, что и многие

другие разработчики, и объединить код действия с кодом хранения. Если я сделаю это, а затем обнаружу, что мое решение для хранения данных слишком дорогое, содержит много ошибок или слишком медленное, единственное, что мне останется, — пойти и переписать весь код.

Если я отделю функцию «делать что-то полезное» от функции «сохранять результаты», то строк в коде станет больше. Возможно, мне придется хорошенько подумать, как задать разделение, но так я открываю окно возможностей для инкрементной работы и поэтапного принятия решений.

Я не считаю себя нескромным, когда говорю вам, что люди, которые работали со мной, считают меня хорошим программистом. Меня называли программистом в квадрате. Если это так, то не потому, что я умнее других, или печатаю быстрее, или знаю лучшие языки программирования. Это потому, что я работаю инкрементно. Я делаю то, о чем рассказываю в этой книге.

Я избегаю чрезмерной сложности в своих решениях. Я никогда не стараюсь включать в них код для действий, в необходимости которых не уверен. Тем не менее я всегда стараюсь разделять ответственность, делиТЬ систему на части, проектировать интерфейсы, которые выражают идеи, содержащиеся в коде, и скрывают внутренние процессы. Я стремлюсь к простым и очевидным решениям, и у меня есть некая внутренняя сигнальная система, которая срабатывает, когда код начинает казаться слишком сложным, слишком связанным или просто недостаточно модульным.

Я мог бы назвать несколько эмпирических правил, например, что мне не нравятся функции, содержащие более десяти строк кода или более четырех параметров, но это только рекомендации. Я стремлюсь не сжать и упростить код, а написать такой код, который я смогу изменить, когда узнаю что-то новое. Моя цель — постепенно расширяемый код, продолжающий выполнять свою функцию по мере того, как углубляется понимание этой самой функции.

Основа грамотного инженерного подхода — это организация работы таким образом, который позволит свободно менять код и направление движения по мере углубления понимания, и это то, на чем строится инкрементализм. Стремление работать инкрементно также означает стремление создавать более качественные системы. Если код трудно изменить, значит, он низкого качества, какую функцию бы он ни выполнял.

ИТОГИ

Любая сложная система создается поэтапно. Иллюзорное представление — что такие системы возникают полностью сформированными в умах какого-то эксперта или экспертов; это не так.

Они являются результатом работы и постепенного накопления знаний и представлений. Организация работы, направленная на облегчение обучения и подкрепление знаний, позволяет нащупать пока еще невидимые пути к успеху. Эти идеи служат залогом прогресса.

ГЛАВА 7

ЭМПИРИЗМ

Эмпиризм в философии науки определяется как «упор на доказательства, особенно полученные экспериментально. Фундамент научного метода — все гипотезы и теории должны проверяться в наблюдениях за естественной средой, а не основываться исключительно на априорных рассуждениях, интуиции или открытиях».

Согласно этому определению, эмпиризм тесно связан с экспериментом. Тем не менее я оставил оба понятия в своем списке из пяти пунктов, потому что эксперименты можно проводить в таких контролируемых условиях, что это превращается в эксперимент с идеями, которые не реализуются в инженерном смысле.

Даже в современной физической инженерии, со всеми компьютерными моделями и симуляциями, инженеры по-прежнему тестируют реальные конструкции, часто даже разрушая их, чтобы узнать, точны или нет их симуляции. Эмпиризм — ключевое свойство инженерии.

Какое это имеет значение для читателей, которым неинтересно считать ангелов на головке семантической булавки¹?

В отличие от чистой науки, инженерия прочно укоренилась в сфере решения реальных задач. Можно поставить цель: достичь некой архитектурной чистоты — некой вершины производительности, которая потребует изобретения новых методов программирования и их исследования. Но если это все не приведет к осозаемой ценности и если мой программный продукт не сможет приносить больше пользы, то кому нужны такие цели?

¹ How many angels can dance on the head of a pin («Сколько ангелов может танцевать на булавочной головке?») — фраза-пример бессмысленного спора о малозначимых деталях. — Примеч. пер.

ОСНОВАНО НА РЕАЛЬНОСТИ

Однако наши продакшен-системы всегда будут нас удивлять — как и должны! Неприятные сюрпризы беспокоят нас не слишком часто, но любое ПО по сути — это всего лишь то, что сумели сделать его разработчики. Выход на продакшен — это возможность учиться, именно так это надо воспринимать.

Такой важный урок мы можем извлечь из применения науки и инженерии в других отраслях. Один из важнейших аспектов научного, рационального подхода к решению задач — скептицизм. Не имеет значения, кому принадлежит идея, насколько нам важно, чтобы идея сработала, или сколько труда мы в нее вложили, — если идея плоха, то она плоха.

Опыт показывает, что даже в лучших софтверных компаниях лишь часть идей дает ожидаемый эффект.

Функции создают, потому что команды считают их полезными, однако большинство идей не способны улучшить ключевые показатели. Только третья идея, протестированных в Microsoft, справилась со своей задачей и улучшила показатели так, как было задумано¹.

Эмпиризм, принятие решений на основе фактов и наблюдений за реальностью, необходим для достижения значимого прогресса. Без него компании будут продолжать действовать, основываясь только на догадках, и инвестировать в идеи, которые обрачиваются потерей денег или репутации.

ОТДЕЛЯЙТЕ ЭМПИРИЗМ ОТ ЭКСПЕРИМЕНТА

Мы можем действовать эмпирически, используя для принятия решений информацию, которую получаем в ходе экспериментов. Об этом я расскажу в следующей главе. Мы также можем действовать эмпирически, осуществляя менее формальное наблюдение за результатами внедрения наших идей. Это не замена эксперимента, а скорее способ улучшить понимание ситуации, когда мы обдумываем следующие эксперименты.

¹ В статье под названием «Online Experiments at Large Scale» (<https://stanford.io/2LdjvmC>) авторы описывают, как от двух до трех третей идей по изменению ПО принесли нулевую или отрицательную ценность для организаций, внедривших эти изменения.

Я осознаю, что, исследуя идеи эмпиризма и эксперимента по отдельности, рискую углубиться в дебри философии и этимологии. Это не входит в мои намерения, поэтому я проиллюстрирую на практических примерах, почему эти два тесно взаимосвязанных понятия стоит рассматривать независимо друг от друга.

«Я ЗНАЮ ОШИБКУ!»

Несколько лет назад мне довелось получить незабываемый опыт — я принял участие в создании с нуля одной из самых эффективных финансовых бирж в мире. Именно в этот период своей карьеры я начал серьезно применять инженерный подход в разработке.

Мы собирались запустить релиз в производство, когда обнаружили серьезную ошибку. Для нас это было довольно необычно. Команда использовала методологии, описанные в этой книге, в том числе непрерывную доставку, поэтому мы все время получали обратную связь от постоянно вводимых небольших изменений. Мы редко сталкивались с крупными проблемами.

Наш релиз-кандидат проходил финальную проверку. Ранее в тот же день один из наших коллег, Даррен, сообщил, что заметил странный сбой обмена сообщениями на своей рабочей станции при выполнении приемочных тестов API. Очевидно, он обнаружил заблокированный поток в базовом коде обмена сообщениями стороннего разработчика. Он попытался дублировать его, и ему это удалось, но только на одной сопряженной станции. Это показалось странным, потому что конфигурация среды была полностью автоматизирована, а управление версиями осуществлялось с помощью довольно сложного подхода «инфраструктура как код».

Позже в тот же день мы начали работу над следующим набором изменений. Почти сразу же конфигурация значительно изменилась и многие приемочные тесты не были выполнены. Мы начали изучать, что происходит, и заметили, что один из наших сервисов показывает очень высокую загрузку процессора. Это было необычно, потому что наше ПО в целом было исключительно эффективно. Дальше мы заметили, что наш новый код для обмена сообщениями, по-видимому, завис. Именно это, должно быть, и видел Даррен. Очевидно, у нас возникла проблема с новым кодом!

Мы отреагировали немедленно — сообщили, что релиз-кандидат может быть не готов к выпуску, и решали, стоит ли взять ветку, чего мы обычно старались избегать, чтобы откатить изменения.

Но затем мы остановились и подумали: «Подождите, это не имеет смысла; мы запускаем этот код больше недели и теперь видим этот сбой уже трижды за пару часов».

Мы остановили работу и обсудили ситуацию заново, собрав все данные, что были доступны. Мы обновили обмен сообщениями в начале итерации; кроме того, у нас был дамп потока, показывавший, что обмен сообщениями завис; то же сообщал и Даррен, но его дамп остановился в другом месте. Мы не раз в течение недели успешно запускали все эти тесты в пайплайне развертывания и добавляли изменения в обмен сообщениями.

Казалось, это тупик. Наша гипотеза о сбое в обмене сообщениями не подтверждалась фактами. Нам требовалось больше данных, чтобы вывести новую гипотезу. Мы начали снова с того места, где обычно начинали решать проблемы, но не сделали этого в данном случае, потому что всеказалось очевидным. Мы охарактеризовали проблему, поэтому начали собирать данные, чтобы воспроизвести по порядку, что произошло. Мы просмотрели файлы журнала и нашли там, как вы уже, наверное, догадались, исключение, явно указывающее на какой-то совершенно новый код.

Коротко говоря, с обменом сообщениями все было в порядке. Очевидная проблема оказалась симптомом, а не причиной. На самом деле дамп потока находился в обычном состоянии ожидания и работал как надо. Мы столкнулись с ошибкой многопоточности в новом коде, не связанном с обменом сообщениями. Ее легко найти и просто исправить, и мы бы нашли ее за пять минут, если бы не полагали, что это проблема обмена сообщениями. Нам действительно удалось исправить ее за пять минут, как только мы остановились, чтобы подумать, и сформулировали гипотезу на основе фактов, а не сделали неверный, но кажущийся очевидным вывод.

Только когда мы остановились и обратились к фактам, мы поняли, что наши поспешные выводы им не соответствовали. Именно это и только это, побудило нас собрать еще больше данных — достаточно, чтобы решить реальную проблему, а не ту, которую мы себе вообразили.

У нас была сложная автоматизированная система тестирования, но мы игнорировали очевидное. Было ясно, что мы сделали что-то, что сломало билд. Но мы объединили несколько разрозненных фактов, что породило неверный вывод: последовательность событий повела нас по ложному пути. Мы построили теорию на песке, не проверяя ее, а на-сливая новые предположения поверх старых. Это привело к тому, что мы приняли за истину, казалось бы, очевидную причину, но она была совершенно неверной.

Наука работает! Сформулируйте гипотезу. Подумайте, как ее доказать или опровергнуть. Проведите эксперимент. Оцените результат и убедитесь, что он соответствует гипотезе. Повторите!

Вывод таков: быть эмпириком сложнее, чем кажется, и для этого нужно больше дисциплины. Когда мы сопоставляли проблему, которую видел Даррен, с неудачными тестами, то должны были действовать эмпирически и реагировать на реальные факты. Но не сделали этого. Мы торопились с выводами и подгоняли факты под предположения. Если бы мы более системно прошлись по тому, что знали, то наверняка бы поняли, что это вовсе не проблема обмена сообщениями, потому что изменения работали всю неделю и что-то делать с ними не было нужды.

ИЗБЕГАЙТЕ САМООБМАНА

Эмпирическое поведение требует от нас большего внимания к сигналам, получаемым из окружающей среды, и к тому, как мы строим на их основе теории, которые можно проверить с помощью экспериментов.

Человек — существо выдающееся... но чтобы быть таким умным, нужно постоянно анализировать различные операции. Наше восприятие реальности — это не сама реальность, и у нас есть своего рода биологические уловки, позволяющие сделать восприятие целостным. Например, наша визуальная частота дискретизации на удивление низка. Плавность восприятия картинки глазами — это иллюзия, созданная мозгом. На самом деле глаза сканируют небольшой участок поля зрения примерно один раз в пару секунд, а мозг достраивает «виртуальную реальность».

Большая часть того, что мы видим, — это предположения, которые делает наш мозг. Это важно: мы эволюционировали, чтобы обманывать самих себя. Мы делаем поспешные выводы, потому что если бы мы занимались подробным и точным анализом того, что видят глаза, в те времена, когда боролись за выживание, хищник сожрал бы нас раньше, чем мы успели бы закончить этот анализ.

Различные виды когнитивных упрощений и нарушений, которые мы развили за миллионы лет, позволили нам выживать в реальном мире. Однако на смену опасной саванне, населенной хищниками, пришла современная высокотехнологичная цивилизация, и мы разработали более эффективный способ решения проблем. Он медленнее, чем поспешные, но зачастую неправильные выводы, но гораздо эффективнее, иногда даже при решении невероятно сложных задач. Ричард Фейнман очень емко охарактеризовал науку:

Первый принцип заключается в том, что вы не должны обманывать себя, а ведь вас обманут проще всего.

Наука — это не то, чем ее считает большинство людей. Это не большие адронные коллайдеры, не современная медицина и даже не физика. Наука — это метод решения проблем. Мы создаем модель проблемы и проверяем, соответствует ли ей то, что мы знаем на данный момент. Затем мы пытаемся доказать, что модель неверна. По словам Дэвида Дойча, модель состоит из «хороших объяснений»¹.

ИЗОБРЕТАЙТЕ РЕАЛЬНОСТЬ, СООТВЕТСТВУЮЩУЮ АРГУМЕНТАМ

Вот еще пример того, как легко обмануть самих себя.

Пока мы создавали нашу сверхбыструю биржу, мы много экспериментировали с созданием очень быстрых программных продуктов, и обнаружили массу интересного. Самым примечательным был подход, который мы назвали **механической симпатией**.

¹ Дойч Дэвид. Начало бесконечности.

Его суть в том, что код создается с учетом принципов работы аппаратной части, чтобы в дальнейшем использовать ее преимущества. Один из наших важных выводов был таков: после устранения тривиальных ошибок¹ на непосредственную производительность фрагмента кода на современном компьютере главным образом влияет промах кэша (*cache-miss*).

Таким образом, работая над важными высокопроизводительными частями кода, мы стали тщательно следить за отсутствием кэш-промахов.

И обнаружили, что одна из самых частых причин кэш-промаха для большинства систем — параллелизм.

В те времена в разработке бытовало такое общепринятое мнение: «Физические возможности оборудования приближаются к своим пределам, что означает, что скорость процессора больше не увеличивается. Поэтому чтобы проекты работали хорошо, их необходимо выстраивать параллельно».

Эта тема освещалась в ряде академических статей; специально разрабатывались языки для упрощения параллельного программирования, чтобы сделать его основным подходом для решения повседневных задач. На самом деле, как мы доказали, эта модель имеет множество недостатков, но здесь я разберу только один. В то время обсуждался академический язык, предназначенный для автоматического распараллеливания решений².

Возможности этого языка были продемонстрированы на примере выделения слов из потока символов при обработке текста книги. Учитывая наш опыт и уверенность, что параллелизм требует больших затрат, по крайней мере когда необходимо объединять результаты различных параллельных потоков выполнения, мы были настроены скептически.

У нас не оказалось доступа к академическому языку, но один из моих коллег, Майк Баркер, провел простой эксперимент. Он написал тот же алгоритм, который предложили ученые, на Scala, а также простой брутфорс на Java, а затем оценил результат, сделав серию прогонов текста «Алисы в Стране чудес» Льюиса Кэрролла.

¹ Самая распространенная ошибка производительности — использование неправильной структуры данных для хранения. Многие разработчики не учитывают время поиска разного рода коллекций. Для небольших коллекций простой массив ($O(n)$ при извлечении) может быть быстрее, чем что-то вроде хеш-таблицы (с семантикой $O(1)$). Для больших коллекций лучше всего подходит решение $O(1)$ для прямого доступа. После этого реализация коллекций может приобрести стоимость.

² Презентация об автоматическом распараллеливании: <https://bit.ly/35JPqVs>.

Параллельный алгоритм на Scala был реализован в 61 строке кода; версия Java заняла 33 строки. Код на Scala мог обрабатывать 400 копий книги в секунду. Это впечатляет; но более простой, легко читаемый однопоточный код на Java обрабатывал 1600 копий в секунду.

Исследователи языка начали с теории, что параллелизм эффективен, но они настолько увлеклись реализацией, что даже не подумали проверить свое исходное предположение о том, что результат будет быстрым. Результат оказался более медленным, а код — более сложным.

ПРИМЕР: ОТДЕЛЯЯ МИФЫ ОТ РЕАЛЬНОСТИ

Очевидно, что развитие CPU достигло предела и тактовая частота уже не увеличивается. Она не росла примерно с 2005 года! Тому есть веские физические причины, связанные с изготовлением кремниевых чипов. Существует зависимость между плотностью пластины и теплом, которое она выделяет при работе. Чип, обладающий скоростью выше 3 ГГц, вполне может расплавиться.

Поэтому если невозможно добиться прироста скорости за счет увеличения частоты линейной обработки инструкций в CPU, можно их распараллелить, что и сделали производители процессоров. Это хорошо: современные процессоры — замечательные устройства, но как мы используем такую мощь? Работая параллельно!

Это подходит для несвязанных независимых процессов, но что, если нужен быстрый алгоритм? Очевидный вывод (предположение) таков: необходимо распараллелить алгоритмы. Идея в том, чтобы ускорить работу, создавая несколько потоков выполнения для задач, которые мы решаем.

Чтобы было удобнее создавать параллельные решения, было разработано несколько языков программирования общего назначения, построенных на данном предположении.

К сожалению, это гораздо более сложная проблема, чем кажется. Для решения некоторых необычных задач параллелизм действительно подходит. Однако как только возникает необходимость снова объединить информацию из потоков, все меняется.

Проанализируем данные обратной связи, вместо того чтобы делать поспешные выводы о том, что распараллеливание всегда эффективно.

Попробуем что-нибудь несложное. Например, напишем алгоритм для увеличения простого целого числа в 500 миллионов раз.

Без обратной связи кажется очевидным, что для решения этой задачи можно создать множество потоков. Однако результаты эксперимента вас удивят.

МЕТОД	ВРЕМЯ МЕТОДА (МС)
Одиночный поток	300
Одиночный поток с блокировкой	10 000
Два потока с блокировкой	224 000
Одиночный поток с CAS	5 700
Два потока с CAS	30 000

В таблице показан результат экспериментов, проведенных с использованием различных подходов. Прежде всего, базовый тест. Напишите код в одном потоке и увеличивайте значение числа. Чтобы получить 500 млн, потребуется 300 мс.

Как только мы добавим код для синхронизации, увидим затраты, которых не ожидали (если только мы не эксперты по параллелизму низкого уровня). Если по-прежнему вся работа будет выполняться в одном потоке, но мы добавим блокировку, позволяющую использовать результаты из другого потока, это прибавит еще 9700 мс. Блокировка стоит дорого!

Если разделить работу между двумя потоками и синхронизировать их результаты, это окажется в 746 раз медленнее работы в одном потоке!

Итак, блокировка – чересчур дорогое удовольствие. Существуют более сложные в использовании, но более эффективные способы координации работы потоков. Самый эффективный – использование низкоуровневой инструкции сравнения с обменом (Compare-and-Swap, CAS). К сожалению, даже это в 100 раз медленнее, чем работа с одним потоком.

С помощью обратной связи мы можем принимать информированные решения, основанные на доказательствах. Если мы хотим, чтобы алгоритм работал с максимальной скоростью, мы должны выполнять основную часть операций в одном потоке.

(Этот эксперимент впервые провел Майк Баркер, с которым мы работали вместе несколько лет назад.)

Пример из врезки иллюстрирует несколько основных концепций, о которых я рассказываю в этой книге. Он демонстрирует важность обратной связи, экспериментов и эмпиризма.

ОПИРАЙТЕСЬ НА РЕАЛЬНОСТЬ

У исследователей в описанном сценарии были благие намерения, но они попали в распространенную ловушку: выдвинули предположение, разработали решение, а затем бросили все силы, чтобы внедрить его, не проверив, верно ли это предположение.

Чтобы показать, что решение неэффективно, Майку понадобилось несколько часов — он написал код, опираясь на данные исследователей. Критическое отношение к собственным идеям дается трудно, но это единственный способ добиться реального прогресса.

Лучший способ начать — **предположить, что то, что вы знаете и о чем думаете, неверно**, а затем выяснить, как определить, *в чем может быть ошибка*.

«Академики» в приведенной истории купились на миф, не подкрепленный реальными фактами. Они построили модель распараллеливания языков программирования, потому что, решив такую задачу, можно показать, какой ты крутой специалист.

К сожалению, при этом не учитывались затраты на параллелизм — реалии современного оборудования и компьютерной науки были проигнорированы. Давно известно, что параллелизм дорого обходится, когда нужно объединить результаты. Закон Амдала показывает, что существует жесткое ограничение на число одновременных операций, если только они не полностью независимы друг от друга.

Академики предположили, что больше параллелизма — это хорошо, но эта идея годится для реализации на какой-то воображаемой теоретической машине, у которой затраты на параллелизм низкие. А таких машин нет.

Эти ученые не эмпирики, хотя и экспериментаторы. Отсутствие эмпиризма означало, что их эксперименты были ошибочными, поэтому модель, которую они построили, не соответствовала реальному опыту.

Эмпиризм — это механизм, с помощью которого мы можем проверить достоверность экспериментов. Их следует поместить в контекст и, по сути, проверить, верна ли симуляция реальности, лежащей в основе экспериментов.

ИТОГИ

Инженерия, в отличие от чистой науки, требует учитывать, насколько практически решения, которые мы разрабатываем. Именно здесь в игру вступает **эмпиризм**. Недостаточно только наблюдать и делать предположения, основываясь на том, что мы видим, а затем считать их правильными только потому, что их подтверждает информация из реального мира. Это плохая наука и плохая инженерия. А инженерия — практическая дисциплина. Поэтому необходимо постоянно подвергать сомнению предположения и эксперименты, которые мы проводим для проверки этих предположений, сверяя эксперименты с реальным опытом.

ГЛАВА 8

БЫТЬ ЭКСПЕРИМЕНТАТОРОМ

Эксперимент определяется как «процедура, проводимая для подтверждения, опровержения или проверки гипотезы. Эксперименты выявляют причинно-следственные связи, показывая, какой результат влечет за собой изменение того или иного фактора».

Экспериментальный подход к решению задач имеет огромное значение. Я бы сказал, что наука и эксперименты, лежащие в ее основе, — это основное отличие нашего современного высокотехнологичного общества от предшествовавших аграрных обществ. Как отдельный вид люди существуют сотни тысяч лет, и тем не менее за последние 300 или 400 лет, со времен Ньютона или Галилея, которые большинство считает началом современной науки, мы достигли такого прогресса, которого не добились за весь предшествующий период. Есть мнение, что объем человеческих знаний в современном мире удваивается каждые 13 месяцев¹.

В значительной степени это связано с применением самых эффективных за всю историю методов решения задач.

Однако разработка ПО на самом деле обычно так не ведется. В основном это упражнение в мастерстве, когда кто-то гадает, что может понравиться пользователям. Специалист высказывает идею, что такой-то дизайн и/или технология окажутся эффективными. Затем разработчики предполагают, выполняет ли код, который они пишут, свои функции и есть ли в нем ошибки. Многие компании пытаются понять, полезен ли их продукт и принесет ли он больше денег, чем стоило его создание.

¹ Бакминстер Фуллер создал кривую удвоения знаний: <https://bit.ly/2WiyUbE>.

Мы можем сделать лучше. Мы можем использовать предположения там, где они уместны, а затем провести эксперименты для проверки этих предположений.

Это кажется медленным, дорогим и сложным, но это не так. На самом деле это всего лишь сдвиг в образе действия и мышления. Речь не о том, чтобы работать усерднее; речь о том, чтобы работать интеллектуальнее. Команды, которые организовали свою работу таким образом, отнюдь нельзя назвать медленными или слишком академичными. Они более дисциплинированы и в результате быстрее находят лучшие и более дешевые решения и создают продукты более высокого качества, которые больше нравятся пользователям.

ЧТО ЗНАЧИТ БЫТЬ ЭКСПЕРИМЕНТАТОРОМ?

Одна из ключевых идей, лежащих в основе научного мышления, не оглядываясь на авторитеты. У Ричарда Фейнмана, как всегда, есть отличные слова на эту тему:

Наука есть вера в невежество экспертов.

Он также сказал:

Не уважайте авторитеты; забудьте, кто это сказал, лучшие посмотрите, с чего он начинает, чем заканчивает, и спросите себя: «Это разумно?»

Несмотря на несколько дерзкую форму выражения, он прав.

Мы не должны принимать решения, основываясь только на мнении авторитетного, харизматичного или известного человека, даже если это Ричард Фейнман; мы должны основываться на доказательствах.

Такой подход подразумевает большие изменения для нашей отрасли. К сожалению, это относится и к обществу в целом, а не только к разработке ПО, поэтому если мы хотим быть успешными инженерами, мы должны работать лучше, чем общество в целом.

Что побудило вас выбрать язык программирования, который вы используете, или фреймворк, или редактор, в котором вы пишете свой код? Спорите

ли вы об относительных достоинствах Java по сравнению с Python? Вы считаете всех, кто использует редактор VI, умными или дураками? Верите ли вы, что функциональное программирование — единственный верный путь или же что объектно-ориентированное программирование — это лучшее изобретение человечества? Да, я тоже!

Я не предлагаю для каждого решения проводить исчерпывающий, контролируемый эксперимент, но мы должны прекратить войны по этим поводам.

Если мы хотим доказать, что Clojure лучшие С#, почему бы не провести небольшой тест и не измерить стабильность и пропускную способность результата? По крайней мере, тогда победят доказательства, пусть даже несовершенные, а не громкость голосов спорщиков. Если вы не согласны с результатами, лучшие проведите эксперимент и приведите аргументы.

Быть экспериментатором не означает основывать каждое решение на строгих доказательствах. Все науки основаны на эксперименте, но с разной степенью контроля. В основе инженерии также лежит эксперимент, но в прагматичной, практической форме.

Вот четыре основные характеристики экспериментального подхода.

- **Обратная связь:** относиться серьезно к обратной связи и понимать, как получать результаты, которые обеспечивают четкое понимание. Нам нужно замкнуть цикл.
- **Гипотеза:** не забывать идею, которую мы проверяем. Мы не блуждаем в потемках, собирая случайные данные. Это не имеет смысла.
- **Измерение:** иметь четкие критерии оценки предположений, которые мы проверяем в гипотезе. Что означает успех или неудача в контексте этой гипотезы?
- **Контроль переменных:** исключить как можно больше переменных, чтобы понять результаты эксперимента.

ОБРАТНАЯ СВЯЗЬ

Важно понимать, какой эффект с инженерной точки зрения может дать рост эффективности и качества обратной связи.

ЖАЖДА СКОРОСТИ

Когда-то я работал в компании, производившей сложное программное обеспечение для трейдинга. Разработчики были толковыми, и компания добилась успеха, но все знали, что способны на большее, и моя работа заключалась в том, чтобы помочь улучшить методы разработки.

Когда я присоединился к команде, уже было внедрено достаточно эффективное автоматизированное тестирование. Разработчики проводили много тестов. Они выполняли ночную сборку – основная часть их продукта представляла собой большой билд C++, что занимало 9,5 часа, включая тесты. Поэтому сборку запускали каждую ночь.

Один из разработчиков сказал мне, что за три года работы было всего три случая, когда прошли все тесты.

Таким образом, каждое утро команда выбирала модули, все тесты которых завершились успешно, и делала их релиз, а модули, тесты которых не прошли, придерживала.

Все шло хорошо, когда успешные модули не зависели от изменений в модулях, не прошедших тест, но иногда такая зависимость была.

Я многое хотел изменить, но мы начали с повышения эффективности обратной связи, ничего больше не меняя.

После долгих экспериментов и напряженной работы нам удалось выйти в быструю стадию, выполнить коммит за 12 минут, а остальные тесты – за 40 минут. Раньше на эту же работу требовалось 9,5 часа! Кроме ускорения сборки и более эффективного получения результатов, других изменений в организации, процессах или инструментах мы не делали.

В первые две недели после этого изменения были две сборки, в которых все тесты завершились успехом. В следующие две недели и все оставшееся время, что я там работал, каждый день была как минимум одна сборка, в которой все тесты выполнялись и весь код был пригоден к релизу.

Не пришлось делать никаких изменений, кроме повышения скорости обратной связи, – этого оказалось достаточно, чтобы команды получили инструменты, необходимые для исправления нестабильности.

Байка под названием «Жажда скорости» — хороший пример того, как применять техники экспериментирования, а также оптимизацию для получения отличной обратной связи. В данном случае мы экспериментировали, чтобы повысить эффективность и качество обратной связи с разработчиками. Мы определили максимальные показатели производительности сборки, использовали улучшенный контроль версий переменных

и принцип «инфраструктура как код», а также провели А/В-тестирование нескольких технических решений и систем.

Только благодаря достаточно дисциплинированному экспериментальному подходу к решению этой задачи, которую прежде уже пытались решить разными способами, мы добились успеха. Несколько наших идей не сработали. Эксперименты показали, что нет смысла тратить время и усилия на некоторые инструменты и методы, потому что они не дадут необходимого ускорения.

ГИПОТЕЗА

Говоря о науке и инженерии, люди часто упоминают «избавление от догадок». Каюсь, я тоже прежде использовал эту фразу. Однако это неправильно. В каком-то смысле наука и основана на догадках и предположениях; просто научный подход институционализирует их и называет гипотезами. Как красноречиво выразился Ричард Фейнман в своей замечательной лекции о научном методе¹:

*Мы ищем новый закон следующим образом: **сначала мы его угадываем!***

Предположения, или гипотезы, являются отправной точкой. Разница между наукой и инженерией по сравнению с другими, менее эффективными, подходами заключается в том, что другие на этом и останавливаются.

В рамках научного подхода, как только мы получаем предположение в форме гипотезы, мы начинаем делать прогнозы, а затем пытаемся найти способы их проверить.

В своей прекрасной презентации Фейнман продолжает так:

Если ваша догадка не согласуется с экспериментом, то она неверна!

В этом суть! Вот куда нам нужно добраться, чтобы заявить, что то, что мы делаем, — это инженерия, а не догадки.

¹ Нобелевский лауреат физик Ричард Фейнман о научном методе: <https://bit.ly/2RiEivq>.

Мы должны иметь возможность проверить наши гипотезы. Тесты бывают разными. Можно или наблюдать реальное поведение (продакшн), или проводить контролируемый эксперимент, например, в виде автоматизированного теста.

Мы можем получать качественную обратную связь от продакшена, чтобы использовать ее для обучения, или тестиировать идеи в более контролируемой среде.

Если мы организуем свою работу в виде серии экспериментов для подтверждения гипотез, мы значительно улучшим качество создаваемого продукта.

ИЗМЕРЕНИЯ

Независимо от способа получения данных, к их оценке следует подходить серьезно. Необходимо понять, что означают данные, и оценить их критически.

Пытаясь сопоставить факты с данными, легко обмануться. Мы можем защититься от таких ошибок, тщательно продумав в рамках нашего эксперимента, какие показатели имеют значение. Нам требуется сделать прогноз на основе гипотезы, а затем решить, как измерить результаты.

На ум приходит множество примеров измерения незначимых показателей. Один из моих клиентов решил, что улучшит качество кода, увеличив количество тестов. Он начал внедрение измерений, собрал данные и утвердил стратегию, предполагающую увеличение покрытия тестами. Он поставил цель добиться 80-процентного покрытия тестами. Далее он использовал этот показатель в программе стимулирования разработчиков, привязав их бонусы к достижению заданного процента покрытия.

Что было дальше? Он достиг цели!

Некоторое время спустя клиент проанализировал процесс тестирования и обнаружил, что более 25% тестов вообще ничего не проверяли. В результате он платил разработчикам бонусы за то, чтобы они писали тесты ради тестов.

В этом случае гораздо эффективнее было бы измерять стабильность. Клиенту требовалось не большее количество тестов, а более качественный код, поэтому прямое измерение сработало бы лучше.

Трудно определить, какие показатели измерять, и это относится не только к метрикам и умению людей понимать систему.

Я более десяти лет занимался финансовыми системами с низкой задержкой. Когда мы начинали, мы уделяли очень много внимания измерениям задержки и пропускной способности, поэтому работали над тем, чтобы фиксировать результаты измерений. Мы ставили задачи наподобие «система должна обрабатывать 100 000 сообщений в секунду с задержкой не более 2 мс». Первые версии наших продуктов мы основывали на средних значениях, которые, как позже выяснилось, не представляли никакой ценности. Нам надо было определить конкретные значения. Как показало будущее, случалось так, что пиковая нагрузка в торговом цикле намного превышала соответствующую скорость 100 000 сообщений в секунду, достигая пиков в миллионы сообщений в секунду. Средняя задержка не имела значения, если экстремальные значения выходили за установленные границы. В реальном мире высокочастотного трейдинга значение 2 мс не было средним — оно было предельным!

В этом примере мы только начинали становиться экспериментаторами. Но благодаря точности измерений, хотя кое-что мы делали неверно, мы быстро учились, и это позволило нам повышать качество и точность измерений и ставить более верные цели. **Все дело в обучении!**

Не всем важна супроточность измерения, но принцип остается тем же, какой бы продукт вы ни создавали. Быть экспериментатором — значит уделять больше внимания измерениям параметров системы, что бы это ни значило в конкретном контексте.

УПРАВЛЕНИЕ ПЕРЕМЕННЫМИ

Чтобы получить обратную связь и провести полезные измерения, нужно контролировать переменные настолько, насколько это практически возможно. Когда Джез Хамбл и я написали книгу *«Continuous delivery»*, мы дали ей подзаголовок *«Reliable Software Releases Through Build, Test, and Deployment Automation»* («Автоматизация процессов сборки, тестирования

и внедрения новых версий программ»). Вряд ли в то время я думал именно так, но на самом деле эта фраза означает следующее: «Управляйте переменными, чтобы сделать релизы надежными».

Контроль версий обеспечивает точность изменений, которые мы осуществляем. Автоматизированное тестирование позволяет точнее оценить поведение, скорость, надежность и качество продукта, который мы создаем. Автоматизация развертывания и отношение к **инфраструктуре как коду** способствует более точному определению среды, для которой предназначено программное обеспечение.

Все эти методы позволяют нам быть гораздо более уверенными в том, что когда мы запускаем наше ПО в производство, оно будет делать именно то, что мы планировали.

Я считаю, что **непрерывная доставка** — это обобщенный подход к разработке, который позволяет действовать с гораздо большей уверенностью. Применяя его, мы избавляемся от значительной части показателей качества работы и можем сосредоточиться на оценке того, хороши ли наши идеи. Мы начинаем более четко понимать, делаем ли мы хороший продукт, потому что мы контролируем то, что делаем его правильно.

Контролируя многие технические параметры, непрерывная доставка позволяет добиваться прогресса увереннее, чем раньше. Таким образом, разработчики получают реальные преимущества методов оптимизации в целях обучения, о чём я и рассказываю в этой книге.

Например, пайплайн развертывания с непрерывной доставкой — идеальная экспериментальная платформа для изучения изменений, которые мы планируем внести в продукты.

Работая так, чтобы продукт всегда был готов к релизу, мы руководствуемся принципом, лежащим в основе непрерывной доставки, и этот принцип максимизирует обратную связь о качестве нашей работы, которую мы можем получить, и стимулирует нас работать небольшими шагами. Это, в свою очередь, означает, что нам приходится работать итеративно и поэтапно.

Если продукт всегда готов к релизу, глупо не воспользоваться этим! Так организации смогут выпускать версии чаще и раньше получать больше обратной связи о качестве своих идей и, следовательно, создавать более качественные продукты.

АВТОМАТИЗИРОВАННОЕ ТЕСТИРОВАНИЕ КАК ЭКСПЕРИМЕНТ

Эксперименты могут принимать разные формы, но у разработки ПО есть огромное преимущество перед любой другой отраслью — фантастическая экспериментальная платформа под названием компьютер!

Мы имеем возможность проводить миллионы экспериментов каждую секунду, если захотим. А эксперименты могут принимать самые разные формы. Мы можем рассматривать этап компиляции как форму эксперимента: «Я предполагаю, что мой код будет компилироваться без предупреждений об ошибках» или: «Я предполагаю, что мой UI-код не будет обращаться к библиотеке базы данных». Однако наиболее гибкой формой эксперимента в разработке считается автоматизированный тест.

Любой автоматизированный тест будет считаться экспериментом, если вы достаточно постараетесь. Однако если вы пишете тесты после создания кода, ценность опыта снижается. Эксперимент должен основываться на гипотезе, а предположение о том, работает ваш код или нет, — довольно хромая гипотеза.

Я считаю, что необходимо организовывать разработку в виде серии небольших повторяемых опытов, которые предсказывают поведение кода и позволяют постепенно увеличивать функциональность продукта.

Самая очевидная форма такого эксперимента — разработка программного обеспечения на основе тестов, или **разработка через тестирование** (test-driven development, TDD).

TDD — это эффективная стратегия, когда мы используем тесты в качестве исполняемых спецификаций поведения системы. Вот наша гипотеза: «Когда в этом конкретном контексте происходит то-то и то-то, мы ожидаем такого-то результата». Мы формулируем этот прогноз в виде небольшого простого теста, а затем, в ходе эксперимента после написания кода, подтверждаем, что прогноз оказался верным.

TDD используется на разных уровнях детализации. Можно начать с создания спецификаций, ориентированных на пользователя, с применением методов **разработки через приемочное тестирование** (acceptance test-driven development, ATDD), иногда их называют **разработкой через поведение** (behavior-driven development, BDD). Эти высокоуровневые

исполняемые спецификации применяются для проведения более подробного и технически ориентированного модульного тестирования.

Продукты, разработанные посредством таких методов, содержат значительно и измеримо меньше ошибок, чем разработанные традиционными способами¹.

Это желанное улучшение качества, но основную ценность представляет влияние снижения ошибок на производительность. В результате команды разработчиков потратят значительно меньше времени на обнаружение, сортировку и анализ ошибок.

Например, высокопроизводительные команды, использующие такие методы, как TDD, непрерывная интеграция и непрерывная доставка, уделяют полезной работе на 44% больше времени². Эти команды гораздо продуктивнее, а их результаты более качественные. Усидеть на двух стульях реально!

Практика экстремального программирования в контексте непрерывной доставки, непрерывной интеграции и TDD обеспечивает замечательную, масштабируемую экспериментальную платформу для оценки и улучшения идей разработки и реализации. Эти методы оказывают существенное влияние на качество работы и скорость создания хорошего продукта. Это именно те результаты, которые в других отраслях являются заслугой инженерии.

ПОМЕЩАЯ РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ В КОНТЕКСТ

Прошу прощения за минутку философии, но полагаю, что вы уже привыкли.

Подумаем, что на самом деле означают тесты, подобные тем, что я описал выше.

¹ Существует несколько академических и неофициальных исследований влияния TDD на сокращение количества ошибок. Большинство исследователей сходятся во мнении, что сокращается от 40 до более чем 250% ошибок. Источник: <https://bit.ly/2LDh3q3>.

² Источник: отчеты State of DevOps (разные годы) и «Ускоряйся! Наука DevOps: Как создавать и масштабировать высокопроизводительные цифровые организации» Н. Форстгрен, Д. Хамбл и Дж. Ким.

Я утверждаю, что научная рациональность является руководящим принципом подхода, который я здесь представляю. Распространенная ошибка разработчиков и, возможно, людей вообще заключается в том, что как только мы говорим о науке, мы почти всегда имеем в виду физику.

Я горячий поклонник физики. Я люблю физику и мысленные модели, которые я строю с ее помощью, чтобы понимать окружающий мир. Я иногда шучу, что физика — единственная истинная наука, но на самом деле, конечно, так не считаю.

Наука гораздо шире физики, но другие дисциплины выходят за рамки упрощенных абстракций, лежащих в основе физики, и зачастую более запутаны и менее точны. Это не умаляет ценности научного стиля рассуждения. Биология, химия, психология и социология тоже науки. Они не так точны, как физика, потому что не подразумевают столь же строгого контроля показателей в эксперименте, но все же дают более глубокое понимание и лучшие результаты, чем ненаучный подход. Я ни в коем случае не ожидаю от разработчиков ПО такой же тщательности и точности, как от физиков.

Тем не менее у разработки ПО есть несколько существенных преимуществ перед инженерией почти в любой отрасли, а также перед некоторыми науками, где проводить эксперименты сложно по этическим или практическим причинам. Мы можем сами создавать и полностью контролировать «вселенную», в которой обитают наши продукты. Мы осуществляем тонкий и точный контроль, если захотим. Мы способны проводить миллионы низкозатратных экспериментов, что позволяет нам пользоваться всей мощью статистики. Если упростить, то именно это на самом деле представляет собой современное машинное обучение.

Благодаря компьютерам мы можем контролировать наши продукты и экспериментировать с ними в масштабах, невообразимых в любой другой отрасли.

Наконец, у разработки есть еще одна, довольно глубокая способность.

Представим на мгновение, что мы физики. Если нам придет в голову новая идея, как узнать, хороша ли она? У нас должно быть достаточно знаний, чтобы понять, что она согласуется с принятыми в физической науке фактами. Не стоит предполагать, что «Эйнштейн ошибался», если мы не знаем, что сказал Эйнштейн. Физика — обширная область знаний,

так что как бы хорошо мы ни были осведомлены, нам необходимо достаточно четко описать идею, чтобы другие могли ее воспроизвести и проверить. Если идея не прошла проверку и мы убедились, что ошибок в тесте не было, мы можем отказаться от идеи.

В разработке ПО все эти этапы занимают считанные минуты и реализуются в виде тестов. Это наша суперсила!

Если мы представим наш продукт, каким бы большим или сложным он ни был, существующим в крошечной «вселенной», которую мы создаем, то сможем четко контролировать эту «вселенную» и оценивать роль продукта в ней. Если мы работаем, чтобы иметь возможность управлять переменными до такой степени, чтобы в надежной форме раз за разом воссоздавать эту «вселенную» — например, инфраструктуру как часть пайплайна развертывания с непрерывной доставкой, — у нас есть хорошая отправная точка для экспериментов.

Полный набор всех написанных нами тестов, включая набор экспериментов, подтверждающих поведение системы в этой контролируемой «вселенной», представляет собой свод знаний о системе.

Мы можем дать определение «вселенной» и свод знаний любому, и тот подтвердит, что они в целом непротиворечивы — все тесты выполняются успешно.

Если мы хотим создать новое знание в системе, мы можем провести еще один эксперимент или тест, а затем добавить это знание в виде рабочего кода, отвечающего условиям эксперимента. Если новые идеи не согласуются с предшествующими, то есть со сводом знаний в нашей контролируемой «мини-вселенной», то эксперименты завершатся неудачей и мы поймем, что идея неверна или по крайней мере несовместима с принятым утверждением.

Теперь я понимаю, что это несколько идеалистическое представление о разработке и тестах, но я участвовал в создании нескольких систем, которые были очень близки к такому идеалу. Однако даже если вы одолели только 80% пути, подумайте, что это значит. Вы можете проверить правильность и последовательность своих идей для всей системы за несколько минут.

Как я уже говорил, это наша суперсила. Это то, что нам доступно, если относиться к разработке как к инженерному процессу, а не только как к ремеслу.

ОБЪЕМ ЭКСПЕРИМЕНТА

Эксперименты бывают разных размеров — от маленьких и незначительных до крупномасштабных и сложных. Иногда нам нужны и те и другие, но необходимость проводить серию экспериментов некоторых очень пугает.

Чтобы успокоить вас, я расскажу об одном частом эксперименте, который провожу регулярно.

Практикуя TDD, я начинаю менять код с теста. Цель — создать тест, завершающийся ошибкой. Чтобы убедиться, что тест действительно проверяет определенную функцию, нам нужно, чтобы он завершился с ошибкой. Итак, я начинаю с теста. Когда тест готов, я задаю *точное сообщение об ошибке*, с которым тест должен завершиться: «Я ожидаю, что этот тест завершится с ошибкой “получено значение 0 вместо x” или что-то по-добное. Это эксперимент; это применение научного метода.

- Я обдумал проблему и охарактеризовал ее: «Я определился, какое поведение требуется от системы, и использовал его в качестве тестового кейса».
- Я выдвинул гипотезу: «Я ожидаю, что мой тест провалится!»
- Я сделал прогноз: «В случае сбоя будет получено следующее сообщение об ошибке...»
- Я провел эксперимент: «Я запустил тест».

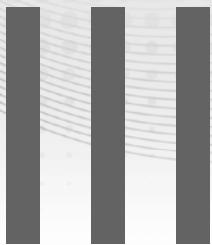
Это небольшое изменение привычки, но оно значительно улучшило качество моей работы.

Более дисциплинированная работа и проведение экспериментов не должны быть сложными или обременительными. Если мы хотим стать инженерами-разработчиками, следует признать правильность такого подхода и последовательно применять его в работе.

ИТОГИ

Основной признак экспериментального подхода — степень контроля над используемыми переменными. В определении эксперимента, которое я привел в начале этой главы, есть слова «показать, какой результат вызывает изменение того или иного фактора». Экспериментальный подход предполагает большую степень контроля в работе. Мы хотим, чтобы результаты наших экспериментов оказались надежными. В техническом плане систем, которые мы создаем, работая экспериментально и контролируя все возможные переменные, эффективное автоматизированное тестирование и методы непрерывной доставки, такие как «инфраструктура как код», обеспечивают большую надежность и воспроизводимость экспериментов. А если смотреть глубже, они также делают наши продукты более детерминированными и, следовательно, более качественными, более предсказуемыми и надежными в использовании.

Любой подход к разработке, достойный называться инженерным, должен приводить к созданию лучшего продукта с теми же усилиями. Это достигается, если организовать работу в виде последовательности множества небольших и, как правило, простых экспериментов.



ОПТИМИЗАЦИЯ ДЛЯ УПРАВЛЕНИЯ СЛОЖНОСТЬЮ

ГЛАВА 9

МОДУЛЬНОСТЬ

Модульность определяется как «степень, в которой могут быть разделены и объединены компоненты системы, часто с целью гибкости и разнообразия в использовании»¹.

Я занимаюсь программированием уже долгое время, и с самого начала, когда я был студентом и писал простые видеоигры на ассемблере, модульность считалась важной характеристикой при проектировании кода.

Тем не менее большая часть кода, который я встречал, — на самом деле подавляющая его часть и, возможно, даже часть кода, который написал я сам, — была далека от модульной. Но в какой-то момент в моей практике все изменилось. Мой код теперь всегда модульный; это его неотъемлемое свойство.

Модульность имеет ключевое значение для управления сложностью систем, которые мы создаем. Современные программные системы — обширные, сложные и часто действительно комплексные. Большинство современных систем превосходят возможности памяти и внимания любого человека.

Чтобы справиться с этой сложностью, мы должны разделить системы на более мелкие, более понятные части, с которыми мы можем работать, не слишком беспокоясь о том, что происходит в других частях.

Это верно всегда, и это опять своего рода фрактал, который работает с разной степенью детализации.

¹ Источник: определение из словаря Merriam-Webster.

Разработка ПО совершенствуется. Когда я начинал карьеру, компьютеры и программы были проще, но нам приходилось работать больше. Операционные системы обеспечили доступ к файлам и возможность отображать текст на экране — всего лишь. Остальное приходилось писать с нуля для каждой программы. Хотите что-нибудь распечатать? Вам нужно разобраться и написать код для низкоуровневых взаимодействий с конкретным принтером.

Мы определенно продвинулись вперед, усовершенствовав абстракцию и модульность операционных систем и других программных инструментов.

Тем не менее многие системы сами по себе не выглядят модульными. Это потому, что разрабатывать модульные системы сложно. Если разработка подразумевает обучение, то по мере того как мы учимся, наше понимание развивается и меняется. Так что возможно — и, скорее всего, именно так — наше мнение о том, какие модули нужны, а какие нет, тоже изменится.

Для меня это показатель настоящего мастерства разработки. Именно модульность отличает код, написанный специалистами, мастерами своего дела, от остального кода. Хотя чтобы добиться модульности в наших проектах, требуется навык. А судя по коду, который я встречал, люди не просто плохо реализуют модульность, они вообще не пытаются это сделать. Они пишут код как рецепт, в виде линейной последовательности шагов, объединяемых в методы и функции, содержащие сотни или даже тысячи строк.

Представьте на мгновение, что в вашей кодовой базе есть функция, которая отклоняет любой коммит, содержащий метод длиннее 30, 50 или 100 строк кода. Пройдет ли ваш код такой тест? Я уверен, что большая часть кода, который я видел, не пройдет.

Когда я начинаю новый проект, я задаю проверку в пайплайне развертывания непрерывной доставки на этапе коммита, которая выполняет именно такой тест и отклоняет любой коммит, содержащий метод длиннее 20 или 30 строк кода. Я также избавляюсь от сигнатур методов более чем с пятью или шестью параметрами. Это значения, которые я для себя определил, исходя из своего опыта и опыта команд, в которых работал. Их не обязательно придерживаться, скорее подобные направляющие важны, чтобы оставаться честными. Как бы ни было мало времени, плохой код никогда его не экономит!

ПРИЗНАКИ МОДУЛЬНОСТИ

Как узнать, является ли система модульной? В упрощенном смысле модуль можно трактовать как набор инструкций и данных, которые включены в продукт. Это своего рода «физическое» представление модуля в виде совокупности битов и байтов.

Однако с практической точки зрения мы ищем то, что делит наш код на небольшие фрагменты. Каждый фрагмент можно использовать много-кратно, в том числе в разных контекстах.

Код в модуле достаточно короткий, чтобы его было легко понять как автономный, вне контекста других частей системы, даже если эти части выполняют полезную работу.

Область действия переменных и функций, которые ограничивают доступ к ним, определенным образом контролируется, так что в некотором смысле существует «внешняя среда» и «внутренность» модуля. Существует некий интерфейс, который контролирует доступ, управляет связью с другими фрагментами кода и с другими модулями.

НЕДООЦЕНКА ВАЖНОСТИ ХОРОШЕГО ДИЗАЙНА

Есть несколько причин, по которым многие разработчики не обращают внимания на идеи, изложенные выше. Как индустрия, мы недооценили важность продуктового дизайна. Мы одержимы языками и фреймворками. Мы спорим, что лучше – IDE или текстовые редакторы или объектно-ориентированное программирование по сравнению с функциональным. Тем не менее ни одно из этих понятий не настолько важно, как, например, модульность или разделение ответственности за качество результата.

Если ваш код обладает хорошей модульностью и хорошим разделением ответственности, какими бы ни были парадигма программирования, язык или инструменты, он будет лучше, с ним легче работать, его проще тестировать и изменять по мере того, как вы будете больше узнавать о задаче и улучшать решение. Он также окажется более гибким в использовании, чем код, не обладающий этими свойствами.

У меня сложилось впечатление, что либо нас вообще не учат этим навыкам, либо в программировании (или программистах) есть что-то, что заставляет нас пренебрегать их важностью.

Очевидно, что модульное проектирование — это навык, отличный от знания синтаксиса языка программирования. Это навык, над которым нужно работать, если мы надеемся добиться мастерства, и мы можем потратить на это всю жизнь и, возможно, так никогда и не довести его до совершенства.

Для меня это суть разработки. Как создавать код и системы, которые будут расти и развиваться с течением времени и при этом разделены на фрагменты, чтобы ограничить влияние потенциальной ошибки? Каким образом разделить системы, чтобы границы между модулями представляли собой возможности для улучшения, а не препятствия, мешающие вносить изменения?

Это важный вопрос, который я пытаюсь решить в книге.

Однажды я вел курс по TDD. Я пытался продемонстрировать, как TDD помогает уменьшить сложность проектов, когда один из слушателей курса (я не буду называть его программистом) спросил, почему так важно, чтобы код был менее сложным. Признаюсь, я был потрясен. Если этот человек не видел, какую разную ценность создают запутанный и сложный либо ясный и простой код, то у него абсолютно иное представление о нашей работе, чем у меня. Я постарался ответить на его вопрос, рассказав о важности обслуживания и преимуществах с точки зрения эффективности, но не уверен, что мои доводы попали в цель.

По сути, сложность увеличивает стоимость владения программным продуктом. Это имеет как прямой экономический эффект, так и более субъективный: работать над сложным кодом не так уж и приятно!

Еще одна проблема заключается в том, что сложный код по определению труднее изменить. Это означает, что у вас есть один шанс сделать все правильно — когда вы пишете его в первый раз. Кроме того, если мой код сложен, то я, вероятно, не так хорошо его понимаю, как мне кажется; в нем больше мест, где могут скрываться ошибки.

Если мы стараемся ограничить сложность кода при разработке, допустимо делать ошибки, потому что у нас больше шансов их исправить. Так что либо мы делаем ставку на собственную гениальность и надеемся, что у нас

все получится с самого начала, либо действуем более осмотрительно. Мы начинаем с предположения, что в ходе работы возникнет что-то, что мы не учли; а еще возможны и другие изменения; значит, скорее всего, нам придется корректировать код. Сложность стоит дорого!

Важно, что мы открыты для новых идей. Необходимо постоянно подвергать сомнению предположения. Однако это не означает, что все идеи имеют одинаковую ценность. Есть глупые идеи, и их следует отбрасывать; есть отличные идеи, и их нужно ценить.

Знания синтаксиса языка недостаточно, чтобы считаться программистом, не говоря уже о том, чтобы считаться хорошим программистом. Отлично владеть языком X менее ценно и менее важно, чем обеспечивать качественный дизайн. Знание сложных деталей API Y не делает вас лучшим разработчиком: вы всегда можете где-то посмотреть ответ на свой вопрос!

Настоящие навыки, которые действительно отличают хороших программистов от плохих, не зависят от языка или фреймворка. Они находятся в другой плоскости.

Любой язык программирования — всего лишь инструмент. Мне посчастливилось работать с несколькими программистами мирового класса. Эти люди могут писать хороший код на языке, который они никогда раньше не использовали. Они напишут хороший код на HTML и CSS, в сценариях оболочки Unix или YAML. Один из моих друзей даже пишет на понятном Perl!

Есть идеи более глубокие и фундаментальные, чем язык, используемый для их выражения.

Модульность — одна из этих идей; если ваш код не является модульным, он почти наверняка не так хорош, как код, который является таковым!

ВАЖНОСТЬ ТЕСТИРУЕМОСТИ

Я одним из первых стал использовать TDD, первые пробные шаги в этом направлении я сделал в ответ на книгу Кента Бека «Экстремальное программирование», опубликованную в 1999 году. Моя команда попробовала интригующий метод Кента, и это привело нас к ошибке в том же году. Тем не менее это был чрезвычайно полезный опыт.

TDD — один из самых значительных рывков вперед, который сделала практическая разработка за всю мою карьеру. При этом довольно странно, что причина, по которой я так высоко ценю этот подход, не имеет почти ничего общего с тестированием в привычном смысле этого слова. На самом деле теперь я думаю, что Кент Бек совершил ошибку, включив слово «тест» в название этой практики, по крайней мере с точки зрения маркетинга. И нет, я не знаю, как ему следовало ее назвать!

В главе 5 я рассказал, как получить быструю и точную обратную связь о качестве дизайна путем тестирования и как тестируемость кода повышает его качество. Это чрезвычайно важная идея.

Быструю и точную обратную связь о качестве дизайна можно получить не только благодаря хорошему вкусу опытного, квалифицированного программиста. О том, хорош или плох наш дизайн, мы можем узнать недели, месяцы или годы спустя, когда попытаемся его изменить, но помимо этого не существует объективных показателей качества, если только мы не станем основывать дизайн на результатах тестов.

Если писать тесты сложно, значит, дизайн плохой. Это немедленный сигнал. Мы получаем обратную связь о качестве дизайна, когда пытаемся усовершенствовать его. Мы получаем ее автоматически, если соблюдаем принципы TDD **«красный», «зеленый», «рефакторинг»** (red, green, refactor). Когда тесты трудно писать, это свидетельствует о том, что дизайн хуже, чем должен быть. Если тесты пишутся легко и просто, проект, который мы тестируем, будет иметь те свойства, которые мы ценим как признаки высокого качества кода.

Это не означает, что, используя подход, основанный на тестировании, мы автоматически создадим великолепный дизайн. Это не волшебная палочка. Мастерство и опыт дизайнера по-прежнему важны. Хороший разработчик все равно сделает лучше, чем плохой. Основывая дизайн на тестах, мы быстрее создадим тестируемый код и системы, что, учитывая ограничения опыта и способностей, дает лучший результат.

Мне не приходит на ум никакая другая техника, которая в такой же степени эффективна! Этот **усилитель таланта** — важный инструмент перехода от ремесла к инженерии.

Если мы стремимся стать инженерами, то советовать людям делать лучше недостаточно. Нужны инструменты, которые будут организовывать нас и помогут достигать лучших результатов. Одним из таких инструментов является создание тестируемых систем.

ТЕСТИРУЕМОСТЬ ПОВЫШАЕТ МОДУЛЬНОСТЬ

Вернемся к теме книги и подумаем о ней в контексте модульности. Как проектирование пригодных для тестирования систем способствует их большей модульности?

Если я хочу испытать эффективность аэродинамического профиля крыла самолета, я могу построить самолет и отправиться в полет. Идея настолько ужасна, что даже братья Райт, которые построили первый управляемый летательный аппарат, не решились на это.

Если вы будете следовать такому довольно наивному подходу, вам придется проделать всю работу, прежде чем вы чему-то научитесь. Как вы оцените эффективность одного аэродинамического профиля по сравнению с другим? Постройте еще один самолет?

Даже если так, как вы сравните результаты? Возможно, во время полета на первом самолете ветер был более порывистым. Или пилот поплотнее позавтракал. Возможно, менялось давление или температура воздуха, поэтому подъемная сила крыла была разной. Иногда партии топлива различаются, поэтому двигатель создает разную мощность. Как управлять всеми этими показателями?

Если вы придерживаетесь целостно-системного, водопадного подхода к решению задачи, сложность системы расширится на всю среду, в которой функционирует аэродинамический профиль.

Чтобы оценить аэродинамический профиль с научной точки зрения, нужно взять под контроль эти переменные и стандартизировать их в экспериментах. Как уменьшить сложность, чтобы добиться наглядных результатов эксперимента? Например, поместить оба самолета в более контролируемую среду, скажем, в большую аэродинамическую трубу. Это позволит точнее контролировать воздушный поток. Возможно, мы могли бы найти среду с регулируемой температурой и давлением. Только при таком контроле мы можем рассчитывать на воспроизводимые результаты.

Если уж на то пошло, на самом деле не нужен ни двигатель, ни система управления полетом, ни сам самолет. Почему бы просто не изготовить две модели крыльев с аэродинамическими профилями, которые мы хотим протестировать, и не испытать их в аэродинамической трубе с регулируемыми температурой и давлением?

Это, безусловно, более корректный эксперимент, чем полет, но все равно придется строить два комплекта крыльев. Почему бы не изготовить маленькую модель каждого профиля? Сделайте каждую модель как можно более точной, используя одни и те же материалы и техники, и сравните их. Если мы уменьшим масштаб, то подойдет более простая аэродинамическая труба.

Такие части самолета являются модулями. Они выполняют свою конкретную отдельную функцию. Правда, эксперименты с ними дают лишь частично верную картину. Аэродинамика самолетов определяется не только крыльями. Но модульность подразумевает лучшую измеримость, поэтому часть или модуль, безусловно, больше пригодны к тестированию, чем целый самолет.

В реальном мире именно так можно определить, каким образом форма крыльев и другие факторы влияют на подъемную силу.

Модульность обеспечивает больший контроль измеримых параметров и большую точность измерений. Перенесем этот пример в мир разработки ПО. Представьте, что вы создаете систему В, которая находится ниже по потоку, чем система А, и выше, чем С (рис. 9.1).

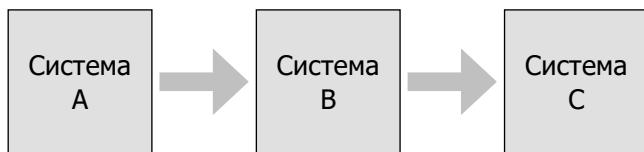


Рис. 9.1. Связанные системы

Это довольно типичная ситуация в крупных организациях, и это ведет к проблеме: как проверить нашу работу? Многие, а может быть, даже большинство организаций, столкнувшихся с такой задачей, решают тестировать всё целиком, чтобы убедиться, что система безопасна в использовании.

Такой подход вызывает много проблем. Прежде всего, если вести измерения в масштабе всей конструкции, мы получаем задачу «самолет целиком». Вся система настолько сложна, что мы не добьемся точности, воспроизводимости, контроля и четкого представления о том, что на самом деле означают полученные результаты.

Мы не сможем точно оценить интересующую нас часть системы, потому что нам будут мешать части, расположенные выше и ниже в потоке, система А и система С. При использовании такого подхода множество тестов просто не проходят.

Что произойдет с системой В, если система А отправит ей некорректное сообщение? Этот случай невозможно проверить, пока реальная система А отправляет корректные сообщения.

Как должна реагировать система В при сбое канала связи с системой С? Опять же, мы не можем тестировать этот сценарий, пока система С работает нормально и не дает имитировать ошибку связи.

Результаты, которые мы сможем получить, малоинформативны. Почему тест завершается с ошибкой? Проблема в нашей системе или в одной из двух других? Возможно, сбой означает неправильную версию систем выше или ниже по потоку. Если тест пройден, означает ли это, что мы готовы к релизу? Или кейсы, которые мы оцениваем, настолько упрощены из-за того, что сложная система не поддается тестированию, что ошибки просто-напросто не удается обнаружить?

При измерении всей сложной системы (рис. 9.2) мы получим неочевидные и спорные результаты. Довольно примитивная диаграмма на рис. 9.2 иллюстрирует важную проблему: необходимо четко представлять, что именно мы измеряем и какова ценность измерений. Когда мы проводим сквозные тесты, представленные на этой диаграмме, какова цель тестирования? Что мы надеемся увидеть? Если цель — убедиться, что все фрагменты работают вместе, это может быть полезно в некоторых контекстах, но такого тестирования недостаточно, чтобы понять, действительно ли система В, которую мы сейчас создаем, работает. Такого рода тесты имеют смысл только в качестве дополнения к более тщательной модульной стратегии тестирования. Только подробное тестирование позволит убедиться в правильной работе системы В!

Так что же необходимо для более детального тестирования? Нам понадобится **точка измерения**, то есть место в системе, где можно разместить датчики. На следующих диаграммах я изобразил эти точки измерения в виде штангенциркулей. На самом деле мы говорим о возможности вводить тестовые данные в нашу **тестируемую систему** (system under test, SUT), вызывать требуемое поведение и собирать выходные данные, чтобы интерпретировать результаты. Я знаю, что штангенциркули выглядят странно, но именно они пришли мне на ум, когда я размышлял о тестировании. Чтобы оценить

систему, я размещу ее в **тестовой среде**. Мне потребуется измерительное устройство (тестовые кейсы и тестовая инфраструктура), чтобы встроить датчики в тестируемую систему и наблюдать за ее поведением.

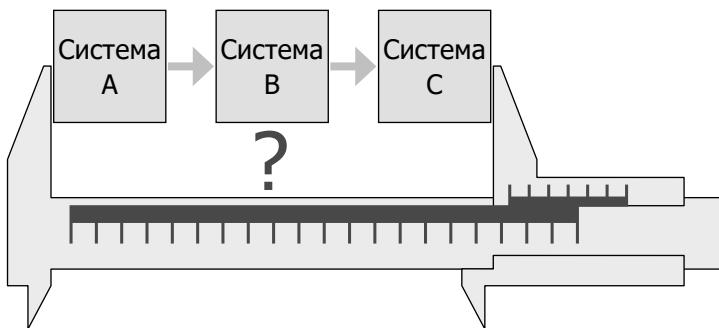


Рис. 9.2. Тестирование связанных систем

Мой штангенциркуль на рис. 9.2 не очень полезен по причинам, о которых мы только что говорили, а также потому, что чем больше и сложнее система, тем более изменчивы результаты. Мы не обеспечили достаточный контроль переменных, чтобы получить четкий повторяемый результат.

Если у вас есть набор тестов, даже автоматических, которые вы запускаете, чтобы определить, готов ли продукт к релизу, и эти тесты каждый раз выдают разные результаты, то как это можно интерпретировать?

Если мы собираемся мыслить инженерно, нужно отнестись к измерениям серьезно. Они должны быть надежными и достоверными, а значит, детерминированными. Любой тест каждой версии тестируемого продукта должен давать одинаковый результат при всяком запуске независимо от количества запусков и окружающих условий.

Итоговая ценность, которую мы получим, стоит того, чтобы приложить дополнительные усилия и добиться воспроизводимости результатов. Это влияет не только на тесты, которые мы пишем, и на то, как мы их пишем, но и, что важно, на дизайн продукта, и именно ценность дизайна подтверждает значимость инженерного подхода.

Готовая система финансовой биржи была полностью детерминирована: мы могли сохранить исходные данные продакшена и воспроизвести их спустя некоторое время, чтобы привести систему в **точно такое же состояние** в тестовой среде. Такой цели мы перед собой не ставили

изначально. Это был побочный эффект степени тестируемости и, следовательно, детерминизма, которых мы достигли.

СЛОЖНОСТЬ И ДЕТЕРМИНИЗМ

По мере роста сложности тестируемой системы точность измерений снижается. Например, если я тестирую фрагмент системы, производительность которого критична, я могу изолировать его и поместить в тестовую среду, где создать серию контролируемых тестовых запусков. Я могу исключать ранние запуски, чтобы устраниТЬ влияние оптимизации времени выполнения, и сделать достаточно прогонов, чтобы обработать полученные данные статистическими методами. При правильной реализации я добьюсь точности и воспроизводимости до микросекунд или даже до наносекунд.

Протестировать таким же образом производительность всей большой системы практически невозможно, потому что количество переменных резко возрастет. Какие еще задачи выполняются одновременно на компьютерах, на которых работает мой код? Что с сетью? Используется ли она для каких-то других процессов, пока идет тест?

Я могу следить за этими параметрами, блокируя сеть и доступ к среде тестирования производительности, но современные операционные системы весьма сложны. Что, если во время выполнения теста запустится сервисный процесс? Это наверняка исказит результаты, не так ли?

Достичь детерминизма становится труднее по мере роста сложности и размера системы.

Реальной причиной отсутствия детерминизма в компьютерных системах является параллелизм. Он может принимать различные формы. Часы, отсчитывающие системное время; ОС, реорганизующая диск во время бездействия, – все это формы параллелизма. В отсутствие же параллелизма цифровые системы детерминистичны. Одна и та же последовательность байтов и инструкций гарантирует один и тот же результат.

Один из способов повысить модульность – изолировать параллелизм, чтобы каждый модуль стал детерминированным и пригодным для тестиования. Проектируйте системы так, чтобы вход в модуль был последовательным, а его результаты – предсказуемыми. С такими системами очень приятно работать.

Кажется не совсем очевидным, но если наблюдаемое пользователями поведение системы является детерминированным в том смысле, как я это описал, оно будет более предсказуемым и проверяемым и не будет вызывать неожиданных побочных эффектов, по крайней мере в рамках тестиования.

Большинство систем построены не так, но их можно сделать такими, если применить инженерный подход.

Если бы мы использовали штангенциркуль для измерения только одного интересующего нас параметра (рис. 9.3), получившийся результат оказался бы гораздо более точным и надежным. Таким же образом можно измерить и другие параметры системы.

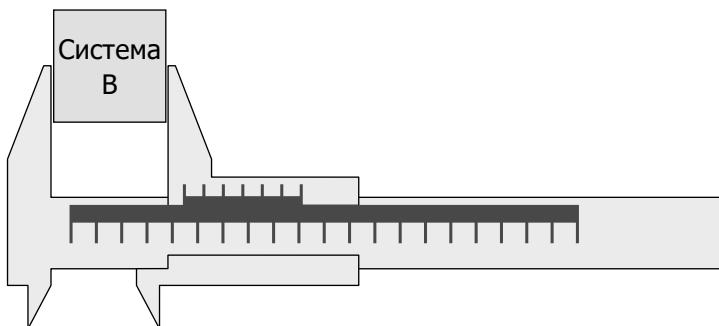


Рис. 9.3. Тестируемые модули

Итак, что необходимо, чтобы проводить измерения точно и конкретно? Мы хотим, чтобы точки измерения были стабильными, а результаты, полученные в одинаковых условиях, — одинаковыми. Мы хотим, чтобы наши оценки были детерминированными.

Кроме того, у нас нет желания создавать точки измерения с нуля при каждом изменении системы.

Для ясности я описываю сейчас стабильный модульный интерфейс части системы, которую мы собираемся тестировать. Мы формируем более крупные системы из мелких модулей входа и выхода с четко определенными интерфейсами. Такой подход к архитектуре позволяет оценивать систему в точках этих интерфейсов.

Я надеюсь, что это очевидно. Проблема в том, что немногие реальные компьютерные системы организованы именно так.

Стимулом к тому, чтобы сделать основной частью работы автоматизированное тестирование создаваемых систем, послужит тот объем дополнительных действий, который нам придется выполнять, если мы ошибемся при создании более модульных систем. Модульность работает как фрактал — на всех уровнях детализации, от целых корпоративных систем до отдельных методов, функций и классов.

Просто невозможно протестировать систему так, как указано в моей книге, если система не является модульной. Нам нужны «точки измерения». Модульность поддерживает и расширяет способность к тестированию, а ее саму, в свою очередь, стимулирует то, что при разработке проектов мы используем подход, основанный на тестировании.

При этом необязательно подразумевается коллекция крошечных независимых компонентов. Такой принцип работает и в случае больших сложных систем. Главное — понимать объем полезных измерений и добиваться того, чтобы эти измерения удавалось легко проводить и они показывали стабильные результаты.

Когда я участвовал в создании финансовой биржи, мы рассматривали всю корпоративную систему как единое целое, но установили понятные и четко определенные точки интеграции для каждого внешнего взаимодействия и симулировали внешние системы. Таким образом мы обеспечили контроль; теперь мы могли добавлять новые учетные записи и собирать данные, которые в реальных условиях отправляются в банки или расчетные центры и тому подобное.

Это позволило нам для некоторых тестов рассматривать всю систему как черный ящик, вводить в него данные, чтобы привести систему в надлежащее состояние, и собирать выходные данные для оценки реакции системы. Мы рассматривали каждую точку, в которой наша система взаимодействовала со сторонней системой, и каждую точку интеграции как точку измерения, где мы могли подключиться к тестовой инфраструктуре. Это стало возможным только потому, что вся корпоративная система с самого первого дня разрабатывалась пригодной к тестированию.

Наша система также была чрезвычайно модульной и слабо связанной. Таким образом, помимо оценки системы в целом мы имели возможность провести более подробное тестирование отдельных компонентов уровня обслуживания. Само собой, практически каждая строка кода, описывающего поведение внутри службы, была создана с использованием точных методов TDD. Мы также могли тестировать отдельные небольшие фрагменты поведения системы. Как я уже сказал, модульность и тестируемость фрактальны.

Тестируемость определила выбор архитектуры и глубоко повлияла не только на очевидные показатели качества, такие как количество обнаруженных нами ошибок, но и, что, возможно, более важно, на архитектурное разделение системы.

Исторически наша отрасль недооценивала или даже отрицала важность тестируемости, и в частности предварительной разработки тестов как инструмента для создания хорошего дизайна и получения быстрой и четкой обратной связи о его качестве.

СЛУЖБЫ И МОДУЛЬНОСТЬ

Понятие служб, или сервисов, в программировании довольно неопределенное. Например, основные языки напрямую не поддерживают идею служб, но тем не менее она довольно распространена. Разработчики спорят о том, что делает службу хорошей, а что — плохой, и проектируют свои системы в соответствии с этой концепцией.

С чисто практической точки зрения можно представить службу как код, который предоставляет некоторую услугу другому коду и скрывает подробности процесса. Это всего лишь идея сокрытия информации, и она чрезвычайно важна, если мы хотим управлять сложностью систем по мере их роста (см. главу 12). Выявление границ в дизайне систем, когда часть системы не должна знать и заботиться о том, что происходит за пределами этих границ, — очень хорошая идея. В ней истинная суть дизайна.

Службы, таким образом, позволяют реализовать маленькие системные отсеки, скрывающие детали. Это полезная идея. Таким образом, службу вполне можно рассматривать как модуль системы. Если это так, то что можно сказать о границах, точках, в которых служба или модуль соприкасаются с тем, что находится с другой стороны? Наличие границ — это то, что вообще придает смысл понятию «служба» в терминах разработки. Существует разница между тем, что известно, и тем, что явно представлено по обе стороны границ.

Одна из самых распространенных проблем в больших кодовых базах — игнорирование этой разницы. Часто код, представляющий границу, не отличим от кода по обеим сторонам от нее. Мы используем одни и те же виды вызовов методов и даже передаем одни и те же структуры данных через границу. В этих точках отсутствует проверка входных данных или сборка и выделение выходных данных. Такие кодовые базы быстро превращаются в кучу-малу, куда трудно вносить изменения.

С этим кое-что можно сделать, но это небольшой шаг, и в какой-то степени мы нащупали его случайно. Это переход к REST API.

Я имею некоторый опыт в высокопроизводительных вычислениях, поэтому довольно скептически отношусь к кодированию информации, передаваемой между службами, в виде текста, XML или HTML. Это слишком медленно! Тем не менее оно побуждает размещать точки преобразования на границах сервисов или API. Вы преобразуете входящее сообщение в более удобную форму для обработки службой и преобразуете выходные данные службы в ужасное, большое, медленное исходящее текстовое сообщение. (Извините, я дал волю эмоциям.)

Однако разработчики до сих пор делают не то, что следует. Даже в системах, построенных по этому принципу, я все еще вижу код, который передает HTML напрямую, и весь сервис взаимодействует с этим HTML, — это чудовищно!

Со швами, или границами, нужно работать очень осторожно. Они должны быть точками преобразования и проверки информации. Точка входа в службу — это своего рода защитный барьер, который ограничивает злоупотребления со стороны пользователей службы. Я говорю о разновидности модели «порты и адаптеры» на уровне отдельной службы. Этот подход должен соблюдаться как для служб, обменивающихся данными через стандартные вызовы методов или функций, так и для служб, использующих HTML, XML или любую другую форму обмена сообщениями.

Основная идея здесь — модульность! Система не является модульной, если видны внутренние процессы смежных модулей. Коммуникация между модулями (и службами) должна быть более защищенной, чем коммуникация внутри них.

РАЗВЕРТЫВАЕМОСТЬ И МОДУЛЬНОСТЬ

В книге «*Continuous delivery*» мы с Джезом Хамблом описали, как организовать работу, чтобы продукт всегда был готов к выпуску. Мы советовали (и продолжаем советовать) работать так, чтобы продукт постоянно находился в состоянии готовности к релизу. Добиться этого можно, обеспечив простое и легкое развертывание.

Завершив книгу «*Continuous delivery*», я еще больше убедился, что развертываемость и тестируемость продукта очень сильно влияют на его качество.

Одна из основных идей моей предыдущей книги — идея **пайплайна развертывания** — механизма, который принимает коммиты на одном конце

и предоставляет пригодный к релизу результат на другом. Это ключевая идея. Пайплайн развертывания, не просто узкий рабочий процесс сборки или тестирования; это механизированный путь от коммита до продакшена.

Такая интерпретация подразумевает, что все, что обеспечивает пригодность к релизу, находится в пределах пайплайна развертывания. Если в пайплайне все в порядке, больше не требуется ничего... никаких проверок интеграции, согласований или промежуточных тестов. Если пайплайн сообщает, что все хорошо, значит, все готово к работе!

Это, в свою очередь, влияет на разумный охват пайплайна развертывания. Если на выходе нужна пригодность к релизу, то требуется обеспечить и независимую развертываемость. Эффективный пайплайн развертывания всегда предоставляет «независимо развертываемый программный модуль».

Такой подход влияет и на модульность. Если выход пайплайна развертывания пригоден к развертыванию, значит, пайплайн предоставляет окончательную оценку продукта, окончательную по крайней мере до той степени, когда мы считаем безопасным и целесообразным объявить его готовность к релизу.

Если мы хотим довести эту идею до логического завершения, эффективны только две стратегии: создавать, тестировать и развертывать все части системы вместе или создавать, тестировать и развертывать эти части по отдельности. Половинчатого решения быть не может. Если мы недостаточно доверяем выходным данным пайплайна развертывания и хотим сверить их с результатами других пайплайнов, возникает проблема. Сообщения, которые отправляет нам наш пайплайн, неинформативны — так не годится, если мы хотим быть инженерами!

Теперь масштаб нашей оценки под вопросом. Когда можно считать работу законченной? Когда завершится работа нашего пайплайна или когда будут запущены все остальные пайплайны, необходимые для проверки его результатов? Если второе, то время цикла¹ наших изменений включает также время цикла всех остальных изменений, поэтому наша система оценки становится единой.

¹ Время цикла — это мера эффективности процесса разработки. Сколько времени занимает путь от идеи до полезного инструмента в руках пользователей? В непрерывной доставке оптимизация времени цикла служит инструментом, который позволяет использовать более эффективные подходы к разработке.

Самый масштабируемый подход к разработке ПО — распределенный. Уменьшите связи и зависимости между командами и их продуктами до такой степени, чтобы каждая команда могла создавать, тестировать и развертывать свой блок независимо от остальных. Именно это сделало возможным беспрецедентный рост Amazon с ее знаменитым правилом двух пицц¹.

Технически один из способов добиться этой независимости — обеспечить такую модульность системы, чтобы каждый модуль стал независимым с точки зрения сборки, тестирования и развертывания в отношении любого другого модуля. То есть мы говорим о микросервисах. Они настолько модульны, что нам нет необходимости тестировать их с другими сервисами перед релизом. Если вы тестируете свои блоки вместе, это не микросервисы, ибо по определению микросервисы могут быть развернуты независимо.

Развертываемость повышает значение модульности. Как мы видели, она определяет правильный охват пайплайна развертывания. Если мы стремимся к высокому качеству, основанному на быстрой и эффективной обратной связи, выбор действительно эффективных решений невелик.

Мы можем создавать, тестировать и развертывать компоненты совместно и избавиться от проблемы управления зависимостями (все находится в одном репозитории), но тогда нам придется обеспечить достаточно быструю обратную связь, чтобы разработчики могли сделать свою работу хорошо. А такая связь, которая лежит в основе любого хорошо организованного процесса, требует больших инвестиций.

В качестве альтернативы можно сделать каждый модуль независимым от других и создавать, тестировать и развертывать каждый из них по отдельности, без необходимости совместного тестирования.

Это означает, что охват сборок, тестов и развертываний невелик. Каждая операция становится проще, тем самым достигается быстрый качественный результат.

Однако за это иногда приходится очень дорого расплачиваться более сложной и распределенной архитектурой системы, которая означает очень высокую модульность.

¹ Amazon провела реорганизацию после выступления ее генерального директора Джека Безоса, который заявил, что «...в команде должно быть столько людей, сколько можно накормить двумя большими пиццами».

Нам приходится прилагать недюжинные усилия. Мы должны владеть методами проектирования протоколов, чтобы взаимодействия между модулями и протокол обмена информацией оставались стабильными и их изменение не вызывало изменений в других модулях. И здесь мы обращаемся к управлению версиями среды выполнения в API и так далее.

Едва ли не каждому хочется найти золотую середину между этими полюсами, но на самом деле ее нет. Золотая середина — это иллюзия, она часто медленнее и сложнее, чем единый подход, которого все так старательно избегают. Более организационно распределенный подход, то есть микросервисы, — лучший из известных способов масштабирования разработки, но он непрост и требует затрат.

МОДУЛЬНОСТЬ В РАЗНЫХ МАСШТАБАХ

Модульность важна в любом масштабе. Разворачиваемость — полезный инструмент, когда речь идет о модулях уровня системы, но его одного недостаточно для создания отличного кода. Современная разработка через скрочу увлечена службами.

Службы я рассматриваю как основной и очень полезный инструмент при проектировании систем, по крайней мере последние три десятилетия. Однако если этим модульность дизайна и ограничивается, итоговый продукт получается сложным в работе и некачественным.

Если, как я утверждаю, модульность помогает управлять сложностью, нам нужно реализовать ее в читаемом коде. Каждый класс, метод или функция должны быть простыми и удобочитаемыми и, при необходимости, состоять из более мелких, независимо понятных субмодулей.

Опять же, такой мелкомодульный код легче писать, применяя TDD. Чтобы тестировать код при таком разрешении, используют внедрение зависимостей, расширяющее площадь его покрытия. А оно значительно влияет на модульность системы.

В меньших масштабах внедрение зависимостей — наиболее эффективный инструмент построения систем, состоящих из множества небольших фрагментов. Зависимости — это штангенциркули, точки измерения, которые мы можем внедрить в систему, чтобы сделать ее более проверяемой. Кроме того, чтобы код был проверяемым, дизайн должен быть действительно модульным — тогда код легче читать.

Некоторые критикуют такие подходы к работе. Обычно приводят аргументы, что код с большей площадью покрытия труднее понять и следить за потоком управления в системе сложнее. Но эти критики упускают суть. Проблема в том, что если необходимо раскрыть некую область для тестирования кода, то это область покрытия кода. Насколько затрудняется его понимание из-за плохого дизайна интерфейса и недостаточного количества тестов? Вопрос в том, что понимать под хорошим дизайном. Я предлагаю определять высокое качество кода на основе того, насколько дизайн ориентирован на управление сложностью.

Тестирование, проведенное качественно, выявляет важные и значимые свойства кода, дизайна и задачи, которые сложно выявить другими способами. Таким образом, это один из самых важных из доступных инструментов для создания более качественных, более модульных систем и кода.

МОДУЛЬНОСТЬ В СИСТЕМАХ, СОЗДАВАЕМЫХ ЧЕЛОВЕКОМ

Подробно о влиянии инженерного мышления мы поговорим в главе 15, но сейчас я хочу отметить особую важность модульности в этом мышлении. Большую часть своей карьеры я посвятил работе с крупными вычислительными системами. В этой среде основной вопрос, который звучит изо дня в день: «Как мы масштабируемся?» Иногда, редко, речь идет о продуктах разработки, но в основном, когда люди в крупных организациях задают этот вопрос, они на самом деле хотят знать, сколько нужно еще людей, чтобы выпускать продукт быстрее.

В действительности же любая компьютерная система серьезно ограничена в этом отношении. Фред Брукс произнес знаменитую фразу:

Девять женщин за месяц ребенка не родят¹.

Но есть и другие варианты, например, девять женщин рожают девять детей за девять месяцев, что в среднем составит одного ребенка в месяц. Здесь наша с Фредом аналогия не работает!

¹ Фраза из известной и по-прежнему актуальной книги Фреда Брукса, написанной в 1970-е годы, «Мифический человеко-месяц». СПб, Издательство «Питер».

Если вернуться от младенцев к разработке ПО, мы найдем источник проблемы в связанности. Пока части действительно независимы друг от друга и не связаны, мы можем распараллелить все, что захотим. С возникновением связей появляются и ограничения на степень параллелизма. Стоимость интеграции становится убийственной!

Как объединять процессы из отдельных рабочих потоков? Если вы такой же зануда, как и я, здесь для вас прозвенит звоночек: проблема затрагивает фундаментальные понятия информации и параллелизма. Стоимость объединения независимых потоков информации может быть чрезвычайно высока, если эти потоки пересекаются. Лучший способ распараллелить вещи — сделать это так, чтобы не возникла необходимость объединять их вновь (девять младенцев). По сути, это микросервисный подход. Микросервисы — это игра с масштабируемостью; на самом деле у них нет другого реального преимущества, но это единственное преимущество огромно, если масштабируемость представляет для вас сложность!

Мы знаем, что команда не начнет работать быстрее, если просто увеличить ее численность. В прекрасном исследовании метаданных более 4000 программных проектов сравнивали относительную производительность (время на создание 100 000 строк кода) команд, состоящих из 5 человек или менее и из 20 или более. За 9 месяцев командам из 5 человек потребовалось на завершение проекта всего на неделю больше, чем командам из 20 человек. Таким образом, продуктивность на человека в небольших командах почти в 4 раза выше, чем в более крупных¹.

Если, чтобы работать качественно и эффективно, нужны небольшие команды, то требуется и способ значительно ограничить связанность между этими командами. Это как минимум проблема настолько же организационная, насколько и техническая. Нам необходимы **модульные организации**, а также **модульные программные продукты**.

Если мы хотим, чтобы организации можно было масштабировать, а действия команд и систем требовали минимального координации, нам нужно отделить их друг от друга. Действительно высокопроизводительные, масштабируемые компании прилагают значительные усилия для поддержания модульной организации.

¹ Исследование «Количественное управление разработкой программного обеспечения» (Quantitative Software Management, QSM) также показало, что код, написанный более крупными командами, содержит в 5 раз больше ошибок.

ИТОГИ

Модульность — залог достижения успеха в ситуации, когда мы не в полной мере представляем, как наш продукт должен работать. Безусловно, можно создать простое решение, не прибегая к модульности. Тем не менее, если мы не изолируем части программного продукта, очень скоро мы столкнемся со сложностями, а в некоторых случаях даже с невозможностью добавления новых опций и развития продукта. Модульность — главный инструмент защиты от сложности.

Модульность как дизайнерская идея фрактальна. И эта концепция более многогранна, чем просто поддержка необходимого синтаксиса языка программирования для модулей любой формы. По сути, фрактальность предполагает способность менять код и параметры в одном месте так, чтобы эти изменения не оказывали влияния на другие фрагменты системы.

Это заставляет задуматься о других сторонах этой проблемы, поэтому модульность тесно связана с другими категориями, которые необходимо учитывать при управлении сложностью систем: абстракцией, разделением ответственности, связанностью и связностью.

Ещё больше книг в нашем телеграм канале:
<https://t.me/bookofgeek>

ГЛАВА 10

СВЯЗНОСТЬ

Связность (cohesion) в computer science определяется как «степень, в которой элементы внутри модуля связаны друг с другом».

МОДУЛЬНОСТЬ И СВЯЗНОСТЬ: ОСНОВЫ ДИЗАЙНА

Описывая хороший программный дизайн, я всегда использую слова Кента Бека:

Отодвиньте друг от друга все, что не связано, и придвиньте ближе друг к другу все, что связано.

В этой простой, немного шутливой фразе есть доля правды. Хороший программный дизайн на самом деле зависит от того, как мы организуем код в системах. Все рекомендуемые мной принципы, помогающие управлять сложностью, на самом деле касаются разделения наших систем. Нам нужно создавать системы из более мелких, понятных и легко тестируемых дискретных частей. Чтобы достичь этого, безусловно, требуются методы, которые позволяют отодвигать несвязанные вещи дальше друг от друга, но не менее важна необходимость сближать связанные вещи. Здесь в игру вступает связность.

Связность — одно из самых зыбких понятий. Я могу сделать что-то простое, например использовать модульный синтаксис, и в результате заявить, что мой код модульный. Но он таким не станет; простое добавление набора несвязанных вещей в файл не делает код модульным ни в каком смысле, кроме самого банального.

Когда я говорю о модульности, я имею в виду, что в системе есть компоненты, которые скрывают информацию от других компонентов (модулей). Если код внутри самого модуля несвязный, то это не работает.

Проблема в том, что такое утверждение зачастую чрезмерно упрощают. Вероятно, это тот случай, когда мастерство, навыки и опыт программиста-практика имеют значение. Баланс между действительно модульными системами и связностью часто сбивает людей с толку.

БАЗОВОЕ СНИЖЕНИЕ СВЯЗНОСТИ

Как часто вы видели код, который извлекает некоторые данные, разбирает их, а затем сохраняет в другом месте? Наверняка шаг «сохранить» связан с шагом «изменить»? Разве это не хорошая связность? Эти шаги нужны все вместе, не так ли?

Не совсем. Давайте рассмотрим пример. Предупреждаю: здесь сложно выделить несколько идей. Приведенный код проиллюстрирует понемногу каждую из идей этого раздела, поэтому я надеюсь, что вы сумеете сосредоточиться на связности и понимающие улыбнетесь, когда я также коснусь разделения ответственности, модульности и так далее.

В листинге 10.1 показан пример довольно грубого кода. Тем не менее он подходит для моей цели — изучить конкретный практический пример. Этот код читает небольшой файл, содержащий список слов, сортирует слова в алфавитном порядке, а затем записывает новый файл с итоговым упорядоченным списком — загружайте, обрабатывайте и сохраняйте!

Это довольно типичный шаблон для множества различных задач: прочитать какие-то данные, обработать их, а затем сохранить результат в другом месте.

Листинг 10.1. Очень плохой код с наивной связностью

```
public class ReallyBadCohesion
{
    public boolean loadProcessAndStore() throws IOException
    {
        String[] words;
        List<String> sorted;

        try (FileReader reader =
              new FileReader("./resources/words.txt"))
        {
```

```
char[] chars = new char[1024];
reader.read(chars);
words = new String(chars).split(" |\0");
}
sorted = Arrays.asList(words);
sorted.sort(null);

try (FileWriter writer =
        new FileWriter("./resources/test/sorted.txt"))
{
    for (String word : sorted)
    {
        writer.write(word);
        writer.write("\n");
    }
    return true;
}
}
```

Я считаю, что этот код очень плох, и мне пришлось заставить себя написать его таким. Код просто кричит о плохом разделении задач, слабой модульности, жесткой связанности и почти нулевой абстракции, но как насчет связности?

Вся работа здесь реализуется в одной функции. И очень часто выпущенный код выглядит так же, только намного длиннее и сложнее, так что в реальности все еще хуже!

Наивное представление о связности таково: все помещается рядом и поэтому легко заметно. Но если на мгновение отвлечься от других методов управления сложностью, так ли легко прочесть этот код? Сколько времени вам понадобится, чтобы понять, что он делает? Долго ли вам придется гадать, если я не дам подсказку в названии метода?

Теперь взгляните на листинг 10.2, где дела обстоят чуть лучше.

Листинг 10.2. Плохой код, связность немного лучше

```
public class BadCohesion
{
    public boolean loadProcessAndStore() throws IOException
    {
        String[] words = readWords();
        List<String> sorted = sortWords(words);
        return storeWords(sorted);
    }
}
```

```

private String[] readWords() throws IOException
{
    try (FileReader reader =
          new FileReader("./resources/words.txt"))
    {
        char[] chars = new char[1024];
        reader.read(chars);
        return new String(chars).split(" |\0");
    }
}

private List<String> sortWords(String[] words)
{
    List<String> sorted = Arrays.asList(words);
    sorted.sort(null);
    return sorted;
}

private boolean storeWords(List<String> sorted) throws IOException
{
    try (FileWriter writer =
          new FileWriter("./resources/test/sorted.txt"))
    {
        for (String word : sorted)
        {
            writer.write(word);
            writer.write("\n");
        }
        return true;
    }
}
}

```

Листинг 10.2 по-прежнему не очень хорош, но он более связный; взаимосвязанные части кода более четко прописаны и располагаются ближе друг к другу. Проще говоря, все, что вам нужно знать о `readWords`, названо и содержится в одном методе. Общий поток метода `loadProcessAndStore` теперь хорошо просматривается, даже если бы я выбрал менее описательное имя. Информация в этой версии более связна, чем в листинге 10.1. Теперь стало значительно понятнее, какие части кода более тесно связаны друг с другом, хотя код имеет тот же функционал, что и раньше. Все это делает эту версию значительно более удобной для чтения, и, как следствие, ее проще изменять.

Обратите внимание, что в листинге 10.2 больше строк кода. Этот пример написан на Java, довольно многословном языке, и шаблонный код на нем весьма дорог, к тому же придется добавить небольшие накладные расходы на улучшение читаемости. Это не обязательно плохо!

Все программисты стремятся уменьшить объем печатаемого текста. Мы ценим ясность и краткость. Большое значение имеет способность выражаться просто, но простоту не измерить количеством набранных символов. Выражение `I Can Write A Sentence Omitting Spaces` короче, но менее читаемо!

Оптимизировать код, чтобы уменьшить количество набираемого текста, — ошибка. Это оптимизация не того, что нужно. Код — инструмент коммуникации; мы должны использовать его для передачи информации. Конечно, он также должен быть машиночитаемым и исполняемым, но на самом деле это не основная цель. Если бы она была такой, мы бы до сих пор программировали системы, переключая тумблеры на передней панели компьютеров или с помощью машинного кода.

Основная цель кода — донести идеи до людей. Мы пишем код, чтобы максимально ясно и просто выражать мысли, — по крайней мере, так это должно работать. Никогда нельзя жертвовать ясностью в угоду краткости. Сделать код читаемым — это, на мой взгляд, и профессиональная обязанность, и один из самых важных руководящих принципов управления сложностью. Поэтому я предпочитаю оптимизировать мышление, а не сокращать текст.

Вернемся к коду: этот второй пример явно более легко читается. Стало понятнее, что он делает. Но код все еще ужасен — он не модульный, почти не предусматривает разделения функций, негибкий, с жестко закодированными строками для имен файлов, и его нельзя проверить, кроме как запустить целиком и работать с файловой системой. Но мы улучшили связность. Каждый фрагмент кода теперь выполняет одну часть задачи. Фрагменты имеют доступ только к тому, что им необходимо для ее выполнения. Мы вернемся к этому примеру в следующих главах, чтобы посмотреть, как его улучшить.

КОНТЕКСТ ИМЕЕТ ЗНАЧЕНИЕ

Я спросил друга, чьим кодом я восхищаюсь, как лучше показать важность связности, и он порекомендовал видео из «Улицы Сезам», которое можно найти на YouTube¹: «Один из этих предметов не похож на другие».

¹ Песенка «Один из этих предметов не похож на другие» (One of these things is not like the other) из «Улицы Сезам»: <https://youtu.be/rsRjQDrDnY8>.

В шутливой форме выражается важная мысль. Связность в большей степени, чем другие инструменты управления сложностью, зависит от контекста. В зависимости от контекста «все эти предметы могут быть не похожи друг на друга».

Мы должны сделать выбор, и этот выбор тесно связан с другими инструментами. Невозможно однозначно отделить связность от модульности или разделения ответственности, потому что эти методы помогают определить, что означает связность в контексте разработки.

Одним из эффективных инструментов для принятия такого рода решений является предметно-ориентированное проектирование (domain-driven design, DDD)¹. В предметной области проще определить направление, которое с большей вероятностью принесет прибыль в долгосрочной перспективе. Это касается и мышления, и разработки.

ПРЕДМЕТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

Предметно-ориентированное проектирование – это подход к проектированию, при котором мы стремимся воссоздать предметную область в коде. Дизайн системы направлен на точное моделирование проблемы.

Этот подход подразумевает ряд важных, ценных идей.

Он позволяет снизить вероятность непонимания. Мы стремимся создать универсальный язык как согласованный и точный способ описания идей в предметной области, когда слова располагаются последовательно и имеют согласованные значения. Затем мы применяем этот язык для описания дизайна систем.

Так что если я говорю о своем продукте и упоминаю, что «лимитный ордер соответствует» («Limit-order matched»), то это имеет смысл с точки зрения кода, где четко представлены понятия «лимитные ордера» и «соответствие», названные `LimitOrder` и `Match`. Это те же слова, которые мы используем, описывая сценарий людям, далеким от технической сферы.

Такой универсальный язык эффективно разрабатывается и совершенствуется благодаря сбору требований и использованию высокоуровневых тестовых сценариев, играющих роль исполняемых спецификаций поведения системы.

В рамках DDD также была введена концепция «ограниченный контекст». Это часть системы, которая содержит общеупотребимые понятия.

¹ «Предметно-ориентированное проектирование» — заголовок знаменитой книги Эрика Эванса и название подхода к разработке систем ПО.

Например, в системе управления заказами понятие «заказ», вероятно, отличается от аналогичного понятия в биллинговой системе, поскольку используется в разных ограниченных контекстах.

Это чрезвычайно полезная концепция, помогающая правильно определять модули или подсистемы при проектировании. Большое преимущество ограниченных контекстов в том, что они более слабо связаны в реальной проблемной области, поэтому с их помощью можно создавать более слабо связанные системы.

Такие концепции, как универсальный язык и ограниченный контекст, применяются, чтобы управлять дизайном систем. Используя их, вы сможете создавать более совершенные системы, а также четко представить основную, существенную сложность разрабатываемой системы и отличить ее от случайной сложности, которая часто мешает понять, что на самом деле делает код.

Если спроектировать систему так, чтобы она симулировала предметную область, как мы ее понимаем, то небольшое изменение в предметной области станет и небольшим изменением в коде. Это полезное свойство.

Предметно-ориентированное проектирование – мощный инструмент для создания более качественных проектов, который включает принципы, организующие наши усилия в проектировании и побуждающие улучшать модульность, связность и разделение функций в коде. В то же время он позволяет создавать крупные слабо связанные модули кода.

Еще один важный инструмент, который помогает нам создавать более совершенные системы, – разделение ответственности. О нем мы поговорим более подробно в следующей главе. А сейчас я лишь скажу, пожалуй, что разделение ответственности – это мой стиль программирования: «Один класс, одна задача; один метод/функция, одна задача».

Мне очень не нравятся оба примера кода, которые я привожу в этой главе, и мне неловко показывать их вам, потому что мой инстинкт проектировщика буквально вопит, что разделение ответственности в обоих случаях ужасно. Листинг 10.2 лучше: по крайней мере, каждый метод теперь делает что-то одно, но класс по-прежнему ужасен. Если вы еще этого не видите, я расскажу, почему это важно, в следующей главе.

Наконец, в моем наборе инструментов есть тестируемость. Я начал писать эти плохие примеры кода, как и любой код: с написания теста. Однако мне пришлось остановиться почти сразу же, потому что я не мог, используя TDD, написать такой плохой код! Мне пришлось начать работу заново,

и, признаюсь, я чувствовал, что перенесся в прошлое. Я написал тесты для примеров, чтобы проверить, работают ли они так, как я ожидал, но этот код нельзя назвать тестируемым.

Тестируемость означает модульность, разделение ответственности и все, что мы ценим в высококлассном коде. Это, в свою очередь, помогает приблизиться к контексту и абстракции, которые нам нравятся в дизайне, и понять, где можно сделать код более связным.

Заметьте: нет никаких гарантий, что это удастся, — и это основная мысль книги. Не существует простых, шаблонных ответов. В книге я показываю вам интеллектуальные инструменты, которые помогают структурировать мышление, когда не получается найти ответы.

Техники, описанные в этой книге, не дают их; ответы вам придется искать самостоятельно. Эти техники предоставляют набор идей и приемов, чтобы двигаться безопасно, если вы еще не знаете ответов, а вы их не знаете, независимо от сложности любой реально создаваемой системы. Мы никогда не знаем ответов, пока не закончим работу!

Эта стратегия выглядит скорее оборонительной, и она такая и есть, но ее цель в том, чтобы сохранить свободу выбора. Это одно из значительных преимуществ работы над управлением сложностью. По мере получения новых знаний мы можем постоянно улучшать код, используя полученные знания. Мне кажется, «инкрементная» — более точный эпитет, чем «оборонительная».

Мы добиваемся прогресса постепенно, проводя серию экспериментов, и используем методы **управления сложностью**, чтобы защитить код от серьезных ошибок.

Так работают наука и инженерия. Мы контролируем переменные, делаем небольшой шаг и оцениваем текущее положение. Если оценка показывает, что мы сделали неверный шаг, мы возвращаемся назад и решаем, что делать дальше. Если все нормально, мы контролируем переменные, делаем еще один небольшой шаг, и так далее.

Можно представить разработку как своего рода эволюционный процесс. Программисты добиваются цели в результате направленной эволюции в сфере обучения и разработки.

ВЫСОКОПРОИЗВОДИТЕЛЬНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Плохой код, такой как в листинге 10.1, часто оправдывают тем, что для высокой производительности системы требуется писать более сложный код. В последние годы я занимаюсь разработкой очень высокопроизводительных систем — и уверяю вас, что это не так. Такие системы требуют простого, продуманного кода.

Задумайтесь на мгновение, что означает *высокая производительность* с точки зрения разработки. Для ее достижения нам нужно выполнять максимальный объем работы при минимальном количестве инструкций.

Чем сложнее код, тем больше вероятность того, что разработанные маршруты не оптимальны, потому что самый простой возможный маршрут через код скрыт сложностью самого кода. Многих программистов это удивляет, но чтобы код работал быстро, он должен быть простым и легко понятным.

Правильность этих рассуждений очевидна, если посмотреть на систему шире.

Вернемся к нашему примеру. Я слышал, как программисты утверждали, что код в листинге 10.1 будет быстрее, чем в листинге 10.2, из-за накладных расходов на вызовы методов, которые появились в листинге 10.2. Но для современных языков это нонсенс. Большинство современных компиляторов просмотрят код в листинге 10.2 и встроят методы. Кроме того, современные оптимизирующие компиляторы способны на большее. Они фантастически оптимизируют код, чтобы он работал эффективно на современном оборудовании. Компиляторы полезны, когда код прост и предсказуем, поэтому чем он сложнее, тем меньше пользы от оптимизатора. Большинство оптимизаторов просто прекращают работу, как только цикломатическая сложность¹ блока кода превышает определенный порог.

Я оценил производительность обеих версий кода. Эти тесты оказались так себе, потому что код плохой. Мы недостаточно контролируем переменные, чтобы действительно понимать, что происходит, но очевидно, что на этом уровне бенчмарка разницы практически не было.

¹ В разработке: метрика, используемая для оценки сложности компьютерной программы.

Различия слишком малы, чтобы их удалось дифференцировать от остальных событий. В одном прогоне версия `BadCohesion` оказалась лучшей, в другом — `ReallyBadCohesion`. В серии тестовых прогонов для каждой из 50 000 итераций метода `loadProcessStore` общая разница составляла не более 300 миллисекунд, так что в среднем это дало примерно 6 наносекунд на вызов, чаще — в пользу версии с дополнительными вызовами методов.

Это плохой тест, потому что то, что нас интересует, — стоимость вызовов методов — ничтожно мала по сравнению со стоимостью операций ввода-вывода. Тестируемость — в данном случае тестируемость производительности — снова поможет добиться лучшего результата. Мы обсудим это более подробно в следующей главе.

Столько всего скрыто от глаз, что даже экспертам трудно предсказать исход. Каков ответ? Если вас действительно интересует производительность кода, не гадайте, что работает быстро, а что медленно, — измеряйте!

ОТСЫЛКА К СВЯЗАННОСТИ

Если мы хотим сохранить свободу исследования и иногда совершать ошибки, нам нужно побеспокоиться об издержках **связанности**, или **сцепления** (*coupling*).

Связанность: для двух строк кода, А и В, связанность означает, что В должна изменить поведение только потому, что изменилась А, — строки сцеплены.

Связность: строки являются связными, когда изменение А позволяет измениться В, так что обе добавляют новую ценность¹.

Связанность — слишком общий термин. Различные виды связанности мы более подробно рассмотрим в главе 13.

Несвязанная система — это забавно. Если мы хотим, чтобы две части системы взаимодействовали, они должны быть в какой-то степени сцеплены. Так что, как и связность, связанность — это вопрос степени, а не какой-то абсолютной меры. Однако цена несоответствующих уровней связанности чрезвычайно высока, поэтому при проектировании важно учитывать ее влияние.

¹ Связанность и связность описаны в известной электронной энциклопедии C2: <https://wiki.c2.com/?CouplingAndCohesion>.

Связанность в некотором смысле является платой за связность. Области системы, которые обладают связностью, вероятно, будут иметь и большее сцепление.

ОБЕСПЕЧЕНИЕ ВЫСОКОЙ СВЯЗНОСТИ С ПОМОЩЬЮ TDD

Использование автоматических тестов, в частности TDD, для управления дизайном дает много преимуществ. Стремление создать проверяемый дизайн и достаточно абстрагированные поведенческие тесты помогут сделать код связным.

Прежде чем создавать код, описывающий поведение, которого мы ожидаем от системы, мы создаем тестовый сценарий. Это позволяет сосредоточиться на разработке внешнего API/интерфейса для кода. Теперь напишем реализацию, которая будет соответствовать созданной нами небольшой исполняемой спецификации. Если кода слишком много, больше, чем нужно для соответствия спецификации, процесс нарушится и связность реализации снизится. Если кода слишком мало, не удастся реализовать желаемое поведение. Методология TDD стимулирует добиваться связности.

Как всегда, никаких гарантий. Это не механический процесс, и результат по-прежнему зависит от опыта и навыков программиста, но этот подход позволяет добиться прогресса, о котором раньше и помыслить было невозможно, и развивает навыки и опыт.

КАК ДОБИТЬСЯ СВЯЗНОСТИ

Ключевой метрикой связности служит степень, или стоимость, изменений. Если вам приходится изменять кодовую базу во многих местах, это не очень связная система. Связность — мера функционального соответствия. Это измерение соответствия цели. И очень зыбкое понятие!

Давайте рассмотрим простой пример.

Класс с двумя методами, каждый из которых ассоциирован с переменной-членом (листинг 10.3), — пример плохой связности, потому что переменные здесь никак не соотносятся. Они принадлежат разным методам, но хранятся вместе на уровне класса, даже если не соотносятся между собой.

Листинг 10.3. Плохая связность

```
class PoorCohesion:
    def __init__(self):
        self.a = 0
        self.b = 0

    def process_a(x):
        a = a + x

    def process_b(x):
        b = b * x
```

В листинге 10.4 представлено гораздо более красивое и связное решение. Обратите внимание, что эта версия не только более связная, но и более модульная, с лучшим разделением функций. Очевидно, что эти концепции взаимосвязаны.

Листинг 10.4. Улучшенная связность

```
class BetterCohesionA:
    def __init__(self):
        self.a = 0

    def process_a(x):
        a = a + x

class BetterCohesionB:
    def __init__(self):
        self.b = 0

    def process_b(x):
        b = b * x
```

В сочетании с остальными принципами управления сложностью стремление создать тестируемый дизайн помогает улучшить связность решений. Хороший пример — внимание к разделению ответственности, особенно когда речь идет об отделении случайной сложности¹ от необходимой².

В листинге 10.5 показаны три простых примера улучшения связности кода за счет сознательного разделения необходимой и случайной сложности.

¹ Случайная сложность системы — это сложность, обусловленная тем, что мы работаем на компьютере. Это побочный эффект решения реальной задачи, которая нас интересует, например сохранения информации, работы с параллелизмом или сложными API и т. д.

² Необходимая сложность системы — это сложность, присущая решению задачи, например расчету процентной ставки или добавлению товара в корзину.

В каждом примере мы добавляем товар в корзину, сохраняем его в базе данных и вычисляем стоимость корзины.

Листинг 10.5. Три примера связности

```
def add_to_cart1(self, item):
    self.cart.add(item)

    conn = sqlite3.connect('my_db.sqlite')
    cur = conn.cursor()
    cur.execute('INSERT INTO cart (name, price)
values (item.name, item.price)')
    conn.commit()
    conn.close()

    return self.calculate_cart_total();

def add_to_cart2(self, item):
    self.cart.add(item)
    self.store.store_item(item)
    return self.calculate_cart_total();

def add_to_cart3(self, item, listener):
    self.cart.add(item)
    listener.on_item_added(self, item)
```

Первая функция — очевидно несвязный код. В нем смешано множество понятий и переменных — необходимой и случайной сложности. Я бы сказал, что это очень плохой код, даже в таком масштабе. Я бы не стал писать такой код, потому что из него трудно понять, что происходит, хотя сценарий чрезвычайно прост.

Второй пример немного лучше. Связность в нем выше. Понятия в этой функции соотносятся друг с другом и представляют более последовательный уровень абстракции, поскольку они в основном связаны с необходимой сложностью задачи. Инструкция «сохранить» выглядит спорно, но по крайней мере этим мы скрыли случайную сложность.

Последний пример интересен. Я бы сказал, что, он, безусловно, связный. Чтобы выполнить полезную работу, следует добавить товар в корзину и сообщить об этом другим потенциально заинтересованным сторонам. Мы полностью разделили задачи хранения и подсчета общей стоимости товаров в корзине. Программа может отреагировать в ответ на уведомление о добавлении, а может и не отреагировать, если соответствующие фрагменты кода не зафиксированы к событию «элемент добавлен».

Код можно считать либо более связным, если в нем заключена вся необходимая сложность задачи, а другие варианты поведения являются побочными эффектами, либо менее связным, если считать события «store» и «total» частями задачи. В конечном счете выбор дизайна зависит от контекста задачи, которую вы решаете.

ЦЕНА ПЛОХОЙ СВЯЗНОСТИ

Связность, пожалуй, наименее поддается количественной оценке как атрибут «инструментов управления сложностью», но она важна. Проблема в том, что когда связность плохая, код и системы становятся менее гибкими, их сложнее тестировать и с ними сложнее работать.

В простом примере в листинге 10.5 влияние связного кода очевидно. Если в коде пересекается функционал, ему не хватает ясности и удобочитаемости, как показано в `add_to_cart1`. Если функционал широкий, труднее увидеть, что происходит, как в `add_to_cart3`. Располагая взаимосвязанные идеи рядом, мы добиваемся максимальной удобочитаемости, как в `add_to_cart2`.

На самом деле у способа проектирования `add_to_cart3` есть некоторые преимущества, и такой код, безусловно, удобнее для работы, чем версия 1.

Я считаю, что это оптимальный вариант связности. Если вы смешаете слишком много концепций, вы потеряете связность на уровне деталей. В примере 1 можно утверждать, что вся работа делается внутри одного метода, но это только наивная связность.

На самом деле добавление товара в корзину (основной бизнес-функционал) смешано с другими функционалами, что делает общую картину неясной. Даже в этом простом примере непонятно, как работает код, пока вы не углубитесь в него.

Другой альтернативе, `add_to_cart3`, хотя она и более гибкая, все еще не хватает ясности. В этом экстремальном случае функционал может быть настолько рассредоточен, что вы не сможете понять общую картину, не прочитав множество строк кода и не разобравшись в нем. Это, наверное, и неплохо, но я считаю, что ограничение в ясности — стоимость слабой связанности и некоторых других преимуществ.

Оба этих недостатка очень часто встречаются в уже готовых системах. На самом деле настолько часто, что в больших сложных системах становятся даже своего рода нормой.

Это ошибка дизайна, и она дорого обходится. Вы с ней наверняка хорошо знакомы, если когда-нибудь работали с «унаследованным кодом»¹.

Существует простой субъективный способ определить плохую связность. Если вы читаете код и думаете: «Я не знаю, что он кодирует», — вероятно, причиной тому плохая связность.

СВЯЗНОСТЬ В ЧЕЛОВЕЧЕСКИХ СИСТЕМАХ

Как и в других случаях, проблемы связности не ограничиваются только кодом, который мы пишем, и системами, которые мы создаем. Связность работает на уровне информации, так что она также важна для построения эффективной структуры в организациях, где мы работаем. Самый очевидный пример — организация работы команд. Выводы из отчета State of DevOps свидетельствуют, что способность принимать собственные решения — без их утверждения у кого-либо вне команды — один из главных признаков высокой производительности, метриками которой служат пропускная способность и стабильность. То есть информация и навыки команды обладают связностью в том смысле, что внутри команды есть все, что требуется, чтобы принимать решения и добиваться успеха.

ИТОГИ

Связность, вероятно, самый отвлеченный из принципов управления сложностью. Разработчики могут утверждать (и иногда утверждают), что связность — это когда весь код находится в одном месте, одном файле и даже одной функции, но это слишком упрощенное представление.

¹ Унаследованный код, или унаследованные системы, — это системы, которые эксплуатируются уже долгое время. Вероятно, они по-прежнему приносят пользу организациям, но часто такой код становится запутанным и непонятным. Майкл Фезерс (Michael Feathers) определяет унаследованную систему как «систему без тестов».

Код, который таким образом случайно объединяет несколько концепций, не является связанным; в нем просто нет структуры. Это плохо, поскольку не позволяет увидеть, что делает код и как его безопасно изменить.

Связность заключается в объединении соотносящихся концепций, которые изменяются в коде совместно. Если они оказались вместе случайно, на самом деле связности между ними нет.

Связность — это детектор модульности и в целом имеет смысл, если рассматривать ее в сочетании с модульностью. Одним из наиболее эффективных инструментов, помогающих найти рабочий баланс между связностью и модульностью, является разделение ответственности.

ГЛАВА 11

РАЗДЕЛЕНИЕ ОТВЕТСТВЕННОСТИ

Разделение ответственности (separation of concerns) определяется как «принцип проектирования, представляющий собой процесс разделения компьютерной программы на отдельные блоки таким образом, чтобы каждый блок отвечал за определенный функционал».

Разделение ответственности — самый мощный инструмент дизайна, который я использую в работе. Я применяю его везде.

Простое бытовое описание разделения ответственности звучит так: «Один класс — одна задача. Один метод — одна задача». Это отличный слоган, который не стоит игнорировать функциональным программистам.

Разделение ответственности подразумевает чистоту и точность кода и систем. Это одна из основных техник повышения модульности, связности и абстракции системы и, как результат, ослабления связанности до эффективного минимума.

Разделение ответственности также работает на всех уровнях детализации. Это полезный принцип как в масштабе целых систем, так и на уровне функций.

Разделение ответственности на самом деле не то же самое, что связность и модульность. Последние два свойства являются свойствами кода, и когда мы говорим о хорошем разделении ответственности в коде, на самом деле мы имеем в виду «то, что не связано, находится далеко друг от друга, а то, что связано, — близко». Разделение ответственности — это особый взгляд на модульность и связность.

Разделение ответственности — это прежде всего способ уменьшить сцепление и улучшить связность и модульность кода и систем.

Тем не менее такая трактовка как бы преуменьшает важность разделения ответственности в проектировании. А это для меня — основной показатель хорошего дизайна. Благодаря ей код и архитектура систем, которые я создаю, остаются, помимо прочего, чистыми, точными, компонуемыми, гибкими, эффективными, масштабируемыми и открытыми для изменений.

СМЕНА БАЗЫ ДАННЫХ

Работая над финансовой биржей, мы внедряли принципы инженерии, о которых я рассказываю здесь. Собственно, именно этот опыт и побудил меня взяться за перо. Биржа получилась фантастической — лучшей кодовой базой больших систем, над которой я когда-либо работал и которая мне встречалась.

Мы строго разделяли ответственность, начиная с уровня отдельных функций и вплоть до архитектуры всей системы. Мы писали бизнес-логику, которая вообще не зависела от окружения, была полностью тестируемой, не совершала удаленных вызовов, не записывала данных, не знала адресов взаимодействующих объектов и не требовала дополнительного внимания к безопасности, масштабируемости или отказоустойчивости.

Службы работали таким образом, потому что за такое поведение отвечали другие части системы. Поведение отдельных служб не зависело от среды, в которой они работали, и от назначения кода.

В результате предметно-ориентированная служба по умолчанию была безопасной, сохраняемой, легкодоступной, масштабируемой, отказоустойчивой и высокопроизводительной.

Однажды мы решили, что нам не нравятся коммерческие условия, на которых мы работали с поставщиком реляционных баз данных. Мы использовали базу данных для хранения части нашего большого хранилища, где содержалась история заказов и другие критически важные для бизнеса данные, объем которых стремительно увеличивался.

Мы загрузили одну из систем управления реляционными базами данных (RDBMS) с открытым исходным кодом, скопировали ее в репозиторий для таких зависимостей, написали развертывания и внесли несколько простых изменений в код, взаимодействующий с RDBMS. Это было просто, поскольку наша архитектура предусматривала разделение ответственности. Затем мы отправили изменение в пайплайн развертывания с непрерывной доставкой. Несколько тестов завершились с ошибкой; мы отследили ошибку и исправили ее и затем передали в пайплайн новую версию. Со второй попытки все тесты были пройдены, и мы поняли, что изменение готово к безопасному релизу. Мы развернули его в рабочей среде в следующем выпуске через несколько дней.

Вся эта деятельность заняла одно утро!

Без хорошего разделения ответственности на это ушли бы месяцы или годы, и в результате, скорее всего, ничего бы не получилось.

Рассмотрим простой пример. В предыдущей главе я приводил три фрагмента кода для решения одной задачи. Здесь я показываю эти фрагменты еще раз.

Листинг 11.1. Три примера разделения ответственности

```
def add_to_cart1(self, item):
    self.cart.add(item)

    conn = sqlite3.connect('my_db.sqlite')
    cur = conn.cursor()
    cur.execute('INSERT INTO cart (name, price)
values (item.name, item.price)')
    conn.commit()
    conn.close()

    return self.calculate_cart_total();

def add_to_cart2(self, item):
    self.cart.add(item)
    self.store.store_item(item)
    return self.calculate_cart_total();

def add_to_cart3(self, item, listener):
    self.cart.add(item)
    listener.on_item_added(self, item)
```

Мы уже обсуждали этот код в контексте связности, и для ее увеличения я использовал принцип разделения ответственности.

В первом плохом примере, `add_to_cart1`, разделение отсутствует. Этот код объединяет основную функцию — добавление товара в корзину — со сложным описанием способа хранения содержимого в реляционной базе данных. Затем, как побочный эффект, вычисляет итоговую сумму. Отвратительно!

Второй пример, `add_to_cart2`, гораздо лучше. Теперь код инициирует хранилище, но не упорядочивает его работу. Магазин может поставляться классу и быть чем угодно. В результате этот код значительно более гибкий. Тем не менее он по-прежнему объединяет функции хранения в корзине и вычисления общей суммы.

Третий пример представляет собой более полное разделение ответственности. Здесь код выполняет основную функцию — добавление в корзину, а затем просто сообщает, что в нее что-то положили. Он не знает — и ему все равно, — что будет дальше. Он полностью независим от функций хранения и вычисления общей суммы. В результате он значительно более связный и модульный.

Понятно, что как и в любом решении, существует возможность выбора. Я бы сказал, что код `add_to_cart1` просто плохой. От него надо отказаться, если мы думаем о разделении ответственности. Главный принцип — сочетание необходимой и случайной сложности. То есть то, как и где мы что-то храним, не должно иметь отношения к основному поведению корзины покупок, которое мы пытаемся создать. Необходимо четко отделить код, который работает с необходимой сложностью, от кода, который работает со случайной сложностью.

Разница между вторым и третьим примерами более тонкая. Это скорее вопрос контекста и выбора. Лично я предпочитаю `add_to_cart3`. Такое решение — самое гибкое из всех. Я могу реализовать или не реализовать разделение с помощью прослушивателя с внедренным методом, но мне очень нравится, что концепция хранилища удалена из области основной функции.

Это код, который я обычно пишу. На мой взгляд, версия `add_to_cart2` все еще недостаточно понятна. Я, конечно, считаю, что `store_item` — лучшая абстракция, чем соединения и SQL, но сама концепция все еще остается в области случайной сложности. Если вы положили что-то в настоящую корзину, вам не нужно удерживать в ней эти предметы!

Версия 3 дает наибольшую свободу выбора при малых реальных потерях. Разумная критика такого подхода в том, что здесь не видно, что хранилище работает, но на самом деле для данного фрагмента кода это неважно. Хранилище — побочный эффект работы компьютера, а не основное поведение при добавлении чего-либо в корзину. В третьем примере ясно видно, когда элемент добавляется, и видно, что может происходить что-то еще; нам просто все равно — что. Если нам не все равно, мы можем посмотреть.

Задумайтесь о тестируемости каждого из этих методов. Версия 1 просто ужасна. Нам понадобится база данных, поэтому провести тест сложно, и скорее всего, он окажется очень ненадежным и медленным. База данных

должна быть либо общей и изменяться вне теста, либо создаваться во время настройки теста, и каждый тестовый запуск будет выполнять еле-еле. Обе другие версии можно легко и эффективно проверить на фейках.

Основной аргумент против версии 3 — она менее наглядна. Я, конечно, согласен, что ясность — достоинство кода. Хотя на самом деле это всего лишь вопрос контекста. Здесь я обращаю внимание на код, отвечающий за добавление товара в корзину. Зачем ему знать, что будет дальше?

Подобное разделение ответственности помогло улучшить модульность и связность кода. В зависимости от того, насколько важно обеспечить взаимодействие нескольких частей, например прослушивателей, которые сохраняют результаты или вычисляют итоговую сумму, мы можем протестировать правильность этого взаимодействия в любом другом месте.

Так что полная картина менее понятна только потому, что мы смотрим не туда. Если бы мы придерживались довольно наивного взгляда на мир, следовало бы нам добавить в общую коллекцию, используемую для представления корзины, сведения о хранении и итоговой сумме? Конечно нет!

Одна из причин, по которой я считаю **разделение ответственности** главным принципом работы, в том, что оно напоминает мне, что стоит сосредоточиться на чем-то одном. Если мой код состоит из фрагментов, функции которых понятны с первого взгляда, я могу им гордиться. Если код приходится изучать дольше нескольких секунд, это плохая работа. Далее вам, возможно, понадобится разобраться, как этот фрагмент используется другими фрагментами, но у этих других фрагментов свои функции, и в идеале их надо выражать так же ясно.

ВНЕДРЕНИЕ ЗАВИСИМОСТИ

Внедрение зависимости — чрезвычайно полезный инструмент для достижения хорошего разделения ответственности. Это процесс, когда зависимости фрагмента кода передаются ему в качестве параметров, а не создаются в нем.

В нашем уже немного заезженном примере `add_to_cart1` соединение с базой данных явно создается и открывается внутри метода. Это значит, что использовать альтернативу невозможно. Мы тесно связаны с такой конкретной реализацией, даже с конкретно названным экземпляром базы

данных. Если **хранилище** в версии 2 передается в качестве параметра конструктора, это немедленно вызывает пошаговое изменение гибкости. Мы можем предоставить все, что реализует `store_item`.

В версии 3 метода `add_to_cart` слушателем может быть все, что реализует `on_item_add`.

Такое простое изменение отношений между поведением кода очень важно. В первом случае, в версии 1, код создает все, что ему нужно, поэтому он тесно связан с одной конкретной реализацией. Структура кода негибкая. В остальных случаях код кооперируется с другими компонентами системы, поэтому он мало знает и мало заботится о том, как они работают.

Внедрение зависимости часто ошибочно понимают как функцию инструмента или фреймворка, но это не так. Внедрение зависимости можно реализовать на большинстве языков и, конечно, на любом объектно-ориентированном или функциональном языке; это мощный инструмент проектирования. Я даже видел пример его эффективной реализации в сценариях оболочки Unix.

Внедрение зависимости — фантастический способ ослабления сцепления до полезного уровня и разделения ответственности. Еще раз отмечу, насколько согласованы все эти техники. Я сейчас говорю о важнейших, глубоких свойствах разработки ПО и ее продуктов, поэтому когда мы рассматриваем эти свойства с разных сторон, мы неизбежно замечаем их взаимосвязанность.

РАЗДЕЛЕНИЕ НЕОБХОДИМОЙ И СЛУЧАЙНОЙ СЛОЖНОСТИ

Эффективный способ улучшить качество разработки — реализовывать разделение ответственности определенным образом, то есть отделять необходимую сложность системы от случайной. Если концепция необходимой и случайной сложности — новая для вас, прочитайте знаменитую статью Фреда Брукса «Серебряной пули нет» (я упоминал ее ранее в этой книге) — эти важные идеи впервые были высказаны именно там.

Необходимая сложность системы — это сложность, присущая решению задачи, например, рассчитать комиссию за обслуживание банковского

счета, стоимость товаров в корзине или даже траекторию движения космического корабля. Обработка такой задачи представляет собой реальную ценность системы.

Случайная сложность — это все остальное, то есть задачи, которые появляются как побочный эффект того, что мы работаем на компьютере. Например, постоянство данных, вывод информации на экран, кластеризация, некоторые аспекты безопасности... все, что не имеет прямого отношения к решению поставленной задачи.

Случайность сама по себе не означает неважности; программы работают на компьютере, поэтому соответствующие ограничения и условия важны, но система, которая прекрасно справляется со случайной сложностью, не имея необходимой сложности, по определению бесполезна! Итак, в наших интересах работать над сокращением случайной сложности и не игнорировать ее.

Четкое разграничение случайной и необходимой сложности системы помогает эффективно разделить ответственность и тем самым улучшить дизайн системы.

Я хочу отделить логику, которая ответственна за вождение машины, от логики, которая отвечает за вывод информации на экран бортового компьютера; отделить логику для оценивания сделки от того, как эта сделка хранится или как о ней сообщается.

Для кого-то это очевидно, а для кого-то нет, но это важно, и лично мне кажется, что обычно код пишут не так. Большая часть кода, который я вижу, объединяет эти два разных класса ответственности. Слишком часто бизнес-логику смешивают с кодом вывода и элементами обеспечения сохраняемости; они вставлены в середину логики, которая ответственна или должна быть ответственна за основную предметную область (основную сложность) системы.

Это еще одна область, где, обращая особое внимание на тестируемость кода и системы, мы значительно улучшим качество дизайна.

Листинг 10.1 четко это демонстрирует. Код в нем невозможно тестировать, кроме как самым сложным способом. Конечно, я мог бы написать тест, который сначала создаст файл words.txt в указанном расположении, а затем запустит поиск в файле sorted.txt в другом указанном расположении. Но тест будет настолько медленным, раздражающе сложным и связанным со средой, что он прекратит работать в случае простого переименования или перемещения файлов.

Большая часть действий из листинга 10.1 даже близко не связана с поведением, которое требуется от кода. Код содержит почти всю случайную сложность, хотя на самом деле должен выполнять более важную функцию — в данном случае сортировку набора слов.

В листинге 10.2 связность лучше, но код по-прежнему нельзя тестировать как единое целое. В этом смысле у него те же проблемы, что и в листинге 10.1.

Листинг 11.2 — попытка улучшить этот код, отделяя случайную сложность от необходимой. На самом деле я бы не стал использовать определения «необходимый» или «случайный» для реального кода; я привожу их только для лучшего понимания.

Листинг 11.2. Разделение случайной и необходимой сложности

```
public interface Accidental
{
    String[] readWords() throws IOException
    boolean storeWords(List<String> sorted) throws IOException
}

public class Essential
{
    public boolean loadProcessAndStore(Accidental accidental) throws
        IOException
    {
        List<String> sorted = sortWords(accidental.readWords());
        return accidental.storeWords(sorted);
    }

    private List<String> sortWords(String[] words)
    {
        List<String> sorted = Arrays.asList(words);
        sorted.sort(null);
        return sorted;
    }
}
```

Если исходить из того, что мы реализуем функции случайной сложности, описанные в нашем случайном интерфейсе в листинге 11.2, этот код делает то же самое, что и в листингах 10.1 и 10.2, только лучше. Разделив ответственность — в данном случае разграничив случайную и необходимую сложность решаемой задачи, — мы значительно улучшили ситуацию. Этот код легче читать, он более ориентирован на важную задачу и в результате получается значительно более гибким. Если потребуется обрабатывать слова из другого расположения, помимо конкретного файла на конкретном

устройстве, мы сможем это обеспечить. Если потребуется сохранять отсортированный список в другом месте, нам тоже удастся это сделать.

Это все еще не очень хороший код. Следует дополнительно поработать над разделением ответственности, чтобы улучшить ориентированность и изолированность в целях удобочитаемости, а также с технической точки зрения.

В листинге 11.3 показано нечто близкое к идеалу. Безусловно, можно спорить о присвоенных мною именах, которые зависят от контекста. Но если смотреть исключительно с точки зрения разделения ответственности, надеюсь, вы замечаете очень большую разницу между кодом в листинге 10.1 и листингах 11.2 и 11.3. Даже в этом простом примере мы улучшили удобочитаемость, тестируемость, гибкость и полезность кода, следуя упомянутым ранее принципам проектирования.

Листинг 11.3. Удаление случайной сложности с помощью абстракции

```
public interface WordSource
{
    String[] words();
}

public interface WordsListener
{
    void onWordsChanged(List<String> sorted);
}

public class WordSorter
{
    public void sortWords(WordSource words, WordsListener listener)
    {
        listener.onWordsChanged(sort(words.words()));
    }

    private List<String> sort(String[] words)
    {
        List<String> sorted = Arrays.asList(words);
        sorted.sort(null);
        return sorted;
    }
}
```

Разделение необходимой и случайной сложности — хорошая отправная точка для перехода к коду с лучшим разделением ответственности. Это очень ценный подход, но он лежит на поверхности. А что с другим смешанным функционалом?

ВАЖНОСТЬ DDD

Предметная область также может быть нашим ориентиром при разработке ПО. Если использовать эволюционный, инкрементный подход к проектированию, то удается выявить и выделить новые понятия предметной области, которые в противном случае могли бы остаться незамеченными.

В листинге 11.4 показан пример кода на Python. Это мой вариант игры «Морской бой», цель которой — потопить флот противника.

В какой-то момент я начал сомневаться в правильности моего дизайна.

Листинг 11.4. Отсутствие понятия

```
class GameSheet:

    def __init__(self):
        self.sheet = []
        self.width = MAX_COLUMNS
        self.height = MAX_ROWS
        self.ships = []
        self._init_sheet()

    def add_ship(self, ship):
        self._assert_can_add_ship(ship)
        ship.orientation.place_ship(self, ship)
        self._ship_added(ship)
```

В определенный момент мне потребовалось добавить корабль в `GameSheet` — игровое поле из ячеек.

При написании кода я использовал тесты (TDD), и у меня уже была значительная серия тестов в `GameSheetTest`, ориентированных на сложности добавления корабля. Из одиннадцати тестов шесть проверяли, могу ли я разместить корабль на `GameSheet`. Я начал добавлять код проверки в `GameSheet`, и у меня получилось около 9–10 строк кода на три дополнительные функции.

Мне не нравился такой дизайн и тесты. Размеры и сложность выросли ненамного, но достаточно, чтобы я начал искать, что здесь не так. Затем я понял, что ошибся в разделении ответственности. Проблема заключалась в том, что я полностью проигнорировал важное понятие.

Мой класс `GameSheet` отвечал за расстановку кораблей и правила игры. Наличие «и» в описании класса или метода — сигнал. Он предупреждает, что у меня две функции, а не одна. В этом случае мне быстро стало понятно,

что в реализации не хватает понятия «правила». Я провел рефакторинг кода и тестов и извлек новый класс `Rules`. В листинге 11.5 показано, как упрощается работа с добавлением `Rules`.

Листинг 11.5. Слушаем код

```
class GameSheet:

    def __init__(self, rules):
        self.sheet = []
        self.width = MAX_COLUMNS
        self.height = MAX_ROWS
        self.rules = rules
        self._init_sheet()

    def add_ship(self, ship):
        self.rules.assert_can_add_ship(ship)
        ship.orientation.place_ship(self, ship)
        self._ship_added(ship)
```

Это сразу же упростило `GameSheet`. Больше не придется поддерживать коллекцию `Ships` и можно удалить 9–10 строк логики проверки, и это только начало изменений в коде, направленных на проверку соответствия правилам.

В конечном счете это изменение предоставило больше гибкости, позволив эффективнее тестировать логику `GameSheet` и `Rules` независимо друг от друга, и, возможно, в качестве побочного эффекта способствовало тому, чтобы код работал с разными версиями `Rules`. Я не беспокоился о том, какими могут быть эти правила. Я не делал ничего для поддержки возможных будущих новых правил, но теперь в моем коде был «шов», который мог оказаться полезным в будущем, а в настоящем позволял эффективнее тестировать код и улучшать дизайн. И все это благодаря только разделению ответственности.

Суть разделения ответственности — использовать задачу, которую вы решаете, для грамотного определения границ в коде. Это справедливо для различных уровней детализации. Можно начать с ограниченных контекстов, чтобы определить крупные модули (или службы), а затем улучшать дизайн по мере получения новых знаний о задаче и лучшего понимания читаемости или других свойств кода.

Очень важно поддерживать крайне низкую терпимость к сложности. Код должен быть простым и читаемым, и как только он перестает быть таким, стоит остановиться и начать искать способы упростить и прояснить не-понятный фрагмент.

В листингах 11.4 и 11.5 сигналом к такой остановке стали всего 10 строк кода и несколько тестов, которые, как мне показалось, располагались не там, где нужно. Это одна из причин, по которым я так ценю разделение ответственности. На самых ранних этапах процесса оно помогает выявлять проблемы, которые, если их не решить, приведут к уменьшению модульности и плохой связности в дизайне.

ТЕСТИРУЕМОСТЬ

Способ инкрементного проектирования с соблюдением принципа разделения ответственности основан на тестировании. Как я уже говорил, внедрение зависимости поможет улучшить дизайн, но еще более мощный инструмент для эффективного разделения ответственности — **тестируемость**.

Мало какой инструмент, за исключением таланта и опыта, способен так эффективно обеспечить качество разработки, как **тестируемость**.

Если мы хотим, чтобы наш код легко тестиировался, мы **должны** разделить ответственность, иначе тестам не хватит ориентированности. Они также станут более сложными, и труднее будет добиться их воспроизводимости и надежности. Необходимость контролировать переменные в целях обеспечения тестируемости побуждает нас создавать системы, обладающие всеми признаками высокого качества, которое мы ценим: модульностью, связностью, разделением ответственности, сокрытием данных и слабой связанностью.

ПОРТЫ И АДАПТЕРЫ

Имея целью разделение ответственности, мы стремимся улучшить модульность и связность систем. Это, в свою очередь, обеспечивает их слабую связанность (сцепление). Необходимо уделять пристальное внимание управлению связанностью систем при разработке на каждом уровне детализации.

Один из уровней, на котором это наиболее очевидно и важно, — те швы в коде, где одна «ответственность» взаимодействует с другой. Здесь следует быть особенно внимательными.

Рассмотрим простой пример (листинг 11.6). Это программа для сохранения чего-либо — в данном случае в хранилище Amazon AWS S3. У нас есть код, который обрабатывает все, что мы хотим сохранить, и код, который вызывает само хранилище, — хорошая отправная точка для разделения ответственности обработки и хранения.

Для работы системы требуются настройки для инициализации объекта `s3client`, чтобы сообщить ему необходимые данные учетной записи хранилища и так далее. Я намеренно не показывал код; уверен, что вы знаете несколько способов привести `s3client` в эту точку. Некоторые из них демонстрируют лучшее или худшее разделение ответственности. Сейчас просто сосредоточимся на том, что у нас есть.

Листинг 11.6. Сохранение строки в S3

```
void doSomething(Thing thing) {  
    String processedThing = process(thing);  
    s3client.putObject("myBucket," "keyForMyThing," processedThing);  
}
```

Код в листинге 11.6 написан с двух разных точек зрения. Мы привыкли к такому коду, но давайте поразмыслим. Всего две строки кода содержат два очень разных фокуса и уровня абстракции.

Первая строка делает что-то, что имеет смысл в мире функций и методов; возможно, «`process (thing)`» («обработать (что-то)») имеет смысл в контексте бизнеса. На самом деле это неважно, кроме того что это, очевидно, фокус, значимая часть кода. Это работа, которую мы хотим сделать, и строка написана с этой целью. Вторая строка — это, скажем так, пришелец. Чужой, который добавил случайную сложность в самое сердце логики.

Один из принципов связности заключается в том, что в заданных пределах уровень абстракции должен оставаться постоянным. Что, если сохранить это постоянство? Листинг 11.7 значительно лучше в этом отношении, хотя все, что мы сделали, — переименовали класс и метод.

Листинг 11.7. Сохранение строки в S3 с помощью порта

```
void doSomething(Thing thing) {  
    String processedThing = process(thing);  
    store.storeThings("myBucket," "keyForMyThing," processedThing);  
}
```

Изменение листинга 11.6 до состояния 11.7 дало результаты. Согласовав «вызов для сохранения» с другими концепциями этой функции, мы повысили степень абстракции. Кроме того, мы начали разработку в другом направлении.

Запомните код, который я не показывал; одним простым изменением я сделал несколько неверных реализаций для этой инициализации. Если я абстрагирую хранилище таким образом, то располагать всю инициализацию в рамках этого класса или модуля будет бессмысленно. Гораздо лучше полностью вывести ее во внешнюю среду.

Итак, теперь я собираюсь расположить всю эту инициализацию в другом месте. Это означает, что я могу протестировать ее абстрактно, отдельно от кода. Если я решу использовать внедрение зависимостей для организации хранилища, мне не понадобится реальное хранилище, чтобы протестировать код. Кроме того, я могу выбирать, где хранить переменные за пределами кода, организуя разные хранилища в разных контекстах, поэтому код становится более гибким.

Новую абстракцию можно представить как **порт**, или вектор, через который проходит поток информации. Решите вы сделать порт полиморфным или нет, полностью зависит от вас и задач вашего кода, но даже если вы его таким не сделаете, этот код все равно лучше. И это связано с тем, что вы повысите степень разделения ответственности и связности, поддерживая более последовательный уровень абстракции, а также сделаете код более читаемым и удобным в обслуживании.

Конкретной реализацией порта является **адаптер**, который действует как транслятор, переводя идеи: в данном примере из контекста «вещи» (things) в контекст «хранилище AWS S3» (AWS S3 Storage).

После внедрения изменения наш код ничего не будет знать об S3; он даже не будет знать, что S3 используется.

Ключевая идея заключается в том, что код пишется более последовательно. И поддерживает эту более последовательную абстракцию.

Схему, которую я здесь описал, иногда называют шаблоном **портов и адаптеров (P&A)**, а при использовании на уровне службы или подсистемы — **гексагональной архитектурой**.

Она имеет очень большое значение в проектировании. Ваш код почти никогда не обращается кциальному компоненту API, который он использует. Вы в большинстве случаев работаете с подмножеством API.

Порт, который вы создаете, должен предоставлять только минимально используемое подмножество, поэтому он почти всегда представляет собой упрощенную версию API, с которым вы взаимодействуете.

Когда пишешь книгу о коде, трудность в том, что примеры кода должны быть краткими и простыми, иначе идеи автора потеряются в сложности кода. Но как быть, когда необходимо проиллюстрировать упрощение?

Так что терпите. Представьте, что у нас есть целая система, написанная по образцу листинга 11.6: десятки, сотни, может быть, даже тысячи взаимодействий через `s3client`. Затем Amazon обновляет интерфейс до службы S3 или по крайней мере до клиентской библиотеки Java. Модель программирования в версии 2 отличается, так что теперь, чтобы воспользоваться преимуществами новой клиентской библиотеки, нам придется изменить десятки, сотни или тысячи строк кода.

Если мы создали собственные абстракции, порт и адаптер для S3, который делает именно то и только то, что нужно нашему коду, мы сможем использовать это не только в одном месте кода. Возможно, этот Р&А будет применим везде, а возможно, для более сложных сценариев придется создать свой Р&А. В любом случае мы значительно сократим объем технического обслуживания. Мы можем полностью переписать адаптер для использования новой клиентской библиотеки. Это никак не повлияет на код, который его задействует.

Такой подход воплощает многие черты хорошего дизайна. Работая над управлением сложностью, мы также ограждаем код от изменений, даже неожиданных или непредсказуемых.

КОГДА ИСПОЛЬЗОВАТЬ ПОРТЫ И АДАПТЕРЫ

Шаблон портов и адаптеров обычно обсуждается в контексте уровня трансляции на границах между службами (или модулями).

Это хороший контекст. В своей книге «Domain Driven Design» («Предметно-ориентированное проектирование»¹) Эрик Эванс рекомендует:

Всегда транслируйте информацию, которая входит в несколько ограниченных контекстов.

¹ В этой книге Эрик Эванс описывает моделирование предметной области в программном обеспечении как руководящий принцип проектирования.

При проектировании системы службы специалисты вроде меня советуют согласовать службы с ограниченным контекстом. Это минимизирует их связанность и улучшает модульность и связность.

Из двух указанных рекомендаций следует простой принцип: «Всегда транслируйте информацию, которая передается между службами», или, другими словами, «Всегда обменивайтесь данными между службами, используя порты и адаптеры».

Когда я начал писать предыдущее предложение, я сначала написал «правило», а не «принцип», а затем быстро поправил себя. Я не могу с чистой совестью назвать это правилом, потому что иногда оно нарушается. Тем не менее я настоятельно рекомендую исходить из того, что по умолчанию весь обмен информацией между службами транслируется через адаптер независимо от технической природы API.

Это не значит, что адаптер должен содержать много кода или быть сложным, но с точки зрения проектирования каждая служба или модуль должны иметь свое собственное «представление о мире» и последовательно его придерживаться. Если адаптер начинает передавать информацию, которая нарушает это представление, возникает серьезная проблема.

Можно защититься двумя способами: использовать адаптер, который приводит поступающую в систему информацию к удобному представлению, что позволяет проверять входные данные в той степени, в какой это нам необходимо; или изолировать данные, которым мы не доверяем, и игнорировать их, чтобы защитить систему от сомнительных внешних изменений.

Например, при создании системы обмена сообщениями существует то, что нам нужно знать, и то, что мы точно не должны знать.

Скорее всего, следует знать, кто отправил сообщение, куда отправил и объем сообщения, а также, вероятно, должна быть возможность повторить попытку отправки, если возникнет проблема. У нас точно не должно быть доступа к содержанию сообщения! Это немедленно свяжет технические аспекты обмена сообщениями с содержанием разговора, и это очень плохой дизайн.

Для кого-то это очевидно, а для кого-то нет, но на практике я часто встречаю код, который содержит именно такую ошибку. Если бы я создавал систему обмена сообщениями, я бы упаковал содержание сообщения в пакет, изолировав систему обмена сообщениями от содержимого пакетов — собственно сообщений.

ЧТО ТАКОЕ API?

Вопрос, что такое API, относится к области философии проектирования. Я бы дал довольно практическое определение:

Программный интерфейс приложения (API) – это вся информация, доступная потребителям службы или библиотеки, предоставляющей этот API.

Мое определение отличается от принятого разработчиками.

Со временем значение понятия API изменилось. Отчасти это, вероятно, связано с ростом популярности подхода REST при создании служб. Обычно, по крайней мере неформально, разработчики используют понятие API как синоним для «текст поверх HTTP». Это, безусловно, одна из форм API, но только одна, а их достаточно много.

Строго говоря, любое средство коммуникации между различными фрагментами программного кода является API. Здесь важно учитывать характер информации, с которой взаимодействует код.

Представьте функцию, которая принимает поток двоичных данных в качестве аргумента. Что такое API?

Это только сигнатура функции? Возможно; если функция рассматривает бинарный поток как черный ящик и никогда не заглядывает внутрь потока, тогда сигнатура функции определяет ее связанность с вызывающими операторами.

Однако, если функция взаимодействует с содержимым бинарного потока, это является частью ее контракта. Уровень взаимодействия определяет степень ее связанности с информацией в потоке.

Если первые восемь байт в потоке используются для кодирования его длины, и это все, что функция знает или должна знать о потоке, тогда API включает в себя сигнатуру функции плюс значение первых восемьми байт и то, как в них кодируется длина.

Чем больше функция знает о содержимом потока байтов, тем сильнее она с ним связана и тем больше площадь поверхности API. Многие команды упускают из виду тот факт, что структуры входных данных, которые понимает и обрабатывает их код, являются частью общедоступного API этого кода.

Адаптеры должны работать со всем API. Это может означать трансляцию или по крайней мере проверку содержимого входящего потока двоичных данных. Иначе код может сломаться, когда кто-то отправит нам неверный поток байтов. Это переменная, которую мы можем контролировать.

Добавление портов и адаптеров в точки связи между модулями и службами при проектировании системы по умолчанию дает больше преимуществ, чем отсутствие этих инструментов. Даже если «адаптер» — это заглушка на будущее, его наличие даст возможность в случае изменения характера API справиться с такими изменениями без необходимости переписывать весь код.

Это классическая модель Р&А. Я рекомендую рассмотреть возможность ее использования и на более низком уровне. Это не означает, что вы обязаны всегда явно предусматривать трансляции, но поддержание постоянного уровня абстракции в любом фрагменте кода, каким бы небольшим он ни был (см. листинг 11.6), — хорошая идея.

Я советую по умолчанию или по возможности всегда добавлять порты и адаптеры, когда код, с которым вы взаимодействуете, находится в другом контексте, например в другом репозитории или другом пайплайне развертывания. Следование защитным вариантам сценария в таких случаях сделает код более тестируемым и более устойчивым к изменениям.

ИСПОЛЬЗОВАНИЕ TDD ДЛЯ РАЗДЕЛЕНИЯ ОТВЕТСТВЕННОСТИ

Я уже рассказывал, как методы проектирования, направленные на улучшение **тестируемости** кода, помогают повышать его качество; не только в упрощенном смысле «работает ли он», но и в более глубоком смысле, когда продукт становится способным к постоянному обслуживанию и развитию.

Если мы ведем разработку, используя в качестве руководящего принципа разделение ответственности, включая поддержание единого уровня абстракции в любом заданном, даже небольшом, контексте, мы оставляем дверь открытой для постепенных изменений. Даже если мы еще точно не знаем, как сущности будут передаваться, храниться или взаимодействовать в целом, мы все равно можем писать код и добиваться прогресса.

Позже, когда мы узнаем больше, мы сможем использовать код так, как мы и не предполагали, когда писали его. Это означает более эволюционный подход к проектированию: мы расширяем систему шаг за шагом, по мере углубления понимания, до гораздо более сложных и функциональных версий.

TDD — самый мощный инструмент достижения подобной тестируемости.

В контексте разделения ответственности писать тесты становится тем сложнее, чем больший функционал объединяется в рамках одного теста. Опираясь на тесты, мы способны гораздо раньше оценить стоимость и выгоду наших решений.

Такая быстрая обратная связь, разумеется, представляет собой значительный плюс, поскольку позволяет обнаруживать недостатки в дизайне гораздо раньше, чем любой другой метод, предполагающий ограниченную степень контроля (кроме такого, при котором мы внезапно станем умнее, чем сейчас). Нет ничего плохого в том, чтобы быть семи пядей во лбу, но лучший способ достичь этого — работать грамотнее, и в этом цель данной книги. TDD — один из таких важных грамотных способов работы.

ИТОГИ

Разделение ответственности, безусловно, признак качественного кода. Если фрагменты кода выполняют одну и ту же функцию и один имеет хорошее разделение ответственности, а другой нет, то первый легче понимать, тестировать, изменять и он более гибкий.

Разделение ответственности также является самой простой эвристикой дизайна, которую можно применить.

Модульность и связность кода или системы могут быть предметом обсуждения. Я считаю эти идеи чрезвычайно важными, но их оценка может быть несколько субъективной. Если мнения, что именно считать плохой модульностью или связностью, скорее всего, совпадут, то оценки идеальной модульности или связности с большой вероятностью разойдутся.

Разделение ответственности — другая история. Если модуль, класс или функция выполняет более одной задачи, ответственность на самом деле не разделена, и спорить не о чем. Это означает, что разделение ответственности — отличный инструмент, направляющий нас по правильному пути, пути разработки более качественного ПО.

ГЛАВА 12

СОКРЫТИЕ ИНФОРМАЦИИ И АБСТРАКЦИЯ

Абстракция определяется как «процесс удаления физических, пространственных или временных свойств или характеристик при изучении объектов или систем, чтобы сосредоточить внимание на более важных деталях».

В названии этой главы я соединил два немножко различающихся понятия computer science; они различны, но взаимосвязаны между собой, и говоря об основных принципах программной инженерии, их лучше всего рассматривать вместе.

АБСТРАКЦИЯ ИЛИ СОКРЫТИЕ ИНФОРМАЦИИ

Я объединяю эти понятия, потому что не считаю разницу между ними достаточно значимой. Под абстракцией — или сокрытием информации — я подразумеваю наличие неких границ, или швов, в коде, чтобы при взгляде извне нам не было важно, какая информация скрыта внутри этих границ. Как потребителю функции, класса, библиотеки или модуля мне не нужно знать *ничего* о том, как они работают, а только то, как их использовать.

Некоторые люди понимают **сокрытие информации** гораздо более узко, но я не вижу в этом дополнительной ценности. Если вы упорно полагаете, что сокрытие информации связано только с данными (это не так), то всякий раз, когда я говорю «сокрытие информации», считайте, что я имею в виду абстракцию.

Если же вы думаете, что «абстракция» означает только «создание абстрактных объектов-понятий», учтите, что я говорю не об этом, хотя это часть определения; я имею в виду «скрытие информации».

Информация, которую я скрываю, — это поведение кода. Оно включает детали реализации, а также любые данные, которые он может использовать или не использовать. Абстракция, которую я представляю внешней среде, должна обеспечивать скрытие этих данных от других фрагментов кода.

Должно быть очевидно, что если наша цель — управлять сложностью, чтобы создавать более изощренные системы, чем те, что мы без труда удерживаем в памяти, то нам следует скрывать лишнюю информацию.

Мы хотим работать только с тем кодом, который сейчас перед нами, не беспокоясь о том, что происходит где-то еще и как ведут себя другие его фрагменты. Казалось бы, это элементарно, однако очень часто код устроен совсем не так. Иногда он уязвим для изменений, когда изменение в одном месте влияет на другие фрагменты. Бывает код настолько сложный для восприятия, что единственный способ добиться прогресса, работая с ним, — стать настолько умным, чтобы понимать, как работает большая часть системы. Это немасштабируемый подход!

ПОЧЕМУ ОБРАЗУЮТСЯ БОЛЬШИЕ КОМКИ ГРЯЗИ?

Иногда мы называем такие трудные для работы кодовые базы *большими комками грязи* (*big balls of mud*). Часто они настолько запутаны, что разработчики боятся их трогать. Такой грязный код время от времени встречается в большинстве организаций, особенно крупных, которые хоть когда-нибудь занимались разработкой ПО.

ОРГАНИЗАЦИОННЫЕ И КУЛЬТУРНЫЕ ПРОБЛЕМЫ

Причины сложны и разнообразны. Одна из самых частых жалоб, которую я слышу от индивидуальных разработчиков и команд, звучит так: «Мой руководитель не разрешает мне XXX», где «XXX» означает рефакторинг, тестирование, улучшение дизайна или даже исправить эту ошибку.

Безусловно, плохие работодатели встречаются. Если вам тоже не повезло, мой совет — ищите другую работу. Однако в подавляющем большинстве случаев эта жалоба просто не соответствует действительности или по крайней мере не совсем соответствует. В худшем случае это оправдание. Однако я не люблю обвинять людей, поэтому пусть это будет недопонимание.

Первое, что нужно отметить, — почему мы, разработчики, вообще должны спрашивать разрешения, чтобы сделать хорошую работу? Мы отраслевые эксперты, поэтому лучше всех понимаем, что работает, а что нет.

Если вы наймете меня, чтобы я написал для вас код, я буду обязан сделать все, что в моих силах. Это значит, что мне нужно оптимизировать свою работу так, чтобы надежно, регулярно и устойчиво поставлять код в течение длительного времени. Он должен решать поставленные задачи и удовлетворять потребности пользователей и запросы моих работодателей.

Итак, моя задача — создавать код, который работает, причем делать это постоянно и надежно в течение определенного времени. Мне необходимо поддерживать способность изменять код по мере того, как я узнаю больше о задаче, которую мы решаем, и о системе, которую мы разрабатываем.

Если бы я был шеф-поваром в ресторане, я, вероятно, мог бы приготовить очередное блюдо быстрее, если бы решил не мыть посуду и рабочее место после готовки предыдущего блюда. Хоть это и отвратительно, но скорее всего, это осталось бы незамеченным один раз или даже два. Но если бы я так поступал постоянно, меня бы уволили!

Рано или поздно я бы отравил гостей. Но даже если меня не уволят, то когда я доберусь до третьего блюда, скорость моей работы снизится, потому что мне будет мешать беспорядок, который я устроил. После приготовления каждого блюда я обязан приводить в порядок стол и все использованные кухонные принадлежности. Но я по-прежнему режу все тем же ножом, который уже затупился, и тому подобное. Знакомо?

Если бы вы были владельцем ресторана и взяли меня на работу поваром, вы бы никогда не сказали: «Я разрешаю вам точить ножи» или: «Вы обязаны убирать свое рабочее место», потому что, как профессионалы, мы оба считаем это само собой разумеющимся. Это все часть моих **обязанностей** как шеф-повара.

Будучи профессионалами, мы обязаны понимать, что требуется для разработки качественного программного обеспечения. Мы должны нести ответственность за качество создаваемого нами кода. Наша **обязанность** — делать свою работу хорошо. Это не альтруизм; это практичность и прагматизм. Такой подход отвечает интересам наших работодателей, наших пользователей и нас самих.

Если мы будем создавать и поддерживать качественный код, клиенты быстрее и проще получат нужные им функции. У них в руках окажется более актуальный и удобный в использовании инструмент, а мы сможем вносить изменения, не беспокоясь о возможных сбоях.

Разработка ПО предполагает планомерные, а не краткосрочные действия. Если вы пренебрегаете тестированием, избегаете рефакторинга или не считаете нужным тратить время на поиск более модульных и связных конструкций для обеспечения быстрой доставки, вы **замедляетесь, а не ускоряетесь**.

Логично, что компании-разработчики стремятся создавать ПО как можно более эффективно, поскольку это в конечном счете влияет на экономические показатели, касающиеся всех сотрудников.

Если мы хотим, чтобы компании, где мы трудимся, процветали, а мы создавали продукты, которые помогают им процветать, нам нужно работать эффективно.

Следует делать все возможное, чтобы быстрее создавать лучшие продукты. Исходные данные для этого уже известны: в книге «*Ускоряйся!*» описывается их часть, и конечно, речь не идет о том, чтобы «срезать углы» в ущерб качеству. Наоборот.

Один из ключевых выводов отчета State of DevOps о производительности команд разработчиков, описанный в книге «*Ускоряйся!*», заключается в том, что **между скоростью и качеством невозможен компромисс**. Вы не создадите продукт быстрее, если не поработаете как следует над его качеством.

Поэтому когда менеджер просит оценить сроки выполнения какой-то задачи, то не в ваших интересах, не в интересах вашего менеджера и не в интересах вашего работодателя экономить на качестве. При такой экономии вы будете работать медленнее, даже если ваш менеджер наивно полагает, что это не так.

Я, конечно, имел дело с организациями, которые намеренно или ненамеренно вынуждали разработчиков ускоряться. Однако обычно именно разработчики и команды решают, что повлечет за собой «ускорение».

Обычно от качества отказываются разработчики, а не менеджеры или организации. Работодатели хотят получить лучший продукт быстрее, а не худший продукт быстрее. На самом деле это даже не компромисс. Речь идет о выборе между лучшим продуктом, полученным быстрее, и худшим продуктом, полученным медленнее. «Лучше» неразрывно связано с «быстрее». Важно, чтобы все мы это осознали и приняли. Самые эффективные команды работают быстро не потому, что отказываются от качества, а потому, что принимают его необходимость.

Инженер-разработчик, если он профессионал, обязан признать эту истину и всегда предлагать такие советы, оценки и проектные решения, которые обеспечивают итоговый результат высокого качества.

Не тратьте усилия на оценки и прогнозы, сколько времени необходимо для хорошей работы; исходите из того, что ваши менеджеры, коллеги и работодатели хотят, чтобы вы делали свою работу хорошо, и делайте ее.

Существует стоимость выполнения работы. В кулинарии это, кроме про-чего, время, необходимое для чистки кухонных принадлежностей и ухода за ними. В разработке это рефакторинг, тестирование, время на обдумывание подходящего способа проектирования, исправление ошибок после их обнаружения, сотрудничество, общение и обучение. И все это не просто желаемые варианты; это основы профессионального подхода к разработке.

Любой человек может писать код, но не в этом заключается наша работа. Разработка ПО — это нечто большее. Мы решаем задачи, и это требует от нас тщательного внимания к дизайну и эффективности создаваемых решений.

ТЕХНИЧЕСКИЕ ВОПРОСЫ И ВОПРОСЫ ПРОЕКТИРОВАНИЯ

Чтобы позволить себе делать работу хорошо, ответьте на следующий вопрос: что для этого нужно? Это то, о чем в действительности я рассказываю в книге. Методы оптимизации процесса обучения, описанные в части II, и методы, описанные в этой части, — это тот набор инструментов, который позволяет нам работать лучше.

В частности, чтобы избежать больших комков грязи или исправлять их, важно принять определенный подход. Он заключается в том, что менять существующий код — хорошо и разумно.

Многие организации либо боятся менять код, либо относятся к нему с чрезмерным благоговением. Но я убежден, что если вы не можете или не хотите менять код, то такой код абсолютно неэффективен. Еще раз процитируем Фреда Брукса:

Как только дизайн замораживается, он устаревает¹.

Мой друг Дэн Норт поделился интересной идеей — у него дар облекать идеи в красноречивые формулировки. Он называл показателем качества «период полураспада продукта команды».

Ни у меня, ни у Дэна нет данных, подтверждающих эту идею, но она интересна. Он считает, что качество созданного командой продукта зависит от периода его полураспада, то есть от времени, которое требуется команде, чтобы переписать половину своего решения.

В модели Дэна хорошие команды, вероятно, сделают это за несколько месяцев; малоэффективные команды могут с этим не справиться никогда.

Теперь я почти уверен, что идея Дэна — контекстуальная; когда он сформулировал ее, он работал в хорошей, динамично развивающейся команде финансового трейдера. Полагаю, что во многих командах это правило не работает. Тем не менее зерно истины в нем, безусловно, есть.

Если в основе разработки, как я утверждаю, лежит способность обучаться, то когда мы изучаем что-то новое, меняющее наш взгляд на дизайн продукта (что бы это ни значило в нашем контексте), мы должны иметь возможность внести необходимые изменения, чтобы отразить этот новый взгляд.

Когда Кент Бек выбирал подзаголовок для своей знаменитой книги об экстремальном программировании, он остановился на формулировке «Embrace change» («Объять изменения»). Я не знаю, чем он при этом руководствовался, но теперь полагаю, что у этого подзаголовка гораздо более широкое значение, чем я предполагал, когда впервые читал эту книгу.

¹ Слова из книги Фреда Брукса «Мифический человеко-месяц».

Если мы убеждены, что должны сохранять возможность менять идеи, команды, код или технологию по мере того, как узнаем больше, то почти все остальное, о чем я говорю в этой книге, естественно следует из этого.

Организовывать работу так, чтобы оставить возможность совершить ошибку и исправить ее; расширять понимание решаемой задачи и отражать это понимание в дизайне; постепенно развивать и улучшать продукты и технологии — все это цели хорошей программной инженерии.

Чтобы реализовать все вышеперечисленное, следует двигаться вперед небольшими шагами, которые легко отыграть назад. Необходимо, чтобы к нашему коду можно было вернуться через несколько месяцев или лет и понять, что происходит. Изменения, вносимые в один фрагмент кода, не должны затрагивать другие. Следует иметь возможность быстро и эффективно проверять безопасность внесенных изменений, а также, в идеале, изменять архитектуру по мере углубления понимания или, возможно, роста популярности системы.

Все идеи в этой книге помогают реализовать такие возможности, но мне кажется, что **абстракция или сокрытие информации** — самый быстрый способ создания пригодных для использования систем.

ПОВЫШЕНИЕ УРОВНЯ АБСТРАКЦИИ

Что нужно сделать, чтобы добиться бруксовского улучшения на порядок? Один из способов — повысить уровень абстракции.

Наиболее часто при этом имеется в виду усиление взаимосвязи между высокоуровневыми диаграммами, которые иногда используются для описания систем. «Я бы хотел использовать схематическое изображение не только для представления системы, но и для ее программирования».

На протяжении многих лет было множество попыток реализовать эту идею, и новые реализации периодически появляются до сих пор. На момент написания этих строк последняя из таких идей — *low-code разработка*.

Однако несколько проблем, очевидно, мешают распространению такого подхода.

Один из популярных вариантов разработки — использование диаграмм для создания исходного кода. Идея в том, чтобы при помощи диаграммы создать общую структуру кода, а затем добавлять детали вручную. Эта стратегия в значительной степени провальна из-за одной трудноразрешимой проблемы: почти всегда по мере развития любой сложной системы появляются новые данные о ней.

В какой-то момент вам придется пересмотреть некоторые первоначальные решения. Это значит, что первая версия диаграммы и, следовательно, скелет системы неверны и их придется менять. Реализация «рейса туда и обратно» или создание скелета для кода, изменение деталей вручную, смена точки зрения, регенерация и корректировка диаграммы из кода с сохранением подробностей – сложные задачи. И до сих пор решить их посредством такого подхода не удалось.

Что, если полностью отказаться от написания кода вручную? Почему бы не применять диаграммы как код? Пробовали и это. Такие системы обычно очень привлекательны и хороши, пока они простые.

Однако на самом деле трудно повысить абстракцию до уровня, при котором рисовать изображения эффективнее, чем писать код. Вы теряете все преимущества традиционных способов программирования, такие как обработка исключений, контроль версий, поддержка отладки, библиотеки кода, автоматическое тестирование, шаблоны проектирования и так далее.

Такие системы хороши в качестве демоверсий, но не масштабируются на реальные решения. Графический язык для лаконичного выражения простых задач создать легко, но гораздо сложнее создать аналогичный визуальный язык, который станет универсальным инструментом выражения любой существующей логики. Тьюринг-полные языки программирования действительно содержат некоторые широко распространенные, но довольно низкоуровневые понятия. Уровень детализации, необходимый для описания и кодирования работающей сложной программной системы, кажется сложным и мелкомасштабным по своей сути.

Рассмотрим операцию добавления графика в таблицу. Большинство программ для работы с электронными таблицами предлагают инструменты, которые позволяют добавить график – внимание! – графическим способом. Можно выбрать несколько строк и столбцов данных в электронной таблице и затем выбрать изображение типа графика, который вы хотите добавить, и в простой ситуации программа сама генерирует график. Это хорошие инструменты.

Однако все становится сложнее, если данные не соответствуют одному из простых предопределенных шаблонов. Чем более специфичны требования к графику, тем более подробными должны быть инструкции для графической системы в электронной таблице. Наступает момент, когда ограничения инструментов усложняют их использование, а не упрощают. Теперь необходимо не только четко представлять, как должен работать график, но и хорошо понимать, как обойти или применить модель программирования, которую представлял себе разработчик этой графической системы.

Текст – это удивительно гибкий и лаконичный способ кодирования идей.

СТРАХ ЧРЕЗМЕРНОГО УСЛОЖНЕНИЯ

Многие обстоятельства подталкивают разработчиков снять с себя ответственность за качество. Одно из них — давление, которое на них оказывают реальное или воображаемое, с тем чтобы они выполняли работу эффективно. Я слышал, что сотрудники, отвечающие за коммерческое продвижение продуктов, беспокоятся, что разработчики переусердствуют в технической части. Этот страх обоснован, и виноваты в этом мы, технические специалисты. Иногда мы увлекаемся и чрезмерно усложняем системы.

АБСТРАКЦИЯ И ПРАГМАТИЗМ

Однажды мне пришлось заниматься проектом для крупной страховой компании. Это была «спасательная операция». Мой работодатель — консалтинговая компания — славился эффективной реанимацией проектов, в которые требовалось вдохнуть вторую жизнь после неудачных попыток реализации.

Проект дважды терпел неудачу. Над ним трудились более трех лет, но в нем ничего реально так и не заработало.

Мы взялись и добились приличного прогресса. К нам обратился архитектор из «стратегической группы» или кто-то подобный. Он настаивал, чтобы наше решение соответствовало глобальной архитектуре. Мне, как техническому руководителю проекта, пришлось изучить, чем это чревато.

У них был грандиозный план распределенной архитектуры компонентов на основе служб, которая абстрагировала весь бизнес. Они предусмотрели технические службы, а также полезное поведение на уровне предметной области. Инфраструктура обеспечивала безопасность и устойчивость, а также полную интеграцию систем предприятия.

Сейчас, я уверен, вы понимаете, что все было не так. Команда более чем из 40 человек написала много документации и кода, который, насколько я видел, не работал. Выпуск опаздывал примерно на три-четыре года. Все проекты должны были использовать заданную инфраструктуру, но ни один проект этого не сделал!

Это звучало фантастически, потому что это и была фантастика, а не реальность.

Мы вежливо отказались и построили свою систему, не воспользовавшись их архитектурой.

На бумаге все выглядело прекрасно, но это была только теория.

Мы — технические специалисты. В нашей работе мы следим за общими тенденциями. Одна из опасностей, о которых следует знать и остерегаться, — это погоня за технически блестящими идеями. Я так же, как и любой другой, интересуюсь техническими идеями. Это привлекательная часть нашей отрасли, знание, которое мы ценим. Однако если мы хотим быть инженерами, мы должны стать в определенной степени прагматиками, даже скептиками. В определение инженерии в начале этой книги я включил дополнение «с учетом экономических условий». Всегда следует искать самый простой путь к успеху, а не самый крутой или технологичный, который мы сможем добавить в резюме.

В любом случае за тенденциями необходимо следить. Будьте в курсе новых технологий или методов работы, но всегда оценивайте их применение в контексте решаемой задачи. Если вы собираетесь применять технологию или идею, то чтобы узнать, полезна ли она, проведите испытание, создайте прототип или эксперимент, но не начинайте сразу строить на ее основе архитектуру, от которой зависит будущее компании. Будьте готовы отказаться от нее, если она не сработает, и не рискуйте всем, опираясь на технологию, которая только выглядит круто.

По моему опыту, если вы серьезно стремитесь к простоте, у вас будет больше, а не меньше шансов в конечном итоге сделать что-то классное — и, как результат, улучшить свое резюме.

Еще одна причина, по которой мы увлекаемся чрезмерной сложностью, — стремление сделать свои решения **перспективными**. Если вы когда-нибудь говорили или думали: «Нам это не нужно сейчас, но, вероятно, понадобится в будущем», — то вы работали на перспективу. Я раньше увлекался этим, как и все остальные, но сейчас полагаю такой подход признаком дизайнерской и инженерной незрелости.

Мы пытаемся разработать дизайн с расчетом на перспективу, чтобы заручиться некоторой гарантией того, что сможем справиться с будущими улучшениями или изменениями требований. Это хорошая цель, но неправильное решение.

Из книги Кента Бека *«Экстремальное программирование»* я узнал о концепции **YAGNI**:

You Ain't Gonna Need It!
(Вам это не понадобится!)

Совет Кента таков: стоит писать код для решения задачи, которая стоит перед нами прямо сейчас, и только для нее. Я решительно поддерживаю этот совет, но он лишь часть целого.

Как я уже много раз повторял, разработка ПО — штука странная. Она почти бесконечно гибкая и чрезвычайно хрупкая. Мы можем создать любую конструкцию, которую захотим, но рискуем повредить ее, изменения. Усложняя свои решения, чтобы сработать на перспективу, разработчики пытаются подстраховать возможность изменения кода.

Они стараются исправить все ошибки, пока работают над задачей, чтобы не возвращаться к ним в будущем. Если вы дочитали до этого места, вы уже понимаете, что я считаю это очень плохой идеей. Так что же делать вместо этого?

Можно конструировать дизайн так, чтобы иметь возможность вернуться к нему в любой момент в будущем, когда мы узнаем что-то новое, и изменить его, воспользовавшись преимуществом почти бесконечной гибкости. Теперь необходимо справиться с хрупкостью кода.

Что нужно, чтобы обрести уверенность в возможности безопасного изменения кода в дальнейшем? Есть три способа, и один из них плохой.

Можно полагаться на свой ум. Вы будете думать, что полностью понимаете код и все его условия и зависимости, чтобы безопасно вносить изменения. Это модель героя-программиста, и хотя она плохая, судя по всему, она очень популярна.

В большинстве организаций обычно есть несколько супергероев¹, которых зовут спасать положение, когда что-то идет не так или требуется вносить сложные, но необходимые изменения. Если в вашей компании есть такой герой, ему необходимо поделиться своими знаниями и поработать с другими членами команды, чтобы сделать систему более понятной. Это гораздо более ценно, чем «спасение утопающих», которым обычно и занимаются супергерои.

По-настоящему избавиться от боязни менять код помогают **абстракция** и **тестирование**. Если мы абстрагируем код, то по определению скрываем

¹ Прекрасный вымыселенный пример можно найти в книге Джина Кима «Проект Феникс» («The Phoenix Project»), где персонаж Брент Геллер — единственный человек, который способен спасти положение.

сложность одной части системы от другой. Это означает, что можно безопасно изменять код в одной части системы с большей долей уверенности в том, что изменение, даже если оно ошибочно, не повлияет на другие части. Для большей уверенности нам нужно тестирование, но с ним не все так просто.

ПОВЫШЕНИЕ АБСТРАКЦИИ С ПОМОЩЬЮ ТЕСТИРОВАНИЯ

На рис. 4.2 я показал плоский график стоимости изменений, иллюстрирующий идеальную ситуацию, когда мы можем вносить любые изменения в любое время с примерно одинаковыми затратами времени и усилий.

Чтобы получить плоскую кривую стоимости изменений, нам понадобится эффективная, единственная стратегия регрессионного тестирования, что на самом деле означает, что она должна быть полностью автоматизированной. Внесите изменения и запустите тесты, чтобы увидеть, где ошибка.

Эта идея — один из краеугольных камней непрерывной доставки, самая эффективная известная мне отправная точка для реализации инженерного подхода. Мы работаем так, чтобы наш продукт всегда был готов к релизу, и мы определяем такую готовность к релизу с помощью эффективного автоматизированного тестирования.

Однако есть еще один важный аспект тестирования, помимо простого обнаружения ошибок, и он менее очевиден, если вы никогда не работали таким образом.

Это влияние тестируемости на дизайн, о котором я говорил ранее. Мы обсудим его более подробно в главе 14. Однако в контексте абстракции, если мы рассматриваем тесты как мини-спецификацию желаемого поведения кода, то описываем это желаемое поведение извне.

Вы не пишете спецификацию после завершения работы; она нужна вам до ее начала. Поэтому мы напишем спецификации (тесты) прежде, чем писать код. Поскольку у нас нет кода, нужно подумать, как максимально их упростить. Наша текущая задача — сделать спецификацию (тест) максимально простой и понятной.

Таким образом, мы неизбежно выражаем или по крайней мере должны выражать желаемое поведение кода с точки зрения пользователя, максимально ясно и просто. На данном этапе не нужно думать о деталях реализации этой мини-спецификации.

Если работать таким образом, мы абстрагируем дизайн по определению. Мы выбираем такой интерфейс для кода, в котором идеи выражать проще, чтобы создать хороший тест. Это означает также, что код просто использовать. Написание спецификации (теста) — это операция разработки. Мы создаем ожидаемый способ взаимодействия программистов с кодом, не затрагивая работу самого кода. И все это — до реализации кода. Подход, основанный на абстракции, помогает отделить то, что должен делать код, от того, как он это делает. На данном этапе мы мало или совсем ничего не говорим о реализации поведения; это следующий шаг.

Это практичный, прагматичный и легкий метод контрактного программирования¹.

СИЛА АБСТРАКЦИИ

Мы все знакомы с силой абстракции на потребительском уровне. Но на уровне производителей ПО мы зачастую уделяем слишком мало внимания абстракции в своем коде.

Ранние операционные системы не отличались абстракцией аппаратных средств, чего не скажешь об их современных преемниках. В наши дни, если я заменяю видеокарту на ПК, целый набор абстракций изолирует мои приложения от таких изменений, поэтому я уверен, что приложения, скорее всего, продолжат работать и отображать на экране все, что нужно.

Современные поставщики облачных услуг стремятся абстрагировать большую часть операционной сложности комплексных, распределенных, масштабируемых приложений. API, подобный S3 Amazon Web Service, обманчиво прост. Я могу отправить любую последовательность байтов вместе с меткой для ее извлечения и именем корзины, в которую ее

¹ Контрактное программирование — это подход к проектированию программного обеспечения, ориентированный на контракты, которые представляют собой спецификации, поддерживаемые системой или ее компонентами.

поместить, и AWS распространит ее по центрам обработки данных всего мира и сделает доступной для всех желающих с предоставлением соглашений об уровне обслуживания, которые обеспечат сохранение доступа во всех случаях самых серьезных сбоев. Это довольно сложно реализовать!

Абстракции также могут представлять организующий принцип в более широком смысле. Структуры данных с семантической маркировкой, такие как HTML, XML и JSON, очень часто используются в коммуникации. Некоторые предпочитают именно их, потому что они представляют собой простой текст, но это не совсем так. В конце концов, что означает простой текст для компьютера? Это поток электронов через транзистор, а электроны и транзисторы — тоже абстракции!

Привлекательность HTML или JSON для сообщений, пересылаемых между разными модулями кода, заключается в том, что структура данных становится явно выраженной при обмене данными, а схема передается вместе с содержимым. Существуют другие, гораздо более производительные механизмы передачи, таких как буферы протокола Google¹ или SBE², но чаще всего мы их не используем.

Разработчикам очень нравятся ужасно неэффективные механизмы вроде JSON или HTML, потому что с ними работает всё. Это возможно из-за другой важной абстракции — простого текста, который на самом деле не простой и не текст. Это протокол и абстракция, благодаря которой мы работаем с информацией, не слишком беспокоясь об организации этой информации, кроме как на довольно базовом уровне потока символов. Тем не менее это все же абстракция, скрывающая информацию.

Экосистема простого текста широко распространена в вычислениях, но это не что-то естественное или очевидное. Она создана искусственно и развивалась с течением времени. Нам следовало согласовать такие понятия, как порядок байтов и шаблоны кодирования, еще до того, как думать об

¹ Буферы протокола Google задуманы как уменьшенная, более быстрая и эффективная версия XML. Подробнее читайте на <https://bit.ly/39QsPZH>.

² Простое двоичное кодирование (Simple Binary Encoding, SBE) используется в сфере финансов. Это подход к кодированию двоичных данных для определения структур данных и генерирования кода в целях их преобразования на любом конце. Он делает некоторые свойства других подходов к кодированию семантических данных, но с меньшими затратами на производительность. Подробнее читайте здесь: <https://bit.ly/3sMr88c>.

основных абстракциях, позволяющих понять аппаратное обеспечение, на котором работают наши программы.

Абстракция «простой текст» чрезвычайно мощная. Еще одна такая абстракция — файлы в вычислениях, доведенная до апогея в модели вычислений Unix, в которой все является файлом. Мы можем подключать логику для создания новых, более сложных систем, передавая файлы с выхода одного модуля на вход другого. Все это создано искусственно и служит просто полезным способом представить и организовать происходящие процессы.

Абстракции лежат в основе нашей способности работать с компьютерами. Это ключ к пониманию систем и работе с системами, которые мы создаем, чтобы повысить ценность компьютеров. Один из способов взглянуть на то, что мы делаем, когда пишем код продукта (и в некотором смысле это единственное, что мы делаем), — это создавать новые абстракции. Главное — создавать их хорошо.

ДЫРЯВЫЕ АБСТРАКЦИИ

Дырявая (иначе «негерметичная») абстракция определяется как «абстракция, пропускающая детали, которые она должна абстрагировать».

Джоэл Спольски сформулировал некий закон:

Все нетривиальные абстракции дырявы.

Я иногда слышу, как люди оправдывают ужасный код, говоря что-то вроде: «Все абстракции негерметичны, так зачем беспокоиться?» Однако это полностью противоречит как смыслу приведенного высказывания, так и понятию абстракции в целом.

Без абстракции не существовало бы компьютеров и программ. Идея дырявых абстракций не аргумент против, скорее она указывает на то, что абстракции сложны и с ними нужно быть внимательными.

Существуют также различные виды утечек. Некоторых невозможно избежать, и самый эффективный способ борьбы с ними — заранее продумать,

где они могут появиться, и постараться минимизировать их влияние. Например, если вы строите систему с малой задержкой, которая обрабатывает данные на пределе возможностей оборудования, то абстракции «сборка мусора» и «оперативная память (RAM)» станут помехой, потому что они способствуют утечке времени, преобразуя задержку в переменную. Современные процессоры в сотни раз быстрее RAM, поэтому если вам важно время, доступ не должен быть произвольным. Стоимость времени различается в зависимости от того, откуда поступает обрабатываемая информация. Поэтому чтобы использовать все аппаратные возможности, нужна оптимизация; необходимо понимать абстракции, кэши, циклы предвыборки и так далее и учитывать их в своем дизайне, если вы хотите свести к минимуму влияние утечки.

Другой тип утечки — точка, в которой иллюзия, которую пытается передать ваша абстракция, разрушается, потому что вам не хватило времени, энергии или воображения, чтобы предусмотреть появление такой бреши в дизайне.

Служба авторизации, которая сообщает о функциональных сбоях как об ошибках HTML, и модуль бизнес-логики, возвращающий исключения `NullPointerException`, — примеры нарушения абстракции бизнес-уровня из-за технических сбоев. Оба этих примера — своего рода разрыв непрерывности иллюзии, которую призвана передать абстракция.

Чтобы справиться с такой утечкой, старайтесь поддерживать постоянный уровень абстракции, насколько это возможно. Удаленный компонент, представленный как веб-служба, вполне может сообщать о сбоях связи по HTML; это проблема в технической сфере абстракции сетей и коммуникаций, а не в самой службе. Об утечке в абстракции свидетельствует использование кодов ошибок HTML для сигнализации об отказах на бизнес-уровнях службы.

Считается, что абстракция в целом тесно связана с моделированием. Наша цель — создать модель задачи, которая поможет рассуждать о ней и работать. Мне нравится эта цитата Джорджа Бокса:

Все модели ошибочны, некоторые модели полезны¹.

¹ Цитата принадлежит статистику Джорджу Боксу, хотя сама идея возникла раньше.

Мы всегда находимся в таком положении. Как бы хороши ни были наши модели, они — представление об истине, а не сама истина. Но они могут быть чрезвычайно полезны, даже если в корне не соответствуют действительности.

Наша цель не достичь совершенства, а создать полезные модели, которые можно использовать в качестве инструментов для решения задач.

ВЫБОР ПОДХОДЯЩИХ АБСТРАКЦИЙ

Природа абстракций, которые мы выбираем, имеет значение. Абстракции — это модели, и они не универсальны.

Хороший пример — карты (географические, а не структура данных компьютерного языка). Все карты, конечно, являются абстракциями реального мира, но типы абстракций различаются в зависимости от наших потребностей.

Если я веду лодку или самолет к месту назначения, мне нужна карта, которая позволяет измерять расстояние между двумя точками. (Этот тип карты, строго говоря, называется диаграммой; я могу определить азимут на карте, и если я буду придерживаться курса, то попаду в нужное место.) Впервые картографическую проекцию для навигации использовал Меркатор в 1569 году.

Чтобы не утомлять вас излишними подробностями, отмечу, что карты с постоянным азимутом построены на основе *линий румба*. Можно определить азимут на такой карте и проплыть (или пролететь) по этому азимуту из точки А в точку Б.

Земля, как мы все знаем, не плоская. Это шар, так что на самом деле полученная траектория будет не кратчайшим расстоянием между А и В, потому что на поверхности сферы кратчайшее расстояние между двумя точками — это кривая, а это значит, азимут постоянно меняется. Таким образом, абстракция диаграммы скрывает более сложную математику искривленных плоскостей и предоставляет практический инструмент для прокладывания курса.

Утечка в этой абстракции в том, что преодолевать приходится расстояние больше абсолютно необходимого, но это нормально, поскольку мы применяем оптимизацию для простоты построения маршрута и передвижения.

Совершенно другая абстракция используется в большинстве схем метро. Такую схему изобрел Гарри Бек в 1933 году.

Она стала классической и используется в метрополитенах всего мира. Гарри понял, что при передвижении в лондонском метро вам все равно, где вы находитесь, когда вы в пути. Поэтому он создал топологически точную карту линий движения поездов, которая не имела привязки к физической географии.

Благодаря такому стилю абстракции пассажирам понятна схема движения поездов и пересадок на другие линии. Но эта абстракция не сработает, если вы попытаетесь использовать ее для перемещения между станциями по земле. Некоторые станции расположены в нескольких шагах друг от друга, хотя карта этого не показывает; другие расположены близко на схеме, но в действительности это не так.

Я хочу сказать, что разные абстракции для одного явления — нормальная практика. Если бы нам поручили протянуть сетевой кабель между станциями лондонского метро, было бы глупо выбирать для работы схему Гарри. Но если мы планируем поехать ужинать на Лестер-сквер от станции «Арсенал», глупо использовать географическую карту.

Абстракция и моделирование, лежащее в ее основе, составляют фундамент проектирования. Чем точнее абстракции отвечают решаемой задаче, тем лучше окажется дизайн. Заметьте, я не сказал: «Чем точнее абстракция». Как ясно показывает схема метро, чтобы быть полезной, абстракция не обязательно должна быть точной.

Опять же, тестируемость обеспечит раннюю обратную связь и вдохновит на создание полезных абстракций.

Один из популярных аргументов против модульного тестирования, а иногда и против TDD, таков: тесты и код при их использовании связываются вместе, в результате чего систему сложнее изменить. Эта критика больше относится к юнит-тестированию, когда тесты пишут после завершения кода. Такие тесты неизбежно тесно связаны с проверяемой системой, потому что они написаны как тесты, а не как спецификации. К TDD это относится в меньшей степени, поскольку мы сначала пишем тест (спецификацию) и затем абстрагируемся от проблемы, как я уже рассказывал.

Тонкость и огромная ценность TDD, однако, в том, что если написать абстрактную спецификацию, сосредоточившись на том, что должен делать

код, а не на том, как он достигает результата, то абстракцией будет служить то, что выражает тест. Итак, если тест чувствителен к изменениям, то и абстракция тоже. Поэтому необходимо создавать лучшие абстракции. Я не знаю другого способа получить качественную обратную связь.

В следующей главе речь пойдет о связанности (сцеплении). Несоответствующая связанность — одна из самых серьезных проблем разработки. Весь этот раздел книги посвящен стратегиям управления связанностью. Проблема в том, что бесплатный сыр бывает только в мышеловке. Чересчур абстрактный дизайн так же плох, как и недостаточно абстрактный. И тот и другой может быть неэффективным и привести к высоким затратам на разработку и обеспечение производительности. Существует золотая середина, и тестируемость системы — это инструмент для ее достижения.

В целом мы должны стремиться к способности менять реализацию и, насколько это возможно, дизайн без больших дополнительных усилий. И здесь нет универсального рецепта. Это навык хорошего разработчика, и он приходит с практикой и опытом. Необходимо развивать чутье, чтобы выявлять варианты дизайна, доступные для корректировки в будущем, тем самым оставляя возможность выбора.

Это означает, что любой мой совет зависит от контекста. И я даю рекомендации, а не жесткие правила.

АБСТРАКЦИИ ИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

Моделирование предметной области поможет выбрать направление дизайна — добиться **естественного для предметной области разделения ответственности**, а также лучше понять задачу, над которой вы работаете. Такие методы, как **событийный штурм**¹, отлично подходят для определения области задачи.

Событийный штурм помогает выявить кластеры поведения, а из них получить интересные концепции; интересные концепции, в свою очередь, — основа для формирования модулей или служб в дизайне вашего ПО.

¹ Событийный штурм (event storming) — это метод совместного анализа, предложенный Альберто Брандолини (Alberto Brandolini), для моделирования взаимодействий в предметной области.

С помощью событийного штурма можно выделить узкие контексты и естественные линии абстракции в предметной области, как правило, с лучшим разделением, чем определяемые технически.

ПРЕДМЕТНО-ОРИЕНТИРОВАННЫЕ ЯЗЫКИ

Предметно-ориентированные языки (*domain-specific language, DSL*) – один из инструментов повышения уровня абстракции. Однако по определению DSL не являются универсальными. Они более узконаправленны и могут быть более абстрактными, скрывающими детали.

Такой эффект особенно хорошо проявляется в системах, основанных на диаграммах. Мы видим влияние DSL – в данном случае графического – на решение узких задач. В этом пространстве подобные более ограниченные способы представления идей чрезвычайно эффективны и полезны.

DSL – очень полезный инструмент, он играет важную роль в разработке мощных, даже программируемых пользователем систем, но это не совсем тема данной книги, поэтому я не буду подробно ее касаться. Однако, если вкратце, нет лучшего способа создать эффективные тестовые кейсы, чем использовать DSL для выражения желаемого поведения системы в виде исполняемых спецификаций.

АБСТРАКТНАЯ СЛУЧАЙНАЯ СЛОЖНОСТЬ

Программы работают на компьютерах. Работа компьютеров представляет собой ряд абстракций и ограничений, без которых не обойтись. Некоторые из них глубокие, на уровне информации и теории информации, например, параллелизм и синхронная либо асинхронная передача данных. Другие относятся к аппаратной реализации, например, архитектура кэша процессора или разница между RAM и онлайн-накопителем.

Эти понятия нельзя игнорировать, кроме как в простейших системах, и в зависимости от характера системы, возможно, их придется глубоко изучать. Однако это дырявые абстракции. Если сеть падает, это влияет и на работу программ.

В целом я всегда стараюсь абстрагировать область случайной и необходимой сложности (предметную область), насколько это возможно. Это требует навыков мышления инженера и проектировщика.

Первый вопрос в том, как представить случайную сложность в области необходимой сложности. Что должна знать логика системы о компьютере, на котором она работает? Нам следует сделать это знание минимальным.

В листинге 12.1 показаны три примера связности из главы 10. Если рассмотреть эти фрагменты кода с точки зрения абстракции и разделения случайной и необходимой сложности, мы увидим их по-новому.

Листинг 12.1. Три примера связности (снова)

```
def add_to_cart1(self, item):
    self.cart.add(item)

    conn = sqlite3.connect('my_db.sqlite')
    cur = conn.cursor()
    cur.execute('INSERT INTO cart (name, price) values (item.name, item.
    price)')
    conn.commit()
    conn.close()

    return self.calculate_cart_total();

def add_to_cart2(self, item):
    self.cart.add(item)
    self.store.store_item(item)

    return self.calculate_cart_total();

def add_to_cart3(self, item, listener):
    self.cart.add(item)
    listener.on_item_added(self, item)
```

Первый пример, `add_to_cart1`, вообще не абстрагируется и в результате беспорядочен.

Следующий, `add_to_cart2`, лучше. Мы добавили абстракцию для хранения информации. Мы создали шов `store` в коде, и это придало коду большую связность. Теперь четко разделяется необходимая сложность функций добавления товаров в корзину и вычисления итоговой суммы покупки и случайная сложность, вызванная тем, что компьютер делает различие между энергозависимой, но быстрой оперативной памятью и более медленным, но энергонезависимым диском.

Наконец, в `add_to_cart3` появляется абстракция, которая делает код необходимой сложности целостным. Абстракция неизменна, с намеком на введение объекта, заинтересованного в событии, `Listener`.

С точки зрения согласованности абстракции `add_to_cart3`, на мой взгляд, лучший пример. Даже концепция хранилища в нем удалена.

Прелесть ее в том, что модель свободна от случайностей, и, как следствие, ее легко тестировать или улучшать с помощью нового поведения `on_item_added`.

Стоимость этой абстракции — утечка, которая в теории мешает считать `add_to_cart3` лучшим вариантом. Что произойдет, если попытка сохранения не удастся? Что, если в пуле соединений базы данных закончатся соединения, или на диске закончится место, или будет поврежден сетевой кабель, соединяющий код и базу данных?

Первый пример не является модульным, ему не хватает связности, в нем пересекаются случайная и необходимая сложности, нет разделения функций; это просто плохой код!

Два других примера лучше, но не в силу каких-то искусственных представлений о красоте или элегантности, а из практических, прагматических соображений.

Варианты 2 и 3 более гибкие, менее связанные, более модульные и связные благодаря хорошему разделению функций и выбору абстракций. Выбор абстракции из этих двух на самом деле зависит от дизайна и должен определяться контекстом, в котором работает код.

Существует несколько способов это представить.

Если, например, нехватка места в хранилище связана с добавлением товара в корзину, нам придется отменить изменение в корзине. Это нежелательно, потому что целостность абстракции нарушает технические особенности хранилища. Можно попробовать ограничить масштаб утечки (листинг 12.2).

Листинг 12.2. Уменьшение утечки абстракции

```
def add_to_cart2(self, item):
    if (self.store.store_item(item))
        self.cart.add(item)

    return self.calculate_cart_total();
```

Здесь мы отошли от полностью абстрагированной версии 3 и позволили концепции хранилища существовать в абстракции. Мы представили

транзакционный характер отношений между хранением и добавлением товара в корзину как возвращаемое значение успеха или неудачи. Обратите внимание, что мы не портим абстракцию, возвращая специфичные коды ошибок и создавая утечку в абстракции уровня предметной области. Мы ограничили техническую природу сбоя булевым возвращаемым значением. Это означает, что проблемы выявления и сообщения об ошибках решаются где-то еще — в данном случае, возможно, внутри реализации хранилища.

Это еще один пример попытки свести к минимуму влияние неизбежных утечек в абстракции. Мы также моделируем сбои и абстрагируем их. Если мы еще раз взглянем на все возможные реализации магазина, то увидим, что в результате наш код стал более гибким.

Есть еще один вариант — организовать менее строгое, более разделенное представление. В `add_to_cart3` в листинге 12.1 можно представить, что за событием `on_item_added` следуют какие-то «гарантии»¹. Представим, что, если по какой-то причине в `on_item_added` возник сбой, событие будет повторяться до тех пор, пока не сработает. (На самом деле нужно придумать что-то поумнее, но чтобы не усложнять пример, оставим так!)

Теперь мы уверены, что в какой-то момент в будущем либо хранилище (`store`), либо что-то другое, отвечающее на `on_item_added`, будет обновлено.

Это, безусловно, усложняет коммуникацию, связанную с `on_item_added`, но лучше сохраняет абстракцию и, в зависимости от контекста, оправдывает дополнительную сложность.

Приводя эти примеры, я стремлюсь не разобрать все возможные ситуации, а продемонстрировать некоторые инженерные компромиссы, зависящие от контекста.

Мыслить как инженер — значит искать, как что-то может пойти не так. Вы, вероятно, помните, что Маргарет Гамильтон, вводя термин «*программная инженерия*», назвала этот способ мышления краеугольным камнем своего подхода.

¹ Специалисты в теории информатики совершенно справедливо отметят, что невозможно обеспечить гарантированную доставку. Они имеют в виду, что нельзя гарантировать доставку только однажды, но мы можем с этим работать. См. <https://bit.ly/3ckjiwL>.

В этом примере мы представили, что произойдет, если в хранилище случится сбой. Мы выяснили, что при этом в абстракции возникает утечка, и нам пришлось еще немного подумать и найти несколько способов устранить эту утечку.

ИЗОЛИРУЙТЕ КОД ОТ СТОРОННИХ СИСТЕМ

Другое явное различие между вариантом 1 `add_to_store` и вариантами 2 и 3 заключается в том, что вариант 1 предоставляет и связывает код с конкретным сторонним кодом, в данном случае `sqlite3`. Это популярная библиотека Python, но код теперь привязан к этой конкретной сторонней библиотеке. Еще одна причина, почему этот код является худшим из трех, заключается в том, что он связан со сторонним кодом.

Вырезание блока кода с привязкой к `sqlite3`, с операторами соединениями и вставки, и перемещение его подальше от основного кода, который не должен иметь с этим дела, — несущественно с точки зрения стоимости. Но это важный шаг к большему обобщению. Так мало работы, но такой большой выигрыш.

Как только мы используем сторонний код внутри нашего кода, мы сцепляемся с ним. Я советую всегда изолировать код программы от стороннего с помощью собственных абстракций.

Небольшое предостережение, прежде чем я продолжу. Очевидно, что язык программирования, который вы используете, и его общие вспомогательные библиотеки тоже представляют собой сторонний код. Я не предлагаю писать собственную оболочку для `String` или `List`, поэтому, как всегда, мой совет — это скорее рекомендация, чем жесткое правило. Тем не менее продумайте как следует, что разрешить внутри своего кода. По умолчанию я допускаю стандартные понятия языка и его библиотеки, но только не сторонние библиотеки, которые не поставляются вместе с моим языком.

Доступ к любым сторонним библиотекам, которые я использую, будет осуществляться через мой собственный фасад или адаптер, который абстрагирует и, таким образом, упрощает интерфейс с этими библиотеками и обеспечивает довольно простую изоляцию между моим кодом и кодом библиотеки. По этой причине я опасаюсь всеобъемлющих фреймворков, которые пытаются навязать мне собственную модель программирования.

Такой подход может показаться немного экстремальным или таковым, но он означает, что мои системы получаются более гибкими и более компонуемыми.

Даже в рассматриваемом здесь простейшем примере `add_to_cart2` представляет собой абстракцию, которая имеет смысл в контексте моей реализации хранилища. Я могу создать версию, которая по сути является блоком кода, реализующим хранилище `sqlite3` из `add_to_store1`, но также могу написать хранилище совершенно другого типа, без необходимости изменять реализацию `add_to_cart2`. У меня есть возможность использовать один и тот же код в разных сценариях и даже написать некую составную версию хранилища, где товары будут располагаться в нескольких местах, если возникнет такая необходимость.

Наконец, мы можем протестировать код на такую абстракцию; это всегда будет проще, чем реальность. В результате мое решение станет более гибким и его будет легче изменить, если я допущу ошибку. Для этого потребуется совсем немного дополнительной работы.

ВСЕГДА СКРЫВАЙТЕ ИНФОРМАЦИЮ, ЕСЛИ ЭТО ВОЗМОЖНО

Еще одна настоятельная рекомендация, которая поможет оставить код открытым для будущих изменений, не нарушая принципа YAGNI, — выбор более общих представлений, а не более конкретных. Но это несколько упрощенный совет. Нагляднее всего его можно проиллюстрировать на примере сигнатур функций и методов.

В листинге 12.3 показаны три версии сигнатуры функции. Одна из них мне кажется намного лучше, чем другие, хотя, как обычно, это зависит от контекста.

Листинг 12.3. Лучше скрывать информацию

```
public ArrayList<String> doSomething1(HashMap<String, String> map);

public List<String> doSomething2(Map<String, String> map)

public Object doSomething3(Object map);
```

Первый пример слишком конкретный. Действительно ли важно, чтоозвращаемое значение — это `ArrayList`, а не другой тип списка? Полагаю, что в каких-то крайне редких случаях это так, но в целом абсолютно не важно. Имеет значение только то, что это список, а его конкретный тип вторичен.

«ОК, — слышу я ваш вздох, — всегда надо выбирать самое абстрактное, самое общее представление». Да, но в разумных пределах, поддерживающих абстракцию. Я поступил бы глупо, если бы последовал этому совету и создал такую плохую сигнатуру функции, как `doSomething3`. Она общая до такой степени, что вообще бесполезна. Опять же, возникают ситуации, когда `Object` — подходящий уровень абстракции, но это происходит или должно происходить редко и всегда относится к сфере случайной, а не необходимой сложности.

Так что в целом `doSomething2`, наверное, оптимальный пример. Он достаточно абстрактный, в нем нет такой сильной привязки к технической специфике, как в `doSomething1`, и одновременно достаточно конкретный, чтобы эффективно представлять и поддерживать указания о том, как обрабатывать создаваемую информацию и мои ожидания от обрабатываемой информации.

Наверняка вы уже устали от повторений, но подчеркну: если вы проектируете системы с учетом их **тестируемости**, то вам проще найти золотую середину для абстракций. Тесты и моделирование работы интерфейса помогают проверить понимание этого интерфейса и применить это понимание к тестируемому коду.

В сочетании с общим принципом сокрытия информации и выбором более обобщенного представления используемой информации это позволит держать двери открытыми для будущих изменений.

ИТОГИ

Абстракция лежит в основе разработки ПО. Это жизненно важный навык для начинающего инженера-программиста. Большинство моих примеров, скорее всего, объектно-ориентированы, потому что я смотрю на код именно так. Однако абстракция справедлива и для функционального, и даже для ассемблерного программирования. Код, каким бы он ни был, становится лучше, если предусмотреть в нем швы, скрывающие информацию.

ГЛАВА 13

УПРАВЛЕНИЕ СВЯЗАННОСТЬЮ

Связанность — одно из самых важных понятий, которое следует учитывать при организации управления сложностью.

Связанность (иначе называемая **цепление**) определяется как «степень взаимозависимости между программными модулями; мера того, насколько тесно связаны две подпрограммы или два модуля; прочность взаимосвязей между модулями».

Связанность — важная часть любой системы, хотя мы часто не обсуждаем ее. Мы говорим о ценности слабосвязанных систем, но давайте внесем ясность: если компоненты системы вообще не связаны, то они не могут взаимодействовать друг с другом. Иногда это полезно, а в других случаях — нет.

Связанность — это не то, от чего всегда стоит избавляться.

СТОИМОСТЬ СВЯЗАННОСТИ

Связанность самым непосредственным образом влияет на наши возможности надежно, воспроизводимо и устойчиво создавать и поставлять программные продукты. Чтобы разрабатывать ПО любого масштаба и сложности, необходимо управлять связанностью в системах и в организациях, которые производят эти системы.

Настоящая причина, по которой важны такие свойства систем, как **модульность** и **связность**, и такие методы, как **абстракция** и **разделение ответственности**, заключается в том, что они помогают уменьшить

связанность. Это напрямую влияет на скорость и эффективность достижения результата, а также на масштабируемость и надежность как продукта, так и всей организации.

Если не уделять внимания проблемам связанности, в программах будут формироваться большие комки грязи и разработчики не смогут вносить изменения в действующие продукты. Управлять связанностью чрезвычайно важно!

В предыдущей главе я рассказал, как абстракция помогает разорвать некоторые связи даже крошечных фрагментов кода. Если не прибегать к абстракции, то код окажется сильно связан и изменения в одной части системы будут влиять на поведение кода в другой.

Кроме того, следует разделять необходимую и случайную сложность, иначе код окажется сильно связан и придется одновременно заботиться об очень сложных процессах, например параллелизме, и в то же время полагать, что баланс на счете правильно суммируется. Это не лучший способ работы!

Это не значит, что тесная связанность — это плохо, а слабая — хорошо; не все так просто.

Однако самая частая общая ошибка разработчиков — стремление к чрезмерной связанности. Издержки слишком слабой связанности тоже существуют, но они, как правило, намного ниже стоимости слишком тесной связанности. Таким образом, в целом **слабая связанность более предпочтительна**, хотя необходимо понимать, что этот выбор требует определенных компромиссов.

МАСШТАБИРОВАНИЕ

Возможно, самое большое коммерческое влияние связанность оказывает на масштабирование разработки. Продукт не станет лучше и не будет готов быстрее, если просто увеличить количество разработчиков, — это осознали еще не все, но нам это давно известно. Существует довольно серьезное ограничение на размер команд: увеличение численности команды замедляет работу (см. главу 6).

Причина этого в связанности. Если ваша и моя команды **связаны разработкой**, хорошо бы согласовывать наши релизы. Например, мы организуем процесс так, чтобы вы могли отслеживать все изменения, которые я вношу в свой код. Это работает лишь для небольшого числа разработчиков и команд, и такое поведение сложно контролировать. Рост стоимости информирования быстро становится неуправляемым.

Существуют способы минимизировать такие накладные расходы и сделать координацию максимально эффективной. Лучший из них — использовать **непрерывную интеграцию** (CI – continuous integration). В этом случае весь код хранится в общем пространстве, репозитории, и каждый раз, когда кто-то из нас что-то меняет, мы проверяем работоспособность системы. Это важно для всех, кто трудится совместно. Даже небольшие группы нуждаются в ясности, которая достигается благодаря непрерывной интеграции.

Этот подход также масштабируется значительно лучше, чем можно ожидать. Например, Google и Facebook применяют его почти ко всему коду. Недостаток такого масштабирования в том, что придется вкладывать значительные средства в разработку репозиториев, билдов, CI и автоматического тестирования, чтобы достаточно быстро получать обратную связь об изменениях и управлять разработкой. Большинство организаций не могут или не хотят вкладывать средства в изменения, необходимые для такого подхода¹.

Эту стратегию можно рассматривать как копирование недостатков связанныности. Мы делаем обратную связь настолько быстрой и эффективной, что продвигаемся вперед, даже когда код и команды связаны.

МИКРОСЕРВИСЫ

Другая эффективная стратегия — в том, чтобы избавиться от связанности или по крайней мере снизить ее степень. Она подразумевает использование **микросервисов**, которые представляют собой наиболее масштабируемые способы разработки, хотя большинство людей имеют о них неверное

¹ В другой моей книге, «Continuous Delivery», описываются методы, необходимые для масштабирования этих аспектов инженерной разработки.

понятие. Микросервисы значительно сложнее, чем кажется, и требуют довольно непростого дизайна.

Как вы, возможно, поняли из этой книги, я верю в сервисную модель организации систем. Это эффективный инструмент выделения модулей и создания прочных швов абстракции, о чём мы говорили в предыдущей главе. Однако важно понять, что эти преимущества сохраняются независимо от способа развертывания ПО. Они на несколько десятилетий опередили идею микросервисов.

Термин возник впервые в 2011 году. Но микросервисы не явились чем-то абсолютно новым. Они объединили все подходы и практики, которые уже так или иначе применялись ранее. Существует несколько определений того, что такое микросервисы. Я для себя использую следующее.

Микросервисы обладают свойствами:

- малый размер;
- нацеленность на решение одной задачи;
- согласованность с ограниченным контекстом;
- автономность;
- возможность независимого развертывания;
- слабая связанность.

Я уверен, вы заметили, что этот список тесно перекликается с моим описанием хорошего дизайна программного обеспечения.

Самое сложное здесь — что сервисы можно развертывать независимо. **Независимо развертываемые** компоненты уже давно используются во множестве различных контекстов, но теперь они становятся частью определения архитектурного стиля и его центральным элементом.

Это ключевая определяющая характеристика микросервисов; здесь нет ничего нового.

Сервисные системы использовали семантический обмен сообщениями по крайней мере с начала 1990-х годов, и все прочие характеристики микросервисов также довольно часто учитывались при создании сервисных систем. Реальная ценность микросервисов заключается в том,

что их можно создавать, тестировать и развертывать независимо от служб, с которыми они работают, и даже от служб, с которыми они взаимодействуют.

Задумайтесь, что это означает. Если мы можем создать сервис и **развернуть его независимо** от других служб, получается, что нам все равно, какой версии эти службы. Это означает, что мы не тестируем наш сервис с другими службами до его релиза. Мы можем сосредоточиться на простом модуле — нашем сервисе.

Он должен быть связным, чтобы не слишком зависеть от других служб или кода. А также очень слабо связан с другими службами, чтобы возможные изменения в любой из них не нарушали работу остальных. В противном случае нам не удастся развернуть сервис, не протестировав его с другими службами перед релизом, и он не будет **независимо развертываемым**.

Такую независимость и следствия из нее обычно упускают команды, которые полагают, что внедряют микросервисы, хотя на самом деле не обеспечивают их достаточного разделения, чтобы развертывать их без предварительного тестирования с другими службами.

Микросервисы — это шаблон масштабирования на уровне организации. В этом его преимущество. Если вам не требуется масштабировать разработку в организации, вам не нужны микросервисы (хотя сервисы могут стать отличной идеей).

Микросервисы позволяют масштабировать разработку, разделяя сервисы и команды, которые работают над этими сервисами¹.

Теперь наши команды имеют возможность работать каждая в своем темпе, независимо от скорости работы другой. Вам не важна версия моего сервиса, потому что ваш сервис достаточно слабо связан с ним.

Такое разделение имеет свою цену. Необходимо строить сервисы так, чтобы они более гибко реагировали на изменения взаимодействующих с ними объектов. Следует применять стратегии, которые изолируют наши сервисы от изменений в других частях системы. Требуется разорвать

¹ В 1967 году Мервин Конвей вывел так называемый закон Конвея, который гласит: «Организации проектируют системы (в широком смысле), которые копируют структуру коммуникаций в этой организации».

связанность разработки, чтобы работать независимо друг от друга. А значит, микросервисный подход может стать неправильным выбором, если вам не нужно расширять команду.

Независимое развертывание, как и все остальное, обходится дорого. Его цена — в необходимости обеспечения лучшей абстракции, лучшей изолированности и более слабой связанности во взаимодействии с другими сервисами. Существует множество способов достижения этой цели, но все они добавляют сложности разрабатываемым системам и увеличивают масштаб задач проектирования.

СНИЖЕНИЕ СВЯЗАННОСТИ МОЖЕТ ОЗНАЧАТЬ БОЛЬШЕ КОДА

Рассмотрим подробнее суть этих издержек, чтобы лучше их понять. Как обычно, за решения приходится платить. Такова природа инженерии; это всегда игра компромиссов. Если мы решим разделить код (снизить связанность), нам почти наверняка придется написать больше кода, по крайней мере в начале работы.

Это одна из частых ошибок, которую совершают многие программисты. Существует мнение, что «меньше кода — хорошо» и «больше кода — плохо», но это не всегда так, и мы разберем ключевой момент, который покажет, что это вообще не так. Вернемся еще раз к примеру, который мы использовали в предыдущих главах. В листинге 13.1 показан код для добавления элемента.

Листинг 13.1. Пример связности (еще раз)

```
def add_to_cart1(self, item):
    self.cart.add(item)

    conn = sqlite3.connect('my_db.sqlite')
    cur = conn.cursor()
    cur.execute('INSERT INTO cart (name, price) values (?, ?)', (item.name, item.price))
    conn.commit()
    conn.close()

    return self.calculate_cart_total();
```

Здесь восемь строк кода, не считая пустых строк. Если мы улучшим этот код, абстрагировав метод, я надеюсь, все согласятся, что он станет лучше, но нам придется добавить еще несколько строк.

В листинге 13.2 уменьшение связанности, лучшая связность и лучшее разделение ответственности стоили двух дополнительных строк кода. Если бы мы пошли дальше — ввели новый модуль или класс, который мы передали в качестве параметра, — мы бы добавили еще несколько строк, чтобы улучшить дизайн.

Листинг 13.2. Ослабление связанности

```
def add_to_cart1(self, item):
    self.cart.add(item)
    self.store_item(item)
    return self.calculate_cart_total()

def store_item(self, item):
    conn = sqlite3.connect('my_db.sqlite')
    cur = conn.cursor()
    cur.execute('INSERT INTO cart (name, price) values (?, ?)', (item.name, item.price))
    conn.commit()
    conn.close()
```

Я знаю, что некоторые программисты отказываются использовать подход, который я описываю в этой книге, а другие не хотят применять автоматизированное тестирование, потому что им приходится больше печатать. Эти ребята оптимизируют не то, что нужно.

Код — средство передачи информации, и в первую очередь другим людям, а не компьютерам.

Наша цель — облегчить жизнь себе и тем, кому придется взаимодействовать с нашим кодом. Это значит, что удобочитаемость не какое-то изысканное, абстрактное свойство кода, которое имеет смысл только для людей, помешанных на стиле и эстетике. Это фундаментальное свойство хорошего кода, и оно оказывает прямое экономическое влияние на его ценность.

Поэтому очень важно позаботиться о том, чтобы код и системы были понятны. Однако важно не только это. Оценивать эффективность, подсчитывая печатаемые символы, нелепо. Неструктурированный связанный код в листинге 13.1 содержит восемь строк. Однако если бы он состоял

из 800 строк, весьма вероятно, что в нем появились бы дублированные фрагменты и он оказался бы избыточным. Управлять сложностью кода важно по многим причинам, и одна из них в том, что это хорошо помогает обнаруживать избыточность и дублирование и избавляться от них.

Чтобы сократить количество кода в реальных системах, необходимо тщательно продумывать его структуру, грамотно проектировать и четко выражать, а не считать, сколько символов напечатано.

Оптимизировать следует мышление, а не набор текста!

СЛАБАЯ СВЯЗАННОСТЬ – НЕ ЕДИНСТВЕННАЯ ВАЖНАЯ ДЕТАЛЬ

У Майкла Нейгарда¹ есть отличная модель для описания связанности. Он делит связанность на ряд категорий (табл. 13.1).

Таблица 13.1. Модель связанности по Нейгарду

ТИП СВЯЗАННОСТИ	ЭФФЕКТ
Операционная	Потребитель не может работать без поставщика
Эволюционная	Изменения в производителях и потребителях необходимо координировать
Семантическая	Изменения вносятся совместно, поскольку концепции являются распределенными
Функциональная	Изменения вносятся совместно, поскольку ответственность является коллективной
Случайная	Изменения вносятся совместно без особой причины (например, критическое изменение API)

Это полезная модель, и дизайн наших систем влияет на все эти типы связанности. Если вы не можете выпустить изменения в продакшен, пока я не закончу свои, значит, мы связаны разработкой. Мы можем решить эту проблему, выбрав определенный дизайн.

¹ Майкл Нейгард – архитектор программного обеспечения и автор книги «Release It. Проектирование и дизайн ПО для тех, кому не все равно». На нескольких конференциях он представлял свою модель связанности: <https://bit.ly/3j2dGIP>.

Если мой сервис не запускается, пока не запустится ваш, то наши сервисы связаны операционно, и опять же, это решается выбором дизайна.

Идентификация этих типов связанности — хороший шаг вперед. Следующий — понять, какой из типов присутствует в вашей разработке и как с ним справляться.

ВЫБОР В ПОЛЬЗУ СЛАБОЙ СВЯЗАННОСТИ

Итак, мы выяснили, что слабая связанность имеет определенную цену и большее количество строк кода может привести к снижению производительности.

СВЯЗАННОСТЬ МОЖЕТ ОКАЗАТЬСЯ СЛИШКОМ СЛАБОЙ

Много лет назад я консультировал крупную финансовую компанию. У них возникла довольно серьезная проблема с производительностью системы управления заказами. Я посетил эту компанию, чтобы посмотреть, смогу ли чем-то помочь.

Архитектор-разработчик очень гордился тем, что они следовали лучшим практикам. Он понимал лучшую практику как ослабление связанных и увеличение абстракции, что, в принципе, неплохо, но они добились этого в том числе построением полностью абстрактной реляционной базы данных. Команда гордилась, что может хранить в базе данных что угодно.

По сути, они создали хранилище «имя – значение» с добавлением пользовательской схемы типа «звезда», где в качестве хранилища использовалась реляционная база данных. Более того, каждый элемент в записи с точки зрения их приложения был отдельным элементом в базе данных вместе со ссылками, которые позволяли получать одноуровневые записи. Это означало высокую степень рекурсии.

Код был очень общим, очень абстрактным, но загрузка почти чего угодно требовала сотен, а иногда и тысяч взаимодействий для извлечения данных из базы, прежде чем с этими данными можно было работать.

Слишком высокая абстракция и чересчур слабая связанность способны навредить!

Поэтому важно знать о таких потенциальных издержках и не увлекаться абстракцией и разделением. Но как я уже говорил, гораздо чаще встречается обратная ситуация — большие комки грязи, а не чересчур абстрактный и разделенный дизайн.

В последнее время я имею дело с очень высокопроизводительными системами, поэтому при их разработке много внимания уделяю производительности. Однако многие ошибочно полагают, что высокопроизводительный код сложен и не может содержать множество вызовов функций или методов. Это устаревшая точка зрения, и от нее нужно избавляться.

Высокую производительность дает простой и эффективный код. Для большинства популярных языков и платформ это код, который легко и, главное, предсказуемо понятен для компиляторов и оборудования. Производительность не оправдание для большого комка грязи!

Тем не менее я принимаю аргумент, что в высокопроизводительных блоках кода нужно тщательно выбирать степень связанности.

Хитрость в том, чтобы правильно провести линии абстракции: в целях обеспечения связности высокопроизводительные части системы следует поместить по одну или по другую сторону от линий абстракции, принимая во внимание, что переход от одного сервиса или одного модуля к другому повлечет дополнительные расходы.

Интерфейсы между службами **предпочитительно проектировать с более слабой связанностью**, поскольку каждая служба скрывает информацию от другой. Такие интерфейсы — очень важные точки в дизайне системы, и их следует проектировать с особой тщательностью. Издержки во время их выполнения будут более высокими, как и количество строк кода. Это допустимый компромисс — он позволяет строить более модульные и гибкие системы.

В ЧЕМ ОТЛИЧИЕ ОТ РАЗДЕЛЕНИЯ ОТВЕТСТВЕННОСТИ?

Может показаться, что **слабая связанность** и **разделение ответственности** похожи, и это, безусловно, так. Тем не менее тесная связанность сочетается с очень хорошим разделением ответственности, а слабая связанность — с низким разделением.

Первое легко представить. Пример — две службы: одна обрабатывает заказы, а другая их хранит. Это хорошее разделение ответственности, но информация, передаваемая между ними, может быть подробной и точной.

В некоторых случаях следует предусмотреть совместное изменение служб. Если одна меняет концепцию заказа, это может нарушить работу второй службы, поэтому они тесно связаны.

Вторую систему, слабосвязанную, но с низким разделением ответственности, вероятно, немного сложнее представить в реальности, чем в теории.

Предположим, две службы управляют двумя отдельными счетами, и с одного счета выполняется денежный перевод на другой. Эти два счета обмениваются информацией асинхронно, через сообщения.

Счет А отправляет сообщение «Счет А дебетован X, кредитовать счет В». Спустя некоторое время аккаунт В видит сообщение и зачисляет средства. Транзакция здесь разделена между двумя службами. Мы хотим, чтобы деньги переместились с одного счета на другой. Это не связное поведение; деньги удаляются с А и добавляются в В, хотя транзакция должна ощущаться целостной.

Реализация описанным способом — очень плохая идея. Она слишком упрощена и обречена на провал. В случае возникновения сбоев при передаче деньги могут исчезнуть.

Определенно, нужно что-то улучшить. Например, инициировать протокол, который будет проверять, что обе части транзакции синхронизированы. Это гарантирует, что если деньги ушли с первого счета, они обязательно поступают на второй, но эти процессы все же слабо связаны — если не семантически, то технически.

DRY – ЭТО СЛИШКОМ ПРОСТО

DRY — сокращение от «Don't Repeat Yourself» («не повторяйся»). Это краткое описание нашего желания иметь единое каноническое представление для поведения каждой части системы. Это хорошо, но не всегда. На практике все гораздо сложнее.

DRY отлично подходит в контексте отдельной функции, службы или **модуля**. Вдобавок можно расширить DRY до репозитория с контролем версий или пайплайна развертывания. Однако это стоит дорого. Иногда даже очень дорого, если DRY применяется между службами или модулями, особенно когда они разрабатываются независимо.

Дело в том, что стоимость одного канонического представления любой идеи во всей системе увеличивает связанность, а стоимость связанности может превышать стоимость дублирования.

В итоге одно уравновешивает другое.

Управление зависимостями — это коварная форма связанности разработкой. Если ваша служба и моя служба совместно используют какую-то библиотеку и вы вынуждены обновлять свою службу, когда я обновляю свою, то наши службы и наши команды связаны разработкой.

Это сильно влияет на способность работать независимо и добиваться прогресса. Вы будете испытывать неудобство, потому что вам придется отложить релиз до перехода на новую версию библиотеки, которую навязала моя команда, или потому что вы работали над другой частью системы и это изменение усложняет вашу задачу.

Преимущество DRY в том, что изменения вносятся только в одном месте, а недостаток — что каждое место, где используется этот код, обладает связанностью.

С инженерной точки зрения полезны несколько инструментов. Самый важный из них — пайплайн развертывания.

При непрерывной доставке **пайплайн развертывания** предназначен для получения четкой и определяющей обратной связи о пригодности системы к релизу. Если пайплайн сообщает, что все хорошо, продукт можно выпускать безопасно. Это неявно обозначает масштаб пайплайна развертывания — независимо развертываемая программная единица.

Итак, если пайплайн сообщает, что все хорошо, можно делать релиз; в этом случае разумно использовать DRY. Как правило, руководствоваться DRY следует внутри пайплайна развертывания, но не между пайплайнами.

Поэтому если вы создаете систему на основе микросервисов, где каждая служба может быть развернута независимо и каждая имеет собственный пайплайн развертывания, не применяйте DRY между микросервисами. **В них не должно быть совместно используемого кода.**

Именно эта интересная мысль в некотором роде и побудила меня написать эту книгу. Мои рекомендации об организации связанности не случайно основаны на понятиях, которые на первый взгляд кажутся далекими друг от друга. От базовой идеи из области компьютерных наук, связности, через

дизайн и архитектуру мы переходим к понятию, которое, очевидно, имеет отношение к способу создания и тестирования продуктов — пайплайну развертывания.

Это часть инженерной философии и подхода, который я пытаюсь описать и популяризировать.

Чтобы соблюдать основной принцип непрерывной доставки — постоянную готовность продукта к релизу, — мы должны учитывать возможность развертывания и объем пайплайна развертывания. Быстро учиться и вызывать состояние сбоя в случае ошибки, о чём я говорил в первой части этой книги, возможно при оптимизации тестируемости систем. Мы получаем возможность создавать более модульный и связанный код с лучшим разделением ответственности и четкими линиями абстракции, сохраняя изолированность и слабую связанность изменений.

Все эти идеи взаимосвязаны. Они дополняют друг друга, и если отнести к ним серьезно и взять за основу в работе, нам удастся быстрее создавать более качественные программные продукты.

Что бы ни представляла собой программная инженерия, если она не помогает создавать лучший продукт быстрее, ее нельзя называть инженерией.

АСИНХРОННОСТЬ КАК ИНСТРУМЕНТ СЛАБОЙ СВЯЗАННОСТИ

В предыдущей главе обсуждалась утечка абстракций. Одна из негерметичных абстракций — синхронные вычисления за границами процессов.

Как только мы устанавливаем границу, какой бы ни была ее природа, синхронность становится иллюзией, и за нее приходится платить.

Негерметичность этой абстракции больше всего вредит в распределенных вычислениях. Если служба A обменивается данными со службой B, нужно найти все места, в которых этот обмен может быть нарушен, если службы будут разделены сетью.

Иллюзия, дырявая абстракция синхронизации, может существовать, но только пока не произойдет сбой, — а он произойдет. На рис. 13.1 показаны точки, где был нарушен обмен данными в распределенной среде.

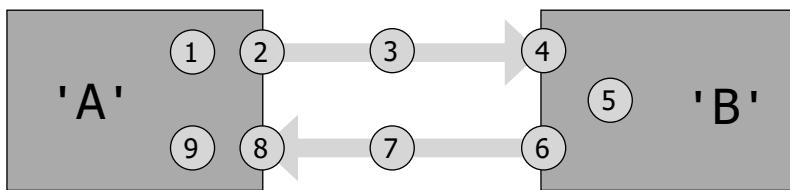


Рис. 13.1. Точки отказа в синхронном обмене данными

1. Возможна ошибка в **A**.
2. **A** может не установить соединение с сетью.
3. Сообщение может быть потеряно при передаче.
4. **B** может не установить соединение с сетью.
5. Возможна ошибка в **B**.
6. Соединение с сетью может быть прервано до того, как **B** отправит ответ.
7. Ответ может быть потерян при передаче.
8. **A** может потерять соединение до того, как получит ответ.
9. Возможна ошибка в обработке ответа **A**.

Помимо 1 и 9, каждая из перечисленных точек отказа является утечкой в абстракции синхронных коммуникаций. Каждая усложняет работу с ошибками. Почти все эти ошибки приводят к асинхронизации **A** и **B**, что усугубляет сложность. Только некоторые из этих отказов отправителю, **A**, удастся обнаружить.

Теперь представьте, что **A** и **B** связаны синхронным поведением на уровне бизнеса. Когда возникает проблема, например обрыв соединения или пропущенное сообщение, такой технический сбой прерывает коммуникацию на бизнес-уровне.

Утечки можно уменьшить, если разобраться, что происходит. Сети — на самом деле асинхронные механизмы передачи сообщений; коммуникация в реальном мире асинхронна.

Когда мы беседуем, мозг не застывает в ожидании ответа после того, как один из собеседников задал другому вопрос; мозг продолжает работать. Утечка в лучшей абстракции, приближенной к реальности, оказывается менее значительной.

Не вдаваясь в конкретные подробности проектирования, отмечу, что следует считать границы процессов асинхронными и организовывать обмен данными между распределенными службами и модулями только через асинхронные события. В сложных распределенных системах это значительно снижает влияние утечек абстракции и уменьшает связанность с базовой случайной сложностью.

Представьте влияние надежной асинхронной системы обмена сообщениями на список точек отказа на рис. 13.1. В ней могут происходить все те же сбои, но если служба А отправляет только асинхронные сообщения, а через некоторое время получает только новое асинхронное сообщение, то после шага 2 ни один из них ее не затронет. Даже когда в центр обработки данных, где работает служба В, попадет метеорит, мы сможем восстановить ЦОД, повторно развернуть копию службы В и переотправить сообщение от службы А. Хоть и с опозданием, все процессы продолжат работу, как если бы обмен сообщениями занял всего несколько микросекунд.

Эта глава посвящена связанности, а не асинхронному программированию или проектированию. Я не убеждаю вас в достоинствах асинхронного программирования, хотя их много, а на приведенном примере показываю, что благодаря грамотному ослаблению связанности, в данном случае между случайной сложностью сетей и удаленных коммуникаций и необходимой сложностью бизнес-функций служб, можно написать один фрагмент кода, который будет работать и в случае нормальной работы системы, и в случае сбоя. Это грамотный инженерный способ решения целого пула проблем.

ПРОЕКТИРОВАНИЕ СЛАБОЙ СВЯЗАННОСТИ

Повторюсь, что стремление обеспечить тестируемость кода способствует созданию более слабо связанных систем. Если код сложно тестировать, обычно причина в том, что степень связанности не соответствует задаче.

А значит, необходимо учесть обратную связь и внести изменения, чтобы ослабить связанность, упростить тестирование и в конечном итоге получить более качественный дизайн. Улучшение качества кода и дизайна — минимум того, что обязан обеспечивать настоящий инженерный подход к разработке.

СЛАБАЯ СВЯЗАННОСТЬ В ОРГАНИЗАЦИЯХ

Я считаю связанность важнейшим аспектом разработки. Именно она позволяет делать сложные программные продукты.

Большинство людей способны научиться писать простую программу за несколько часов. Люди очень хорошо разбираются в языках, даже в таких странных, грамматически ограниченных и абстрактных, как языки программирования. Проблема не в этом. Легкость, с которой большинство усваивает несколько понятий, позволяющих им написать пару строк кода, ведет к тому, что новички начинают переоценивать свои способности.

Профессиональное программирование — это не перевод инструкций с человеческого языка на язык программирования. Это могут делать даже машины¹. Профессиональное программирование — это решение задач, а код — инструмент для формулировки таких решений.

Программирование требует много знаний, но вы можете уже на этапе обучения приступить к работе и, решая простые задачи самостоятельно, добиться хорошего прогресса. Сложность возникает по мере расширения и усложнения систем, которые мы создаем, и команд. Именно тогда начинает сказываться влияние связанности.

Как я уже говорил, речь не только о коде, но и о корпоративных связях в организациях, где создают код. Связанность разработки — частая и ресурсозатратная проблема, характерная для крупных компаний.

Если мы будем решать ее, интегрируя работу, то неизбежно возникнут издержки. Свою книгу «*Continuous Delivery*» я, по сути, посвятил стратегиям эффективного управления связанностью в компаниях.

Я видел много крупных компаний с плохой организационной связанностью. В таких компаниях практически невозможно выпустить какое-либо изменение, потому что они годами игнорируют затраты на связанность и теперь для внедрения малейшего изменения требуются десятки или сотни людей, координирующих работу.

¹ ГПТЗ — это система машинного обучения, построенная на массиве текстов интернета. Получая инструкции на английском языке, она может кодировать простые приложения. См. <https://bit.ly/3ugOpzQ>.

Есть только две разумные стратегии: скоординированный либо распределенный подход. У каждого имеются плюсы и минусы. Это часть природы инженерии.

Важно отметить, что оба подхода сильно зависят от эффективности обратной связи, поэтому непрерывная доставка так важна. Она основана на высокой степени оптимизации циклов обратной связи в процессе разработки, когда обратная связь о качестве продукта поступает фактически непрерывно.

Если вам нужна согласованность большого и сложного фрагмента программы, используйте скоординированный подход. Он предусматривает совместное хранение, создание, тестирование и развертывание.

Так вы получите самое четкое и точное представление, но придется работать быстро и эффективно. Обычно я советую получать обратную связь несколько раз в день. Но это может потребовать значительных затрат времени, усилий и технологий.

Подобный подход не мешает нескольким командам работать над одной системой и не означает, что системы, создаваемые таким образом, имеют тесную связанность. Здесь мы говорим о том, какую область готового релиза необходимо оценивать. В данном случае — систему целиком.

Там, где отдельные команды работают полунезависимо, они координируют свои действия с помощью общей кодовой базы и пайплайна развертывания непрерывной доставки для всей системы.

Это позволяет командам, создающим код, сервисы или модули, которые обладают наиболее тесной связанностью, добиться хорошего прогресса с минимальными затратами на обратную связь, но, повторюсь, нужно потрудиться, чтобы обеспечить высокую скорость ее получения.

Сейчас более популярен распределенный подход, использующий микросервисы. В таком случае принятие решений намеренно распределено. Команды микросервисов работают независимо друг от друга, каждый сервис развертывается независимо, и между командами отсутствуют прямые затраты на координацию. Однако существуют косвенные затраты, связанные с дизайном.

Чтобы ослабить организационную связанность, важно избежать совместного тестирования сервисов. Если сервисы можно развертывать

независимо, значит, и тестировать их можно независимо. Кроме того, как определить возможность развертывания без тестирования? Если мы протестируем два сервиса вместе и выясним, что версия 4 одного из них работает с версией 6 другого, сможем ли мы выпустить версии 4 и 17 без тестирования? Если нет, значит, они не независимы.

Микросервисы — самая масштабируемая стратегия разработки ПО. Она позволяет привлекать к работе любое количество команд и разработчиков, по крайней мере столько, сколько вы сможете найти и оплатить.

Стоимость ее заключается в отказе от координации или хотя бы приведении ее к самой простой, общей форме. Вы можете предложить централизованное управление, но не способны обеспечить его работу, потому что это повлечет затраты на координацию.

Организации, грамотно использующие микросервисы, сознательно ослабляют контроль; на самом деле использовать микросервисы практически не имеет смысла, если его не ослаблять.

Оба этих подхода — единственные, которые действительно имеют смысл, — основаны на разных стратегиях управления связанностью команд. Когда связанность сильная, вы повышаете частоту проверки ошибок; когда же она слабая, вы не проверяете их вообще, по крайней мере перед релизом.

Любая стратегия сопряжена с издержками, и золотой середины нет, несмотря на то что многие организации стремятся ее достичь.

ИТОГИ

Связанность — это монстр разработки ПО. Как только сложность продукта превышает базовую, то залогом того, будет ли продукт успешным или провалится, становится надлежащий уровень связанности или, по крайней мере, работа над управлением уровнем связанности, который вы установили в системе.

Если команды разработчиков добиваются результата без координации своих действий, отчет State of DevOps отмечает, что вероятность регулярной поставки высококачественного кода становится выше.

Существуют три способа добиться этого. Можно работать с более связанным кодом и системами, но благодаря непрерывной интеграции и непрерывной доставке получать достаточно быструю обратную связь, чтобы немедленно выявлять проблемы. Можно разрабатывать менее связанные системы, чтобы изменять их безопасно и уверенно, без необходимости вносить изменения в других местах. Или работать с согласованными и фиксированными интерфейсами, не предназначенными для изменений. Это единственные доступные стратегии.

Игнорировать затраты на связанность как в программном продукте, так и во всей организации можно только на свой страх и риск.

IV

ИНСТРУМЕНТЫ ПРОГРАММНОЙ ИНЖЕНЕРИИ

ГЛАВА 14

ИНСТРУМЕНТЫ ИНЖЕНЕРНОЙ ДИСЦИПЛИНЫ

Когда я размышляю, какой должна быть настоящая инженерия для разработки ПО, я мало думаю о конкретных инструментах, языках программирования, процессах или методах построения диаграмм. Скорее я думаю о результатах.

Любой подход, достойный называться **инженерным**, должен строиться на потребности учиться, исследовать и экспериментировать. Самое главное: если он не помогает создавать лучшие продукты быстрее, то это скорее мода, а не дело. Инженерия — это то, что работает; если это не так, мы будем менять подход, пока он не заработает.

Если я не думаю о конкретных инструментах, это не значит, что их нет. В книге я показываю, что в разработке используются некоторые универсальные интеллектуальные инструменты, позволяющие создавать продукты гораздо быстрее и лучше. Не все идеи одинаковы; есть просто плохие, и мы должны уметь выявлять их и отбрасывать.

В этой главе мы еще раз вернемся к некоторым идеям, которые обсуждали ранее. Эти идеи — базовые в разработке ПО. Даже если вы пропустите все остальное, о чем говорилось в книге, и возьмете на вооружение только эти базовые принципы, вы обнаружите, что ваши результаты значительно улучшатся. А со временем вы откроете для себя и все остальные идеи, которые я описал, — потому что они логически вытекают из базовых.

ЧТО ТАКОЕ ПРОГРАММНАЯ РАЗРАБОТКА

Разработка ПО требует, безусловно, большего, чем просто знания синтаксиса языка и его библиотек. Идеи, которые мы выражаем, во много раз важнее, чем инструменты, которые мы используем для их выражения. В конце концов, нам платят за решение задач, а не за использование инструментов.

В чем смысл создания ПО (для любых целей), если мы не знаем, будет ли оно работать?

Изучать написанный код, не запуская его, довольно опрометчиво. Люди так не работают. Даже в таких свободно интерпретируемых языках, как разговорная речь, мы все время делаем ошибки. С вами случалось когда-нибудь такое: пишете, например, электронное письмо, отправляете его без проверки и после этого замечаете грамматические или орфографические ошибки — но уже слишком поздно?

Мои редакторы и я очень старались исправить все ошибки в этой книге, но я уверен, что несколько вы все же обнаружили. Люди склонны делать ошибки. Мы плохо проверяем выполненную работу, потому что часто видим то, что хотим увидеть, а не то, что есть на самом деле. Это не только критика свойственной человеку лени, сколько признание ограничения его биологических возможностей. Мы часто делаем поспешные выводы — очень полезное качество для наших предков во враждебной среде обитания.

Разработка ПО не терпит ошибок; вычитки и код-ревью недостаточно. Нужно тестировать код, чтобы убедиться, что он работает. Тестирование можно выполнять по-разному, но в любом случае — запускаем ли мы код, просто чтобы посмотреть, что происходит, или в отладчике, чтобы увидеть, что и как меняется, или запускаем набор сценариев разработки через поведение (BDD), — мы пытаемся получить обратную связь о работе продукта.

Как отмечалось в главе 5, чтобы обратная связь создавала ценность, она должна быть быстрой и эффективной.

Итак, если необходимо тестирование, единственный важный вопрос — как сделать его максимально эффективным и информативным?

Можно протестировать весь код целиком в конце работы. Или просто выпустить готовый продукт и положиться на пользователей, которые бесплатно его протестируют? Не самый лучший путь к успеху! Плохая работа обернется убытками; вот почему так важен инженерный подход.

Вместо того чтобы скрещивать пальцы и надеяться, что код будет работать, лучше испытать его перед релизом. Сделать это можно по-разному.

Если мы будем ждать готовности всего продукта, мы явно не получим качественную и своевременную **обратную связь**. Мы, скорее всего, забудем некоторые мелочи, поэтому тестирование окажется неглубоким. Кроме того, оно будет весьма изнурительным.

Многие организации предпочитают нанять специальных людей, которые будут выполнять эту изнурительную работу. Мы возвращаемся в исходную точку — опрометчиво полагаем, что наш продукт работает, и рассчитываем, что кто-то другой выяснит, что это не так. Это, безусловно, шаг вперед по сравнению с ожиданием реакции от пользователей, но все еще не то, что нужно.

Такие действия, как привлечение к работе дополнительных сотрудников, не повышают скорость или качество обратной связи. Я не критикую способности людей: все мы слишком медлительны, слишком непостоянны в своих действиях и дорого обходимся, чтобы конкурировать с автоматизированным сбором обратной связи.

Мы получим информацию слишком поздно и в ходе разработки не будем иметь ни малейшего представления о том, насколько хорошо или плох наш продукт. Мы упустим ценные возможности обучения, которыми могли бы воспользоваться, если бы обратная связь была своевременной. Вместо этого мы дожидаемся окончания всей работы, а затем получаем некачественную, несвоевременную обратную связь от людей, которые, даже будучи квалифицированными и прилежными, просто не знают механизмов работы системы, построенной без учета ее тестируемости.

Вполне возможно, что в конечном итоге качество продукта нас приятно удивит, но подозреваю, что, вероятнее, нас шокируют наши глупые ошибки. Помните: мы до сих пор не провели никаких тестов продукта, даже не запустили его.

Наверняка вы уже поняли, что я считаю это неприемлемым.

Это плохая идея, поэтому мы должны добавить проверку на более ранних стадиях рабочего процесса. Узнавать о том, что пользователи не могут войти в систему, а наша классная новая функция на самом деле разрушает диск, поздно, когда продукт готов к релизу.

Так что к тестированию следует подходить с умом. Как организовать работу, чтобы минимизировать ее объем, но в ходе разработки получать максимальную информацию о поведении системы?

В части II мы говорили об оптимизации для обучения. Так что же мы хотим узнать и как это сделать наиболее эффективно?

Когда мы приступаем к написанию кода, нам нужно ответить на четыре вопроса:

- ту ли задачу мы решаем;
- ведет ли себя система так, как мы думаем;
- каково качество нашей работы;
- эффективно ли мы работаем.

Это, безусловно, сложные вопросы, но, по сути, это все, что нас должно интересовать.

ТЕСТИРУЕМОСТЬ КАК ИНСТРУМЕНТ

Программный продукт должен быть написан так, чтобы его было легко тестировать. Я уже рассказывал (в главе 11), как **разделение ответственности и внедрение зависимостей** делают код более тестируемым. На самом деле трудно представить тестируемый код, который не обладает модульностью, связностью, хорошим **разделением ответственности** и скрытием информации. Если ему свойственны все эти признаки, он также неизбежно обладает **оптимальной связанностью**.

Рассмотрим эффект улучшения **тестируемости** кода на простом примере. Здесь я всего лишь хочу иметь возможность тестировать код. В листинге 14.1 показан простой класс `Car`.

Листинг 14.1. Пример простого класса Car

```
public class Car {
    private final Engine engine = new PetrolEngine();

    public void start() {
        putIntoPark();
        applyBrakes();
        this.engine.start();
    }

    private void applyBrakes() {
    }

    private void putIntoPark() {
    }
}
```

В этом классе есть двигатель, `PetrolEngine`. Когда вы «заводите машину», он выполняет несколько действий. Двигатель останавливает машину (`PutIntoPark`), нажимает тормоза (`applyBrakes`) и запускает `Engine`. Все выглядит нормально; многие написали бы похожий код.

Теперь протестируем его (листинг 14.2).

Листинг 14.2. Тест простого класса Car

```
@Test
public void shouldStartCarEngine() {
    Car car = new Car();
    car.start();
    // Нечего подтвердить!!
}
```

Сразу видна проблема. Мы сможем выполнить тест `Car`, только если нарушим инкапсуляцию машины и сделаем приватное поле `engine` общедоступным или найдем еще какую-нибудь лазейку, чтобы считывать приватную переменную (кстати, оба варианта никуда не годятся). Этот код просто нетестируемый, потому что эффект «заведения машины» не виден.

Проблема в том, что мы попали в некую конечную точку. Наша последняя точка доступа к `Car` — вызов метода `start`. После этого внутренняя работа становится невидимой. Чтобы протестировать `Car`, нам нужно каким-то образом разрешить доступ, а это нетривиальная задача тестирования. Нам нужно увидеть двигатель.

В данном случае решение — добавить точку измерения путем **внедрения зависимостей**. Вот лучший пример: вместо того чтобы скрывать Engine, мы передадим Engine, который хотим использовать, в BetterCar. В листинге 14.3 показан BetterCar, а в листинге 14.4 — его тест.

Листинг 14.3. BetterCar

```
public class BetterCar {  
    private final Engine engine;  
  
    public BetterCar(Engine engine) {  
        this.engine = engine;  
    }  
  
    public void start() {  
        putIntoPark();  
        applyBrakes();  
        this.engine.start();  
    }  
  
    private void applyBrakes() {  
    }  
  
    private void putIntoPark() {  
    }
```

В листинге 14.3 внедряется Engine. Этот простой шаг полностью меняет связанность с PetrolEngine; теперь наш класс более абстрактен, потому что он оперирует Engine, а не PetrolEngine. Разделение ответственности и связность улучшились, потому что BetterCar больше не связан с созданием PetrolEngine.

В листинге 14.4 представлен тест BetterCar.

Листинг 14.4. Тест BetterCar

```
@Test  
public void shouldStartBetterCarEngine() {  
    FakeEngine engine = new FakeEngine();  
    BetterCar car = new BetterCar(engine);  
    car.start();  
    assertTrue(engine.startedSuccessfully());  
}
```

Этот BetterCarTest использует FakeEngine, показанный для полноты в листинге 14.5.

Листинг 14.5. FakeEngine для тестирования BetterCar

```
public class FakeEngine implements Engine {
    private boolean started = false;

    @Override
    public void start() {
        started = true;
    }

    public boolean startedSuccessfully() {
        return started;
    }
}
```

FakeEngine необходим только для записи вызова `start`¹.

Это простое изменение улучшило код и сделало его тестируемым. Помимо того что код теперь модульный и связный, он стал проще и практичнее.

Тестируемость добавила коду большую гибкость. Создать BetterCar с PetrolEngine просто, но так же просто создать BetterCar с ElectricEngine, или FakeEngine, или даже JetEngine, если кому-то это вдруг взбредет в голову. BetterCar — лучший код, и он стал таким потому, что мы стремились к более простому тестированию.

Улучшение **тестируемости** приводит к созданию кода более высокого качества. Это, конечно, не панацея. Если вы плохой программист, ваш код может так и остаться плохим, но он станет лучше, если вы постараетесь обеспечить его тестируемость. Если вы хороший программист, то сделаете код еще лучше, если он станет тестируемым.

ТОЧКИ ИЗМЕРЕНИЯ

В нашем примере FakeEngine использует еще одну важную идею — **точки измерения**. Если мы хотим, чтобы код был тестируемым, нам нужно иметь возможность управлять переменными — вводить именно ту информацию, которая требуется, и только ее. Тестирование заключается в том, что мы инициируем некоторое поведение, чтобы получить видимые и измеримые результаты.

¹ В реальном teste лучше использовать библиотеку Mocking, а не писать этот код самостоятельно. Я добавил код FakeEngine для наглядности.

Именно таким образом можно обеспечить тестируемость. Системы необходимо проектировать так, чтобы они содержали множество точек измерения, где можно исследовать поведение системы, не нарушая ее целостности. Эти точки имеют разный вид в зависимости от характера компонента и степени тестируемости.

Для подробного тестирования модулей мы используем параметры и возвращаемые значения функций или методов, а также **внедрение зависимостей**, как показано в листинге 14.4.

Для более общего тестирования на уровне системы мы симулируем внешние зависимости, чтобы вставить **зонды точек измерения** в систему и вводить исходные данные или собирать результаты тестов, как я рассказал в главе 9.

СЛОЖНОСТИ С ОБЕСПЕЧЕНИЕМ ТЕСТИРУЕМОСТИ

Многим командам не удается добиться такой тестируемости, которую я здесь описываю, и виной тому две основные причины. Одна из них — техническая сложность; другая относится к сфере культуры разработки.

Как мы уже выяснили, любая форма тестирования требует грамотного размещения точек измерения. В большей части кода проблем не возникает. С помощью таких методов, как внедрение зависимостей и модульность, мы делаем код тестируемым; но это становится сложнее на границах системы, в точках, в которых она взаимодействует с реальным миром (или по крайней мере с его компьютерной моделью).

Если задача кода — запись информации на диск, рисование изображений на экране или управление другим аппаратным устройством либо взаимодействие с ним, то такой код трудно тестировать. Как внедрить в эту часть системы тестовый код для ввода данных или для сбора результатов теста?

Очевидное решение — спроектировать систему так, чтобы вынести граничные точки кода на поля и минимизировать их сложность. То есть ослабить связность основной системы с этими точками. Это, в свою очередь, снижает зависимость от сторонних программных элементов, делает код более гибким и требует небольших дополнительных усилий.

Мы создаем подходящую абстракцию, которая представляет взаимодействие в граничной точке, пишем тесты, которые оценивают взаимодействие системы с симулятором этой абстракции, а затем создаем простой код для перевода абстракции в реальное взаимодействие с этими границами. Это долгий способ добавления уровня косвенности.

В листинге 14.6 показан простой пример кода отображения неких данных. Можно создать робота с камерой для записи вывода на экран, но это лишнее. Проще абстрагировать действие вывода на экран, внедряя часть кода, которая обеспечит отображение текста.

Листинг 14.6. Отображение данных

```
public interface Display
{
    void show(String stringToShow);
}

public class MyClassWithStuffToDisplay
{
    private final Display display;

    public MyClassWithStuffToDisplay(Display display)
    {
        this.display = display;
    }

    public void showStuff(String stuff)
    {
        display.show(stuff);
    }
}
```

Хороший побочный эффект абстрагирования отображения данных в том, что класс с отображаемыми данными теперь отделен от любого реального устройства вывода, по крайней мере за пределами этой абстракции. Очевидно, это также означает, что теперь код можно тестировать без реального *Display*. Пример такого теста — в листинге 14.7.

Листинг 14.7. Тестирование отображаемых данных

```
@Test
public void shouldDisplayOutput() throws Exception
{
    Display display = mock(Display.class);
    MyClassWithStuffToDisplay displayable = new MyClassWithStuffToDisplay
(display);
```

```

    displayable.showStuff("My stuff");

    verify(display).show(eq("My stuff"));
}

```

Наконец, можно создать конкретную реализацию `Display`. В простом примере в листинге 14.8 это `ConsoleDisplay`, но при необходимости его можно заменить такими вариантами, как `LaserDisplayBoard`, `MindImprintDisplay`, `3DGameEngineDisplay` и так далее.

Листинг 14.8. Отображение данных

```

public class ConsoleDisplay implements Display
{
    @Override
    public void show(String stringToDisplay)
    {
        System.out.println(stringToDisplay);
    }
}

```

Примеры 14.5 — 14.8 довольно просты; абстракция явно должна быть сложнее, если в граничной точке мы будем взаимодействовать с более сложной технологией, но принцип все тот же.

ГРАНИЧНОЕ ТЕСТИРОВАНИЕ

В одном из проектов, где я участвовал, мы абстрагировали веб-DOM¹ таким образом, чтобы сделать логическую единицу веб-страницы юнит-тестируемой.

Сейчас известны более оптимальные техники, но в то время было сложно проводить юнит-тестирование веб-приложений без реального браузера. Мы не хотели замедлять тестирование из-за необходимости запускать экземпляр браузера для каждого теста, поэтому решили изменить способ написания пользовательского интерфейса.

Мы написали библиотеку UI-компонентов, которые «стояли перед DOM» (портов и адаптеров для DOM). Поэтому если нам требовалась таблица, мы создавали таблицу JavaScript, пользуясь собственной DOM-фабрикой. Тем самым во время выполнения мы получали тонкий фасадный объект — таблицу для использования, а во время тестирования — заглушку, благодаря которой не требовался реальный браузер или DOM.

¹ DOM – Document Object Module (объектная модель документа).

Вы всегда можете повторить это решение. Вопрос лишь в том, насколько проста или сложна технология, которую вы пытаетесь абстрагировать, и насколько это важно, чтобы тратить усилия.

Для граничных точек затраченные усилия почти всегда того стоят. Иногда, например при тестировании веб-интерфейса или мобильного приложения, сделать эту работу за вас может кто-то другой, но граничное юнит-тестирование следует проводить именно так.

Проблема с этим способом, да и вообще с любым способом решения подобной задачи, лежит в сфере культуры разработки. Если с самого начала разработки заботиться о testability системы, все будет довольно просто.

Сложнее, если код создавали без учета testability или разработчики не считали ее важной. Столкновение этих двух корпоративных культур — серьезная проблема.

С кодом, вероятно, справиться проще всего, хотя «проще» вовсе не значит «просто». Всегда можно добавить свои собственные абстракции, даже если они нетерметичны. Или включить негибкий «пограничный» код в тест, если это действительно нужно. Это не лучший компромисс, но иногда он работает.

Самое сложное — люди. Я не настолько самонадеян, чтобы утверждать, что ни одна команда, которая действительно соблюдала принцип TDD «сначала пишем тест, а потом — код», не столкнулась с неработающим кодом, но я таких не встречал.

На самом деле многие команды жаловались мне: «Мы пробовали TDD, и это не сработало», — но все они имели в виду попытки писать юнит-тесты после написания кода. А это далеко не одно и то же.

Суть в том, что TDD помогает разрабатывать **тестируемый код**, а юнит-тестирование — нет. Юнит-тестирование, которое проводят после того, как код написан, помогает слаживать углы, нарушать инкапсуляцию и тщательно соединять тест с имеющимся кодом.

TDD — краеугольный камень инженерного подхода к разработке. Я не знаю другого метода, столь же эффективного при проектировании, — об этом я и рассказываю в этой книге.

Самый сильный аргумент против использования TDD, который я иногда слышу: TDD ставит под угрозу качество проектирования и ограничивает возможность внесения изменений, поскольку тесты связываются с кодом. Но я никогда не встречал такого в кодовых базах, созданных по принципу TDD «сначала тест». Зато эта угроза и это ограничение вполне обычны — я бы сказал, неизбежны — в подходе «сначала код, потом юнит-тест». Так что я подозреваю, что если люди говорят, что TDD не работает, они на самом деле и не пытались его применять. И пусть это утверждение верно не в 100% случаев, но в большинстве, и поэтому его можно считать очень близким к истине.

Критический подход к качеству разработки мне особенно близок, потому что, как вы, наверное, уже убедились из этой книги, я уделяю качеству очень большое внимание.

Я бы лукавил, если бы отрицал свои навыки разработки, проектирования и TDD. Они довольно хороши, и я могу только догадываться почему. Конечно, у меня есть опыт. Возможно, имеет значение и какой-то природный талант, но гораздо важнее, что у меня есть несколько полезных привычек, которые помогают избегать сложностей. Из всех техник, которые я знаю, TDD позволяет получать самую четкую обратную связь о качестве дизайна по мере его развития, и в своей работе я использую именно эту методологию и рекомендую ее другим.

КАК УЛУЧШИТЬ ТЕСТИРУЕМОСТЬ

Часть II посвящена важности оптимизации для обучения. Не в академическом, а в более узком, практическом смысле повседневной работы. Необходимо действовать итеративно, добавляя тест к конкретному фрагменту, с которым мы работаем в данный момент. Тест должен обеспечивать быструю, эффективную и четкую обратную связь, чтобы каждые несколько минут мы могли узнавать, что код ведет себя именно так, как планировалось.

Необходимо разделить систему на части — это позволит нам ясно интерпретировать обратную связь. Следует постепенно создавать небольшие отдельные фрагменты кода, ограничивая область оценки, чтобы по мере работы мы понимали, что происходит.

Стоит представлять каждый тестовый кейс как небольшой эксперимент, который предсказывает и проверяет желаемое поведение кода. Мы пишем тест, чтобы сформулировать, как должна вести себя программа. Мы предполагаем возможные сбои, прежде чем запустить тест, дабы убедиться, что он действительно проверяет то, что мы хотим проверить. Затем мы пишем код, для которого тест завершается успехом, и используем эту стабильную успешную комбинацию кода и теста в качестве платформы для проверки дизайна и внесения небольших безопасных изменений, сохраняющих поведение. Таким образом мы оптимизируем качество кода и тестов.

Это способствует более глубокому пониманию системы, поскольку мы погрузимся в нее не только до уровня «Проходит ли она тест?». Обеспечение тестируемости кода помогает добиваться более качественного результата.

У нас нет набора инструментов, которые способны сделать все это за нас. Игнорируя тестируемость, мы подвергаемся риску. Слишком многие независимые разработчики и команды не уделяют должного внимания тестируемости и в результате работают медленнее и хуже, чем могут и должны.

Если тест писать сложно, значит, вы создаете плохой код и его нужно улучшать.

Тестируемость систем фрактальна. Мы можем наблюдать за ней и использовать как на уровне всей организации, так и на уровне нескольких строк кода, и следует помнить, что это один из самых мощных из доступных нам инструментов.

На уровне мелких модулей — функций и классов — самым важным аспектом тестируемости, на который нужно обратить особое внимание, являются точки измерения. От них зависит легкость, с которой удается перевести код в определенное состояние, и легкость, с которой можно наблюдать и оценивать результаты его поведения.

На более системном и мультисистемном уровнях основное внимание необходимо уделять области оценки и тестирования. Точки измерения по-прежнему важны, но область оценки приобретает неменьшую важность.

РАЗВЕРТЫВАЕМОСТЬ

В книге «*Continuous Delivery*» мы описали подход к разработке, основанный на том, чтобы поддерживать продукт в постоянной готовности к релизу. Необходимо оценивать возможность релиза продукта после каждого небольшого изменения, получая обратную связь несколько раз в день.

Для этого используется механизм, называемый *пайплайном развертывания*. Он предназначен для определения готовности продукта к релизу, насколько это практически возможно, за счет высокого уровня автоматизации.

Что означает «готовый к релизу»? Это всегда зависит от контекста.

Требуется убедиться, что код делает то, чего от него ожидают разработчики, а затем — что он делает то, чего от него ожидают пользователи. После этого необходимо понять, достаточно ли продукт быстрый, безопасный, отказоустойчивый и соответствует ли он применимым нормам.

Все это задачи пайплайна развертывания. До сих пор я описывал пайплайн развертывания с точки зрения готовности продукта к релизу, но я хочу кое-что пояснить, прежде чем двигаться дальше.

На самом деле, говоря о пайплайнах развертывания, необходимо различать пайплайны **выпуска** и пайплайны **развертывания**. Это тонкий момент, но в разработке важно отделять готовность к **развертыванию** изменения в рабочей среде от выпуска функции для пользователей.

Непрерывная доставка предполагает свободу создания новых функций в ходе серии развертываний. Здесь я перехожу от **готовности к релизу**, подразумевающей полноту функций и ценность для пользователей, к **возможности развертывания**, что означает, что продукт можно безопасно выпускать на продакшен, даже если некоторые функции еще не готовы к использованию и скрыты.

Таким образом, **возможность развертывания** системы предполагает соблюдение ряда условий; программный модуль можно развертывать, и он обладает всеми признаками готовности к релизу, значимыми для системы: достаточной скоростью, безопасностью, отказоустойчивостью, работоспособностью и так далее.

Идея возможности развертывания хорошо себя зарекомендовала на системном и архитектурном уровнях. Если пайплайн развертывания сообщает, что система может быть развернута, она готова к развертыванию в рабочей среде.

Многие не до конца понимают функцию пайплайна непрерывной доставки, но она именно такова. Если пайплайн развертывания сообщает, что изменение хорошее, значит, больше не требуется тестирование, утверждение и тесты интеграции с другими частями системы. Проводить развертывание в рабочей среде необязательно, но мы можем это сделать при желании, так как изменение одобрено пайплайном и готово к релизу.

Согласно этому представлению, развертываемость определяется как состояние, для которого больше делать ничего не нужно; для достижения такого состояния необходимо обеспечить надлежащую модульность, связность, разделение ответственности, связанность и сокрытие информации на уровне **развертываемых единиц ПО**.

Областью оценки всегда остается **независимая развертываемая единица программного обеспечения**. Если изменение невозможно выпустить без доработок, то единица оценки, область пайплайна развертывания, неверна.

Существует несколько способов того, как с этим работать. Можно включить в область оценки (в область пайплайна развертывания) всю систему либо разделить систему на независимо развертываемые программные модули — остальное не имеет смысла.

Мы можем организовать сборку компонентов системы в разных местах, из отдельных репозиториев, но область оценки определяется требованиями развертываемости. Так что если мы выбираем этот путь, но при этом понимаем, что перед выпуском необходимо оценить все компоненты вместе, то областью оценки, областью пайплайна развертывания, по-прежнему останется вся система. Это важно, потому что какой бы быстрой ни была оценка небольшого фрагмента системы, в действительности имеет значение время, необходимое для оценки развертывания изменения. Так что оптимизировать следует именно эту область.

Таким образом, обеспечение возможности развертывания — чрезвычайно важная задача разработки. Работа в этом ключе помогает сосредоточиться на решаемой задаче. Как добиться своевременной обратной связи, чтобы направить рабочие усилия в нужное русло?

СКОРОСТЬ

Так мы приходим к **скорости**. Как я уже говорил в части II, скорость и качество обратной связи, которую мы получаем в ходе разработки, необходимы, чтобы оптимизировать обучение. В главе 3 мы обсудили важность метрик и рассмотрели **стабильность** и **пропускную способность**. Пропускная способность как мера эффективности нашего процесса разработки явно зависит от скорости.

Когда я помогаю внедрять непрерывную доставку, я советую направлять усилия на то, чтобы сократить время получения обратной связи.

Обычно стоит оптимизировать разработку так, чтобы получать готовый к релизу, разверываемый программный модуль финального качества несколько раз в день: чем чаще, тем лучше. Оптимально, чтобы развертывание изменений в рабочей среде занимало не более часа после коммита изменений.

Цель непростая, но подумайте, что для нее потребуется. Команды не могут быть слишком большими, потому что издержки коммуникации очень замедляют работу. По этой же причине они также не могут быть и достаточно разделенными. Чтобы получить готовый результат менее чем за час, необходимо обеспечить налаженный процесс автоматизированного тестирования, механизмы обратной связи, такие как непрерывная интеграция и непрерывная доставка, хорошую поддерживающую архитектуру, оценку независимо развертываемых модулей ПО и многое другое.

Если вы выбираете итеративный, экспериментальный подход, только чтобы повысить скорость обратной связи, он становится своего рода фитнес-функцией для всех принципов agile, lean, непрерывной доставки и DevOps.

Внимание к скорости и обратной связи неумолимо приведет вас к применению этих принципов. Это гораздо более мощный и измеримый способ добиться лучших результатов, чем разные ритуалы или примеры из готовых процессов. Именно эти принципы я имею в виду, когда говорю об *инженерной разработке ПО*.

Скорость — это инструмент для достижения более качественных и эффективных результатов.

УПРАВЛЕНИЕ ПЕРЕМЕННЫМИ

Если мы хотим иметь возможность быстро, надежно и многократно тестировать и развертывать системы, стоит ограничить вариативность и **контролировать переменные**. Мы должны получать одинаковые результаты при каждом развертывании, поэтому необходимо автоматизировать его и управлять конфигурацией развертываемых систем.

Большое внимание следует уделить точкам контакта системы с неподконтрольными нам областями среды. Развертывая продукт в неконтролируемой среде, необходимо сократить зависимость от нее до минимума. Абстракция, разделение ответственности и слабая связанность — ключевые инструменты для этого.

Результаты тестов одной и той же версии продукта всегда должны быть одинаковыми. Если они разные, придется приложить больше усилий для лучшей изоляции теста от внешних воздействий или повысить детерминированность кода. Ключевыми инструментами здесь вновь выступают модульность и связность, разделение ответственности, абстракция и связанность.

Желание использовать длительное или ручное тестирование часто свидетельствует о недостаточном контроле переменных.

Мы часто не принимаем контроль переменных всерьез.

ЦЕНА ПЛОХОГО КОНТРОЛЯ

Однажды я консультировал крупную организацию, разрабатывавшую сложную распределенную систему. В проекте было занято более 100 команд разработчиков. Они попросили меня наладить тестирование производительности.

Разработчики создали большой комплекс сквозных тестов производительности для всей системы.

Они четырежды запускали эти тесты, но теперь не могли интерпретировать результаты.

Те оказались настолько разными, что их не удавалось сопоставить.

Одна из причин заключалась в том, что тесты проводились в корпоративной сети, поэтому на них влияли другие выполняемые в это время процессы.

Все усилия, затраченные на создание и выполнение тестов, оказались почти напрасными, потому что никто не понимал, что означают результаты.

Компьютеры предоставляют нам фантастические возможности. Избегающие космических лучей и столкновений нейтрин с помощью вентиляй NAND, имеющие аппаратные протоколы исправления ошибок, компьютеры и программы, которые на них работают, являются детерминированными. При одинаковых входных данных компьютеры всегда генерируют одинаковый результат. Единственное исключение — параллелизм.

Компьютеры также обладают невероятной скоростью работы; это отличная площадка для экспериментов. Мы можем отказаться от этих преимуществ либо же взять их под свой контроль и использовать для достижения целей.

Способы, используемые при проектировании и тестировании систем, влияют на степень такого контроля. Это еще одно преимущество разработки через тестирование.

Хороший тестируемый код не является многопоточным — за редким исключением.

Параллельный код трудно тестировать, потому что он не детерминирован. А значит, если мы заботимся о тестируемости кода, мы уделяем параллелизму больше внимания и стремимся перенести его на контролируемые, хорошо понятные границы системы.

По моему опыту, в этом случае код становится намного легче тестировать, потому что он детерминирован; его также намного легче понять; это означает, конечно же, что он становится намного более эффективным.

НЕПРЕРЫВНАЯ ДОСТАВКА

Непрерывная доставка — это философия, которая объединяет описанные выше идеи в эффективный, действенный и практичный метод разработки. Чтобы ПО всегда было готово к релизу, мы сосредоточиваемся на области оценки в пайплайне развертывания и на развертываемости ПО. Это позволяет структурировать код и организации и создавать **независимо развертываемые программные модули**.

Непрерывная доставка не просто автоматизация развертывания; это гораздо более важный принцип организации работы, цель которого — создание почти непрерывного потока изменений.

Чтобы создать такой поток, потребуется структурировать все процессы разработки. Это влияет на структуру организации, сводя к минимуму внутренние зависимости и способствуя автономии небольших команд, работающих быстро и качественно без необходимости координировать свои усилия с другими.

Достижение высокого уровня автоматизации, особенно в тестировании, позволяет быстро понимать, что наши изменения безопасны. В результате мы уделяем большое внимание тестированию, чтобы получить проверяемый продукт и извлекать выгоду из всех преимуществ, которые оно дает.

Непрерывная доставка помогает тестировать развертывание и конфигурацию систем и заставляет серьезно относиться к управлению переменными, добиваясь повторяемости и надежности в тестах и, как побочный эффект, в готовых развертваниях.

Непрерывная доставка — очень эффективная стратегия, на основе которой можно построить сильную инженерную разработку ПО.

ОБЩИЕ ИНСТРУМЕНТЫ ДЛЯ ПОДДЕРЖКИ РАЗРАБОТКИ

Это общие инструменты. Они применимы для решения любой задачи разработки.

Рассмотрим простой пример. Представьте, что мы хотим добавить в систему какой-то программный компонент — может быть, сторонний компонент, подсистему или фреймворк. Как его оценить?

Конечно, он должен работать и приносить пользу, но сперва, я полагаю, можно применить в качестве метрик идеи, изложенные в этой книге.

Является ли компонент развертываемым? Можно ли автоматизировать развертывание системы, чтобы оно стало надежным и многократным?

Является ли компонент тестируемым? Можно ли убедиться, что он делает то, что нужно? В нашу задачу не входит всеобъемлющее тестирование стороннего продукта; если нам приходится это делать, то, вероятно, он не очень хорош и недостаточно высокого качества. Однако в контексте нашей системы делает ли он то, что нужно, правильно ли он сконфигурирован, работает ли он, когда это нужно, и так далее? Как это проверить?

Позволяет ли компонент контролировать переменные? Можно ли развертывать его безопасно и многократно? Или управлять версиями развертывания и конфигурации?

Достаточна ли скорость компонента для работы в режиме непрерывной доставки? Удастся ли развернуть его за разумное время и запустить достаточно быстро, чтобы использовать и получать обратную связь несколько раз в день?

Если это программный компонент, для интерфейса которого мы будем писать код, то удастся ли поддерживать модульный подход или он навязывает собственную модель программирования, которая повлияет на работу других компонентов системы?

Ответ «нет» на любой из этих вопросов почти наверняка приведет к отказу от использования такого компонента еще до проверки того, как он работает и насколько полезен в других контекстах.

Если сервис, предоставляемый компонентом, заменяемый, я рекомендую искать альтернативы. Если сервис незаменим, стоит попытаться придать компоненту указанные выше свойства. Эти издержки необходимо учитывать при расчете его рентабельности.

В небольшом примере я попытался представить общий подход к разработке, в рамках которого инструменты обучения, управления сложностью и другие используются для поддержания инженерного стиля при разработке информированных решений и выборе на каждом этапе работы.

ИТОГИ

В этой главе я объединил взаимосвязанные идеи, описанные в книге, в согласованную модель эффективной разработки ПО. Взяв эти идеи за основу, мы получим более высокий результат, чем если решим игнорировать их.

Это лучшее из того, что может дать любой инструмент, процесс или дисциплина. Успех не гарантирован, но я верю, что, применяя мышление, которое я описываю в этой книге, вы будете создавать более качественный код, причем делать это быстрее.

ГЛАВА 15

СОВРЕМЕННЫЙ ИНЖЕНЕР-РАЗРАБОТЧИК

Все идеи этой книги тесно переплетены между собой. Существует их пересечение и избыточность. Невозможно по-настоящему разделить ответственность, не улучшая модульность.

Модульность, связность и разделение ответственности помогают эффективнее собирать **обратную связь** и тем самым облегчают **экспериментирование**.

Эти понятия я упоминал неоднократно на протяжении всей книги. Это неизбежно, и, полагаю, это говорит о многом.

Описанные здесь идеи не только тесно связаны между собой, но и практически универсальны, и это очень важно.

Техническими деталями легко увлечься. Какой язык, операционную систему, текстовый редактор или фреймворк выбрать — это в конечном счете должно значить меньше, чем необходимые для работы с ними навыки.

Как я уже говорил, лучшие разработчики, которых я встречал, всегда создавали хороший продукт, какими бы инструментами они ни пользовались. Конечно, многие из них прекрасно разбирались в инструментах, но не этим определялись их способности и ценность для организаций, где они работали.

Все изложенные мной идеи, думаю, вам знакомы, но скорее всего, вы не рассматривали их как основополагающие для своей работы. Моя книга призвана изменить ваше мнение. Я хочу не только напомнить, что такие подходы существуют, но и рекомендовать вам взять их за основу вашей профессиональной деятельности.

Они позволяют оптимизировать работу, кратно увеличивают возможность обучаться и управлять сложностью создаваемых систем и составляют реальную основу подхода к разработке ПО, который с полным правом можно назвать инженерным.

Если мы будем следовать им, у нас будет больше шансов на успех, чем в противном случае.

Это не механический подход. У вас ничего не получится, если вы просто скопируете мой или чей-то еще опыт, точно так же как вы не создадите отличный автомобиль, следуя некоему теоретическому руководству по сборке автомобилей.

Необходимо действовать вдумчиво, усердно, осторожно и интеллектуально. Создавать программные продукты непросто. Разработка, естественно, включает написание кода, но как я уже говорил, она подразумевает гораздо больше, чем просто кодирование.

Модель кажется простой, но ее сложно применить.

Простота обуславливается тем, что модель включает всего десять основных идей в двух группах и несколько инструментов для их реализации: тестируемость, развертываемость, скорость, контроль переменных и непрерывная доставка. Однако применение этих идей довольно сложно и требует тщательного обдумывания.

Овладение указанными инструментами и использование указанных идей в качестве основополагающих принципов увеличивает шансы на успех, и это то, на чем, по моему мнению, базируется программная инженерия.

Я не утверждаю, что разработка — это просто; это сложно, поэтому необходимо подойти к ней вдумчиво.

Для меня это значит, что необходимо тщательно обдумывать задачи с разных сторон. Это поможет найти ответы на вопросы, решений для которых у нас пока нет.

Десять идей, о которых я рассказывал, позволяют работать именно так, и многие разработчики извлекают из них выгоду.

Понимание природы нашей отрасли влияет на возможность добиваться прогресса. Понимание сложности систем, которые мы создаем, и принципов работы продуктов важно для успеха. Легкомысленное отношение

к разработке как просто к написанию кода для полулинейной последовательности инструкций всегда обречено на провал, кроме как для простейших задач.

Нам нужны особые инструменты мышления и умение адаптировать их к любым обстоятельствам. Мне кажется, это основа всей инженерной разработки.

ИНЖЕНЕРИЯ КАК ЧЕЛОВЕКО-ОРИЕНТИРОВАННЫЙ ПРОЦЕСС

В контексте разработки ПО термин «инженерия» часто определяют неправильно.

Большинство определений *инженерии* начинаются с чего-то вроде «изучение работы инженера», а затем идет описание использования данных математики и естественных наук. В действительности речь идет о процессе, подходе к работе.

Определение, которое я привел в начале этой книги, как мне кажется, раскрывает суть понятия.

Инженерия — это эмпирический научный подход к поиску эффективных, экономичных решений практических задач при разработке ПО.

Инженерия эмпирична в том смысле, что мы не применяем научный подход постоянно, чтобы результаты всегда были идеальны. (На самом деле наука тоже так не работает, она просто стремится к этому.)

Инженерия — это принятие рационально обоснованных решений, часто исходя из неполных данных, а затем наблюдение за практической реализацией этих решений на основе обратной связи.

В ее основе лежит научное мышление. Мы измеряем то, что можем измерить. Используем экспериментальный подход при внесении изменений. Контролируем переменные, чтобы понять влияние изменений. Разрабатываем и поддерживаем модель, гипотезу, понимание которой постоянно углубляем по мере того, как она развивается.

Важно, чтобы решения, которые мы находим, были эффективны, как и то, каким образом мы это делаем.

Создаваемые нами системы должны быть наиболее простыми и работать максимально быстро, потребляя при этом минимум необходимых ресурсов.

Кроме того, их создание должно отнимать как можно меньше времени и усилий. Это важно по экономическим причинам и исходя из необходимости эффективного обучения. Своевременность обратной связи служит хорошим показателем эффективности работы. Качественная обратная связь, как мы видели в главе 5, также лежит в основе эффективного обучения.

В дополнение к применению инженерного мышления к разработке в целом важно понять, что организации и команды, в которых мы работаем, также являются информационными системами, поэтому идеи управления сложностью в равной, если не в большей, степени применимы и к ним.

ОРГАНИЗАЦИИ – ЛИДЕРЫ В ЦИФРОВОЙ СФЕРЕ

Компании и бизнес-лидеры часто говорят о цифровом прорыве, под которым подразумевают применение цифровых технологий для переосмысления и трансформации традиционного бизнеса. Amazon реорганизует цепочку розничных поставок, Tesla меняет основы подхода к производству автомобилей, Uber превращает услуги такси в гиг-экономику. Их идеи бросают вызов традиционному бизнесу и бизнес-мышлению.

Одна из определяющих характеристик таких организаций — ими почти всегда руководят инженеры. Разработка программного обеспечения там рассматривается не как затратная часть или функция поддержки; это сам бизнес. Даже такая компания, как Tesla, производящая физический продукт, выстроена на принципах разработки ПО.

Компанию Tesla можно с полным правом считать компанией непрерывной доставки в том смысле, что в ней существует возможность переналадить производство, часто с помощью программных инструментов, чтобы внедрить любую новую идею.

Программные средства меняют способ ведения бизнеса и в этом плане бросают вызов многим традиционным инструментам.

Одну из моих любимых моделей разработал Ян Буш; он описывает ее как «BAPO и OBAP»¹.

Рисунки 15.1 и 15.2 объясняют ее.

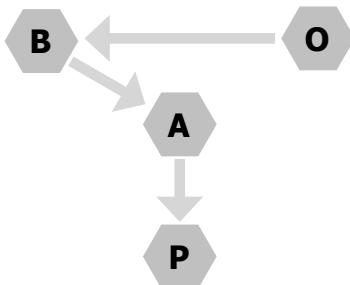


Рис. 15.1. Как работает большинство организаций (OBAP)

Большинство компаний применяют модель OBAP (Organization, Business, Architecture, Process – организация, бизнес, архитектура, процесс) (рис. 15.1). Сначала они выстраивают структуру организации, отделов, команд, обязанностей и так далее. Затем определяют бизнес-стратегию и способы получения дохода, прибыли или других результатов, исходя из ограничений этих организационных решений. Затем подбирают подходящую архитектуру систем и, наконец, процесс, обеспечивающий работу этой системной архитектуры.

Это странный подход. В этом случае бизнес-видение и цели ограничены организационной структурой.

Более разумная модель – рассматривать структуру организации как инструмент BAPO (Business, Architecture, Process, Organization – бизнес, архитектура, процесс, организация).

Мы определяем бизнес-видение и цели, решаем, какими техническими средствами этого достичь (архитектура), выясняем, как создать что-то подобное (процесс), а затем подбираем подходящую организационную структуру для его поддержки.

¹ Ян Буш описывает эти идеи в блоге Structure Eats Strategy (<https://bit.ly/33GBrR1>) и в книге «Speed, Data and Ecosystems».

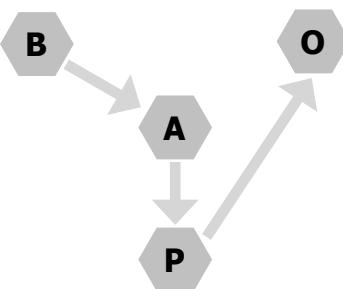


Рис. 15.2. Как необходимо организовывать работу (BAPO)

Если рассматривать способы организации людей в качестве инструмента достижения цели, то для успешного его использования необходимо применять инженерное мышление, описанное в моей книге.

Как и в случае с любой другой информационной системой, управление связанностью в организациях является одним из ключей к успеху. Точно так же как оно верно для разработки, оно верно и для организаций. Модульные, связные организации с грамотным разделением ответственности и команд, когда абстракция позволяет им скрывать информацию от других структурных подразделений, более масштабирумы и более эффективны, чем команды с сильной связанностью, где продвигаться в работе можно только сообща.

Это одна из причин, по которой так сложно масштабировать организации. По мере их роста стоимость связанности увеличивается. Современные крупные, быстро растущие компании необходимо проектировать таким образом, чтобы свести к минимуму связанность между командами.

Неслучайно исследование, лежащее в основе книги «Ускоряйся!», показало, что одним из определяющих признаков высокоеэффективных команд, основанных на метриках стабильности и пропускной способности, служит умение принимать решения внутри команды, без необходимости координации с другими группами. Такие команды информационно разделены.

Это важно. В этом заключается разница между, например, Amazon, производительность которой возрастает более чем вдвое, когда размер компании увеличивается в два раза, и фирмой, имеющей традиционную

структурой. Производительность последней повышается всего на 85% при двукратном увеличении размера¹.

РЕЗУЛЬТАТЫ И МЕХАНИЗМЫ

Когда я подошел к заключительной части книги, я вступил в онлайн-полемику о важности результатов и механизмов. Я был абсолютно уверен, что все согласятся со мной в том, что результаты важнее механизма. Однако это оказалось не так.

Я не думаю, что собеседники не разделяли мое мнение по глупости. Анализируя их ответы, я пришел к выводу, что они в конечном итоге приняли мою точку зрения. Они не отрицали важность результатов; они имели в виду какие-то неочевидные, но важные для них вещи или механизмы, которые им нравились и помогали достигать результатов.

Успешный результат разработки — это комплексное понятие. Начать можно с некоторых очевидных параметров, которые легко измерить. Например, с коммерческих результатов для некоторых видов бизнеса и программных продуктов; это один из показателей успеха. Можно измерить объем использования, и успешность продукта с открытым исходным кодом часто оценивается количеством скачиваний.

Еще применяют метрику производительности и качества, **стабильности и пропускной способности DORA**, которая показывает, что успешные команды выпускают программные продукты высокого качества и очень эффективно. Или оценивают удовлетворенность клиентов, используя различные метрики.

Желательно получить высокий балл по всем этим параметрам. Некоторые из них зависят от контекста, а некоторые нет; но качественная эффективная работа (высокий балл **стабильности и пропускной способности**) будет более успешной в любом контексте, поэтому я считаю такую оценку правильным инструментом.

¹ У Джеймса Льюиса, изобретателя термина «микросервисы», есть интересная презентация, касающаяся работы Института сложности Санта-Фе в области нелинейной динамики. См. <https://youtu.be/tYHJgvJzbAk>.

О том, что «результаты важнее механизмов», я упомянул в контексте сравнения идеи о непрерывной доставке и идеи DevOps¹.

Я исходил из того, что непрерывная доставка определяет именно желаемый результат, а не механизм, поэтому она более полезна как общий организующий принцип для руководства стратегией развития.

DevOps — чрезвычайно полезный набор практик; если вы будете грамотно применять их все, вы сможете непрерывно создавать ценность для пользователей и клиентов. Однако если какой-то процесс выходит за рамки DevOps, то становится менее очевидно, как с ним работать.

Повторю принципы непрерывной доставки: обеспечивать постоянную готовность продукта к релизу, оптимизировать продукт для получения быстрой обратной связи и стремиться к получению наиболее эффективной обратной связи от этапа идеи продукта до его доставки пользователям.

Если серьезно отнестись к этим принципам, они помогут найти уникальные инновационные решения для задач, с которыми мы никогда раньше не сталкивались.

С тех пор как я и другие разработчики стали применять принципы непрерывной доставки, нам еще не довелось поработать над созданием автомобиля, космического корабля или телекоммуникационной сети. Каждая из этих сфер требует совершенно разной организации тех систем, которые мы создавали, когда с Джезом Хамблом писали нашу книгу.

Своим клиентам я рекомендую реализовывать конкретные цели и задачи на основе обратной связи от их пайплайнов развертывания. Обычно я советую стремиться к тому, чтобы развертывание занимало от пяти минут после этапа коммита до одного часа для всего пайплайна. Создавайте продукт, готовый к релизу каждый час.

Однако это невозможно для Tesla, которая конструирует автомобиль, или SpaceX, которая строит ракету, или Ericsson, которая развертывает глобальную мобильную инфраструктуру: этому препятствуют физические процессы сжигания кремния или металлообработки.

Однако принципы непрерывной доставки остаются в силе.

¹ Если вам интересны мои размышления о непрерывной доставке и DevOps, посмотрите это видео на моем канале в YouTube: <https://youtu.be/-sErBqZgKGs>.

Работайте так, чтобы ваш продукт всегда был готов к релизу. Вы по-прежнему можете проводить тщательное тестирование, немедленно отказываясь от любых изменений, если хотя бы один тест завершается ошибкой. Оптимизируйте продукт для получения быстрой обратной связи. Автоматизируйте тесты, чтобы выполнять большинство из них на симуляции и чтобы обратная связь всегда была быстрой и эффективной.

Более того, идеи, основанные на научном подходе, на которых базируется непрерывная доставка, являются самыми устойчивыми.

- **Характеристика:** проведите наблюдение текущего состояния.
- **Гипотеза:** создайте описание, теорию, объясняющую ваше наблюдение.
- **Прогноз:** составьте прогноз на основе гипотезы.
- **Эксперимент:** проверьте свой прогноз.

Чтобы воспользоваться полученными знаниями, необходимо контролировать переменные. Это можно сделать несколькими способами. Например, двигаться небольшими шагами, чтобы понять влияние каждого шага. Или же обеспечивать полный контроль конфигурации системы и ограничивать область изменений с помощью методов управления сложностью, которые мы обсуждали.

Вот что я имею в виду под инженерией — идеи, методы и инструменты, которые повышают шансы на успех.

Возможно, вы не достигнете целей обратной связи, которые я обычно рекомендую, но вы можете задать их в качестве ориентира и работать над их достижением в рамках физических или, возможно, финансовых ограничений.

УСТОЙЧИВОСТЬ И ШИРОКАЯ ПРИМЕНИМОСТЬ

Настоящая инженерная разработка не должна зависеть от технологии. Она должна быть долговечной и полезной, помогая отвечать на абсолютно новые вопросы и понимать совершенно новые идеи и технологии.

Можно попробовать!

Вся моя карьера — это совместная с коллегами проектная разработка программного обеспечения, но применим ли такой подход к другим формам разработки, например к машинному обучению (МО)?

На рис. 15.3 показан типичный рабочий процесс МО. Обучающие данные организуются, очищаются и готовятся к использованию. Выбираются подходящие алгоритмы МО, определяются фитнес-функции для применения к входным данным, а затем алгоритмы МО запускаются на обучающих данных. Они циклически используют различные решения, пока не будет достигнута желаемая точность соответствия фитнес-функции. На этом этапе сгенерированный алгоритм может быть развернут в рабочей среде.

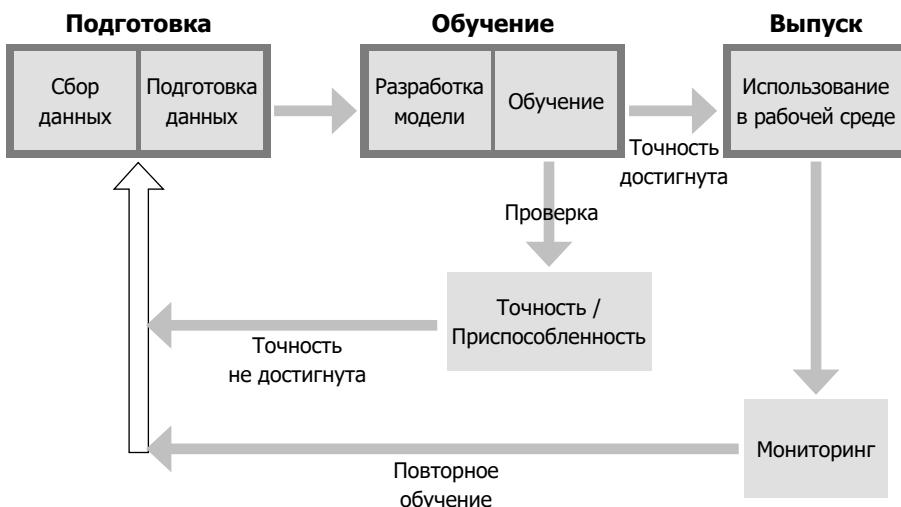


Рис. 15.3. Типичный рабочий процесс машинного обучения

Если точность не достигается, процесс повторяется, и разработчики/специалисты по анализу данных меняют обучающие данные и фитнес-функции в поисках эффективного решения.

После того как алгоритм запущен в производство, его можно отслеживать и в случае обнаружения проблем вернуть в цикл для повторного обучения.

Как здесь использовать инженерную модель?

Очевидно, что создание систем машинного обучения связано с обучением, а не только с машинами. Разработчикам необходимо оптимизировать

свою работу, чтобы знать, какие данные использовать для обучения систем и какие параметры фитнес-функций управляют таким обучением.

Обучающие системы МО содержат большие массивы данных, поэтому для достижения успеха необходим тщательный анализ и активное управление сложностью. Специалисты по анализу данных часто теряются в этих массивах и не способны получить воспроизводимый результат.

Сам процесс разработки, очевидно, лучше всего организовать как итеративный. Сбор и подготовка обучающих данных, а также задание и корректировка фитнес-функций — принципиально итеративные процессы. Обратная связь предоставляется в форме точного соответствия фитнес-функции. Очевидно, что итерации должны быть короткими, а обратная связь — быстрой и четкой. Весь процесс — это повторение и корректировка экспериментов.

Организуя процессы таким образом, мы сможем работать лучше. Разумно оптимизировать процессы так, чтобы разработчики действовали быстро, и улучшать качество обучения на каждой итерации. Это означает, что мы движемся небольшими шагами и четко понимаем характер и качество обратной связи.

Рассматривая каждый маленький шаг как эксперимент, мы лучше контролируем переменные, например версию, управляющую сценариями и обучающими данными.

Странно даже представить, что эта часть процесса организована и работает не по принципу динамического, итеративного, основанного на обратной связи эмпирического открытия, а как-то иначе.

Почему эмпирического? Потому что данные не организованы, а результаты достаточно сложны, чтобы не быть детерминированными на уровнях контроля, устанавливаемых в машинном обучении.

Это вызывает следующий интересный вопрос. Возможен ли больший контроль? У меня состоялся однажды разговор с экспертом в области машинного обучения. Он заинтересовался моей простой схемой (рис. 15.3): «Что вы имеете в виду под мониторингом? Как узнать результат?»

Применяя инженерный подход, мы рассматривали бы выпуск модели на продакшен как эксперимент. Если это эксперимент, то необходимо сделать прогноз и проверить его. При создании системы МО мы описываем, что

хотим сделать. Мы можем спрогнозировать желаемые результаты. Это больше, чем фитнес-функция, и похоже на определение области, содержащей правильные ответы.

Если система МО создана для повышения продаж книг, она, скорее всего, не станет работать, если область допустимого ответа обозначить как «весь мир».

Как насчет управления сложностью? Одна из проблем машинного обучения заключается в том, что люди, которые им занимаются, часто не имеют опыта разработки ПО. В результате они не используют многие ставшие обычными техники, даже базовые, например контроль версий.

Тем не менее возможные способы применения инженерных принципов, о которых я рассказал в этой книге, очевидны: модульный подход к написанию скриптов сбора и очистки данных, а также заданию фитнес-функций. Это код, поэтому используйте инструменты, необходимые для написания хорошего кода. Управляйте переменными, держите взаимосвязанные идеи рядом с помощью связности и разделяйте несвязанные идеи с помощью модульности, разделения ответственности, абстракции и ослабления связности. Эти же принципы работают и для используемых данных.

Применение таких идей к данным и выбор модульных обучающих данных (в том смысле, что они фокусируются на правильных аспектах задачи) позволяют разработчикам систем МО выполнять итерации быстрее. Это ограничивает изменения и фокусирует процесс обучения и, возможно, помогает более эффективно и масштабируемо управлять обучающими данными. Это одна из действительных целей *очистки данных*.

Также важно обеспечить разделение ответственности для данных и фитнес-функций. Разумно полагать, что системы МО способны принимать неверные решения, основываясь на неверно установленных зависимостях, к примеру между экономическими условиями и этническими группами или заработной платой и полом. Это происходит из-за плохого разделения ответственности в обучающих данных, а также из-за неверных представлений о процессах в обществе.

Здесь я остановлюсь, поскольку не очень знаком с областью машинного обучения. Я хочу сказать, если эти интеллектуальные инструменты применимы во всех сферах, их можно использовать и для решения таких задач, о которых мы не подозреваем.

Я не утверждаю, что нашел правильные ответы для своего примера, но моя модель позволила задать вопросы, которые, насколько я понимаю, обычно не задают при машинном обучении. Эти вопросы помогут оптимизировать процессы, улучшить качество создания систем МО и даже сами системы.

Всего этого мы и ожидаем от по-настоящему инженерного подхода. Он не дает готовых ответов, но предоставляет способ их получить.

ОСНОВЫ ИНЖЕНЕРНОЙ ДИСЦИПЛИНЫ

Идеи, изложенные в этой книге, составляют основу инженерной дисциплины, которая может повысить шансы на успех.

Не столь важно, какой язык программирования вы выберете, какой фреймворк или методологию используете. Главное — идеи, о которых я вам здесь рассказал.

Дело не в том, что эти инструменты не имеют отношения к нашей работе; конечно, имеют. Они важны в той же степени, как важна конкретная модель молотка, которым плотник забивает гвозди.

В разработке их выбор не только личный, поскольку он влияет на совместную работу команды. Но по сути предпочтение одной технологии перед другой меньше влияет на результат, чем то, как эта технология применяется.

В книге я постарался описать идеи, которые помогут более эффективно использовать выбранные инструменты.

ИТОГИ

В своей работе я уже много лет использую идеи и принципы, с которыми познакомил вас в этой книге. Я надеюсь, что смог сформулировать их максимально понятно.

В последние годы я почти исключительно занимаюсь созданием сложных систем. Мне посчастливилось решать задачи, с которыми мало кто, если вообще кто-то, сталкивался прежде. Всякий раз, когда я и моя команда

начинали боксовать, мы применяли описанные здесь идеи. Они служили для нас ориентиром, направляя к лучшим результатам, независимо от рода задачи, даже когда мы совершенно не представляли, что делать дальше.

Сейчас я главным образом консультирую крупные международные бизнес-компании и часто сталкиваюсь с инновациями беспрецедентных масштабов. И мои идеи по-прежнему работают и помогают решать действительно сложные задачи.

Когда я пишу код для себя, что до сих пор приносит мне огромное удовольствие, я применяю эти же идеи в малом и часто самом простом масштабе.

Если, чтобы обучаться максимально эффективно, вы всегда будете оптимизировать свою работу, вы станете работать лучше.

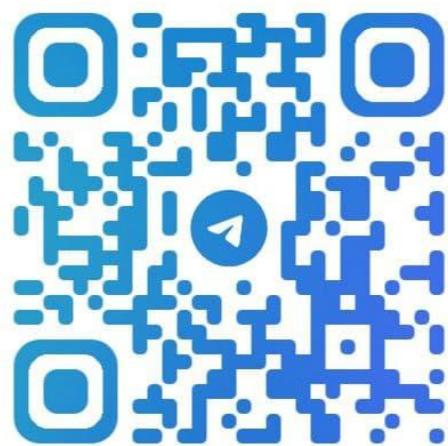
Если всегда, при любом масштабе задачи, вы научитесь управлять ее сложностью, вы получите неограниченные возможности для совершенствования своей работы.

Это то, что отличает по-настоящему инженерную разработку. Применяя инженерный подход, мы с большей вероятностью будем создавать лучшие продукты быстрее.

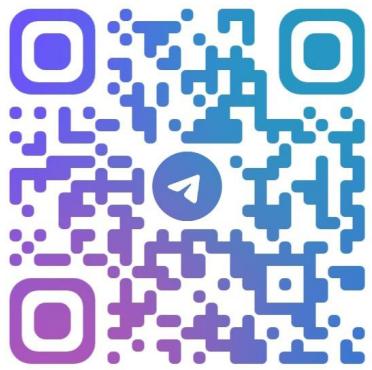
Это очень важные и ценные идеи. Я надеюсь, что мне удалось изложить их так, чтобы они принесли пользу вам и вашей работе.



@BOOKOFGEEK



@JAVALIB



@KOTLINSENIOR



@BOOKOFGEEK

