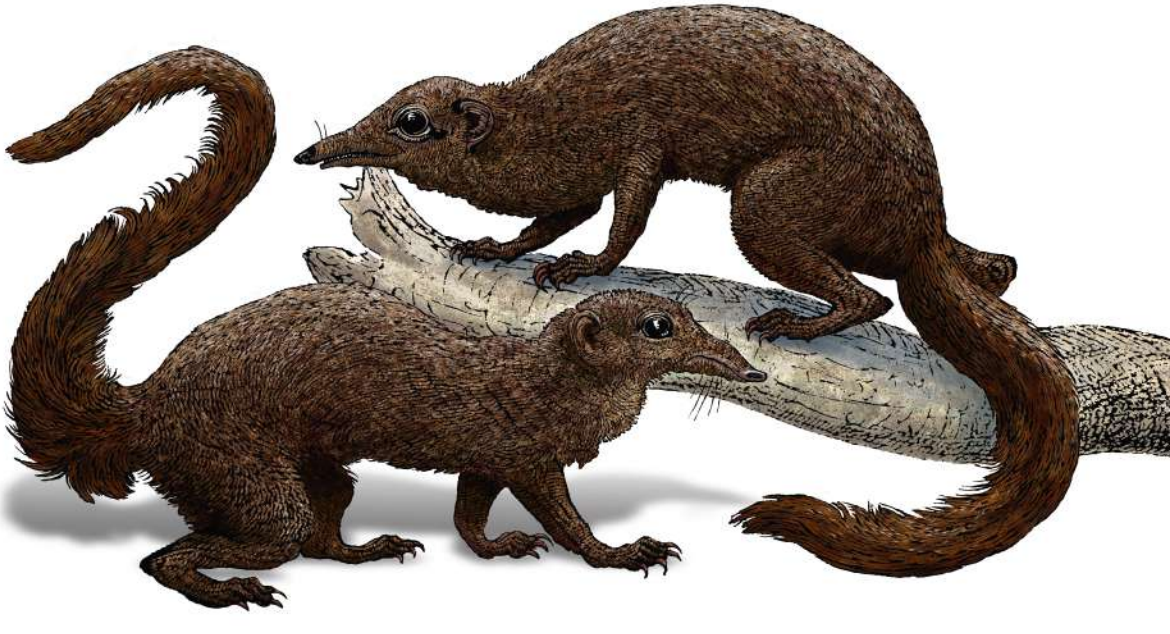
O'REILLY®

Метрики программной архитектуры

Кейсы, повышающие качество ПО



SPRINT book

Нил Форд, Эндрю Хармел-Лоу,
Кристиан Чичери, Дэвид Фарли и др.

Software Architecture Metrics

Case Studies to Improve the Quality of Your Architecture

Neal Ford, Dave Farley, Christian Ciceri, Andrew Harmel-Law, Michael Keeling, Carola Lilienthal, João Rosa, Alexander von Zitzewitz, Eoin Woods, Rene Weiss

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Метрики программной архитектуры

Кейсы, повышающие качество ПО

Нил Форд, Дэвид Фарли,
Кристиан Чичери, Эндрю Хармел-Лоу,
Майкл Килинг, Карола Лилиенталь, Жоао Роза,
Александр фон Цитцевиц, Эойн Вудс, Рене Вайсс

SPRINT book 2026

ББК 32.973.26-018.2
УДК 004.415.53
Ф79

Форд Н., Фарли Д., Чичери К., Хармел-Лоу Э., Килинг М., Лилиенталь К., Роза Ж., фон Цитцевиц А., Вудс Э., Вайсс Р.

Ф79 Метрики программной архитектуры. Кейсы, повышающие качество ПО. — Астана: «Спринт Бук», 2026. — 240 с.: ил.

ISBN 978-601-12-9322-8

Как объективно оценить качество архитектуры? Как вовремя заметить накопление технического долга и связать архитектурные решения с бизнес-результатами? Десять признанных экспертов делятся проверенными метриками и практическими подходами, которые позволяют архитекторам и техническим лидерам принимать обоснованные решения.

В современном мире разработки ПО архитектура приложения играет ключевую роль. Но как убедиться, что выбранная архитектура действительно отвечает всем требованиям проекта? Именно здесь на сцену выходят метрики — комплексный процесс проверки соответствия архитектурных решений заданным требованиям и критериям качества. Чем позже обнаружена проблема, тем дороже обходится ее исправление. А когда речь идет об архитектурных решениях, цена ошибки возрастает в геометрической прогрессии.

Качественная архитектура — это не статичное состояние, а процесс постоянной оценки и адаптации, основанный на данных и метриках.

Эта книга не про теорию, а про практику применения и реализацию. Она основана на реальном опыте и написана для архитекторов ПО и разработчиков. В книге представлены ключевые метрики программной архитектуры, которые помогут определить правильные показатели эффективности (KPI) и измерить полученные результаты. Вы узнаете об эффективности принятия решений и сбора метрик.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.26-018.2
УДК 004.415.53

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

ISBN 978-1098112233 англ.

Authorized Russian translation of the English edition of Software Architecture Metrics ISBN 9781098112233 © 2022 Apiumhub S.L.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-601-12-9322-8

© Перевод на русский язык ТОО «Спринт Бук», 2026
© Издание на русском языке, оформление ТОО «Спринт Бук», 2026

Краткое содержание

От издательства 12

Предисловие 13

Глава 1. Четыре ключевые метрики в действии 19

Глава 2. Пирамида тестирования с использованием фитнес-функций: аналогия для тестов и метрик архитектуры 42

Глава 3. Эволюционная архитектура: управление архитектурой через тестируемость и развертываемость 67

Глава 4. Повышение качества архитектуры с помощью индекса зрелости модульности (MMI) 74

Глава 5. Локальные сборки и метрики: инструменты для выживания в переходный период к DevOps 92

Глава 6. Масштабирование организации: центральная роль программной архитектуры 112

Глава 7. Роль метрик в программной архитектуре 140

Глава 8. От метрик к инженерным практикам 163

Глава 9. Использование метрик ПО для обеспечения сопровождаемости 182

Глава 10. Измерение неизвестного с помощью подхода «цель — вопрос — метрика» 213

Об авторах 232

Иллюстрация на обложке 237

Оглавление

От издательства **12**

О научном редакторе русского издания 12

Предисловие **13**

Что вы узнаете? 14

Для кого эта книга 14

Условные обозначения 15

Благодарности 15

От Кристиана Чичери 16

От Дэвида Фарли 16

От Нила Форда 16

От Эндрю Хармел-Лоу 16

От Майкла Килинга 17

От Каролы Лилиенталь 17

От Жоао Розы 17

От Александра фон Цитцевица 18

От Рене Вайсса 18

От Эойна Вудса 18

Глава 1. Четыре ключевые метрики в действии **19**

Определение и сбор данных 20

Рефакторинг схемы 22

Пайплайны как первая отправная точка 23

Определение точек сбора данных 25

Сбор данных и расчет 30

Отображение и понимание данных 33

Целевая аудитория 33
Визуализация 34
Главная страница 39
Обсуждения и понимание 39
Ответственность и улучшение 40
Заключение 41

Глава 2. Пирамида тестирования с использованием фитнес-функций: аналогия для тестов и метрик архитектуры 42

Фитнес-функции и метрики 42
Фитнес-функции: покрытие тестами 44
Фитнес-функции: интеграционные тесты с сетевыми задержками 45
Категории фитнес-функций 46
Обязательные категории фитнес-функций 47
Необязательные категории фитнес-функций 51
Категории фитнес-функций: обзор каталога 52
Пирамида тестирования 53
Пирамида фитнес-функций 55
Верхний слой 57
Средний слой 57
Нижний слой 58
Примеры и их полная категоризация 59
Полная категоризация примеров верхнего уровня 61
Разработка собственных фитнес-функций и метрик 63
Заключение 66

Глава 3. Эволюционная архитектура: управление архитектурой через тестируемость и развертываемость 67

Важность обучения и открытий 68
Инструменты для устойчивых изменений 68
Тестируемость: создание высококачественных систем 69
Развертываемость: масштабирование разработки систем 71
Заключение 72

Глава 4. Повышение качества архитектуры с помощью индекса зрелости модульности (MMI) 74

Технический долг 74
Возникновение технического долга 75
Оценка через MMI 77
Модульность 78
Иерархичность 80
Соответствие паттернам 82
Расчет MMI 84
Ревью архитектуры для определения MMI 88
Заключение 91

Глава 5. Локальные сборки и метрики: инструменты для выживания в переходный период к DevOps 92

Ключевые термины 93
CI/CD 93
DevOps 95
Сдвиг ответственности 96
Контроль над локальной средой 96
Приватная сборка 98
Кейс: нестабильная основная ветка 100
Ошибка A1 101
Ошибка A2 101
Ошибка A3 102
Ошибка A4 102
Кейс: замкнутый круг в консалтинговой компании 103
Метрики 105
Время до получения обратной связи 106
Количество предотвратимых проблем интеграции в развернутом приложении за итерацию 106
Время на восстановление стабильности основной ветки за итерацию 107
Затраты на приватные сборки 108

Метрики на практике ... 108
Длительный срок получения обратной связи, много предотвратимых проблем интеграции, короткий срок стабилизации основной ветки ... 108
Короткий срок получения обратной связи, много предотвратимых проблем интеграции, короткий срок стабилизации основной ветки ... 109
Длительный срок получения обратной связи, мало предотвратимых проблем интеграции, короткий срок стабилизации основной ветки ... 109
Мало предотвратимых проблем интеграции и длительный срок стабилизации основной ветки ... 110
Заключение ... 110

Глава 6. Масштабирование организации: центральная роль программной архитектуры ... 112

YourFinFreedom: отказ от монолита ... 114
Реализация «большого распределенного кома грязи» ... 117
В поисках направления ... 119
От максимальных усилий к целенаправленным действиям ... 120
Используйте метрику, чтобы повысить соответствие программной архитектуры поставленным целям ... 123
Управление ожиданиями и коммуникация ... 133
Изучение и развитие архитектуры ... 136
Как сейчас дела у Анны? ... 138
Заключение ... 138

Глава 7. Роль метрик в программной архитектуре ... 140

Внедрение метрик в программную архитектуру ... 141
Способы получения метрик ... 144
Получение метрик приложений и инфраструктуры в работающей системе ... 144
Анализ ПО ... 145
Анализ дизайна ... 145
Оценки и модели ... 145
Фитнес-функции ... 146

Измерение свойств системы ... 147
Производительность ... 147
Масштабируемость ... 149
Доступность ... 152
Безопасность ... 154
С чего начать ... 156
Вымышленный пример ... 157
Подводные камни ... 160
Заключение ... 161

Глава 8. От метрик к инженерным практикам ... 163
Путь к фитнес-функциям ... 163
От метрик к инженерным практикам ... 166
Автоматизация превращает метрики в рабочий инструмент ... 169
Кейс: связанность ... 170
Кейс: проверка безопасности на наличие уязвимостей нулевого дня ... 175
Кейс: фитнес-функции для проверки точности воспроизведения систем ... 177
Заключение ... 181

Глава 9. Использование метрик ПО для обеспечения сопровождаемости ... 182
Зачем использовать метрики ... 182
Энтропия убивает ПО ... 183
Токсичность циклических зависимостей ... 185
Как могут помочь метрики ... 186
Почему метрики не используются повсеместно ... 187
Инструменты для сбора метрик ... 188
Полезные метрики ... 189
Метрики для оценки связанности и структурной эрозии ... 189
Метрики для измерения размера и сложности ... 201
Метрики истории изменений ... 204
Другие полезные метрики ... 206
Архитектурные фитнес-функции ... 208

Как отслеживать динамику метрик ... 210
Несколько золотых правил для улучшения ПО ... 211
Заключение ... 212

Глава 10. Измерение неизвестного с помощью подхода «цель — вопрос — метрика» ... 213
Подход «цель — вопрос — метрика» ... 214
Создание GQM-дерева ... 214
Расстановка приоритетов для метрик и разработка стратегии сбора данных ... 217
Кейс: команда, которая научилась видеть будущее ... 220
Контекст системы ... 220
Инцидент № 1: слишком много запросов к сервису Foo ... 221
Инцидент № 2: заглядывая в будущее ... 225
Выводы ... 226
Проведение воркшопа по GQM ... 226
Краткое описание воркшопа ... 226
Этапы воркшопа ... 228
Рекомендации и советы по организации воркшопа ... 229
Заключение ... 230

Об авторах ... 232

Иллюстрация на обложке ... 237

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@sprintbook.kz (издательство «SprintBook», компьютерная редакция).

Мы будем рады узнать ваше мнение!

О научном редакторе русского издания

Игорь Курочкин — эксперт в Enabling.team. Помогает развивать инженерную культуру, процессы и практики, платформенные и продуктовые команды в технологических и Enterprise-компаниях. Руководил направлением анализа и технологического консалтинга в Express 42. Работал платформенным и SRE-инженером в Microsoft и Skype, стартапах Qik и Scalaxy. Выступал на конференциях HighLoad, South HUB, AgileDays, TeamLeadConf, DevOpsConf, DevOops.

Предисловие

Метрики программной архитектуры применяются для оценки качества архитектуры и удобства сопровождения[1] проекта программной разработки, а также для раннего предупреждения о рисках накопления архитектурного или технического долга. В этой книге десять ведущих практиков — Кристиан Чичери (Christian Ciceri), Дэвид Фарли (David Farley), Нил Форд (Neal Ford), Эндрю Хармел-Лоу (Andrew Harmel-Law), Майкл Килинг (Michael Keeling), Карола Лилиенталь (Carola Lilienthal), Жоао Роза (João Rosa), Александр фон Цитцевиц (Alexander von Zitzewitz), Рене Вайсс (Rene Weiss) и Эойн Вудс (Eoin Woods) — представляют ключевые метрики программной архитектуры, которые должен знать каждый архитектор. Все авторы являются признанными экспертами, они публикуют статьи и книги по программной архитектуре, регулярно выступают на международных конференциях и проводят практические семинары.

Конечно, теория и практика должны быть сбалансированы, но конкретно эта книга посвящена именно практике и реализациям — проверенным подходам, которые уже доказали свою эффективность, — и включает полезный опыт и примеры решений. Наша цель — не только повысить качество архитектуры, но и связать объективные метрики с важными для бизнеса результатами, учитывая конкретный контекст проекта и неизбежные компромиссы.

Мы провели опрос и выяснили, что потребность в материалах по метрикам программной архитектуры высока, однако количество таких доступных материалов оказалось довольно скромным. Мы надеемся, что наша работа поможет изменить ситуацию и позволит вам правильно выбирать ключевые показатели эффективности (KPI), а также точно и обоснованно измерять результаты.

Мы благодарны Global Software Architecture Summit, который вновь собрал нас вместе и подал идею совместной книги о метриках программной архитектуры. Все главы и примеры в книге так же разнообразны, как

[1] Maintainability также переводят как «сопровождаемость» или «поддерживаемость». — *Примеч. науч. ред.*

и сами авторы: мы специально использовали примеры из разных отраслей и для решения различных задач, чтобы каждый читатель мог найти подходящее решение или источник вдохновения.

Что вы узнаете?

Прочитав эту книгу, вы научитесь:

- оценивать, насколько хорошо ваша архитектура ПО соответствует поставленным целям;
- направлять развитие архитектуры так, чтобы она была более тестируемой и развертываемой;
- расставлять приоритеты в работе над программной архитектурой;
- формировать предсказуемость поведения, используя возможности наблюдения;
- определять KPI для проекта программной разработки;
- создавать и автоматизировать дашборд для метрик;
- анализировать и измерять успешность проекта или процесса;
- создавать программную архитектуру под конкретную цель.

Для кого эта книга

Эта книга написана архитекторами ПО для архитекторов ПО. Эта книга для вас, если вы хотите изучить успешные примеры из практики и больше узнать о том, как эффективно принимать решения и выполнять измерения, независимо от того, являетесь ли вы штатным сотрудником компании-разработчика или независимым консультантом.

Десять авторов с богатым практическим опытом делятся своими советами и наработками, предлагая разнообразные взгляды и идеи. В зависимости от ваших проектов некоторые главы могут оказаться для вас полезнее других. Книгой можно пользоваться как регулярно, так и эпизодически — сначала лишь для того, чтобы определить KPI, а позже — чтобы обучить и вдохновить новых членов команды.

Правильно подобранные метрики и инструменты для оценки программной архитектуры делают ее проверку гораздо быстрее и дешевле. Они позволяют запускать проверки с самого начала проекта и до конца его жизненного цикла. Метрики также помогают оценивать архитектуру на каждом спринте, чтобы убедиться, что она не скатывается в состояние,

которое невозможно поддерживать. Кроме того, они помогают сравнивать архитектуры, чтобы выбрать ту, которая лучше всего подходит под требования конкретного проекта.

Условные обозначения

В этой книге используются следующие условные обозначения:

Курсив
: Курсивом выделены новые термины, URL-адреса, адреса электронной почты, имена и расширения файлов.

`Моноширинный шрифт`
: Используется для листингов, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные сред, операторы и ключевые слова.

У этой книги есть веб-страница, где мы публикуем списки опечаток, примеры и любую дополнительную информацию. Страница доступна по адресу https://oreil.ly/software-architecture-metrics.

Если у вас возникнут вопросы технического характера, касающиеся содержания книги, направляйте их по электронной почте на адрес bookquestions@oreilly.com.

Благодарности

Эта книга не была бы написана без вклада авторов, редакторов O'Reilly и компании Apiumhub, которая нас всех объединила. Отдельные слова благодарности:

- руководителю отдела маркетинга Apiumhub Екатерине Новосельцевой за управление процессом написания и публикации книги совместно с O'Reilly, а также за написание введения;
- старшему редактору по закупкам O'Reilly Мелиссе Даффилд (Melissa Duffield) за заботу и за то, что благодаря ей взаимодействие с O'Reilly прошло гладко и позитивно;
- редактору по развитию O'Reilly Саре Грей (Sarah Grey) за структурирование текста книги, который стало проще читать;
- производственной команде O'Reilly: Кэтрин Тоузер (Katherine Tozer), Адаму Лоуренсу (Adam Lawrence), Стиву Фентону (Steve Fenton), Гре-

гори Хайману (Gregory Hyman) и Кристену Брауну (Kristen Brown) — за редакторскую правку и распространение книги.

От Кристиана Чичери

Выражаю благодарность Екатерине Новосельцевой и компании Apiumhub (https://apiumhub.com) за предоставленную возможность написать эту книгу, идею которой я давно вынашивал. Организаторам Global Software Architecture Summit (https://gsas.io) — за знакомство с архитекторами ПО, которые вдохновляют меня и инициируют интересные дискуссии. Приложению VYou (https://www.vyou-app.com/en) — за то, что оно мотивирует придумывать что-то новое и создавать новые метрики архитектуры ПО. И отдельное спасибо моему коту, который всегда рядом и поддерживает меня в любой ситуации.

От Дэвида Фарли

Благодарю команду Apiumhub и O'Reilly за то, что собрали нас всех вместе и организовали работу над этой книгой.

От Нила Форда

Спасибо Екатерине и коллегам из Apiumhub за организаторскую работу, благодаря которой этот проект стал реальностью. Спасибо моей компании Thoughtworks и всем ее сотрудникам за их неизменную страсть к технологиям и вовлеченность в их мир. И конечно, спасибо моей жене Кэнди за терпение к моему увлечению писательством, из-за которого я вынужден отрываться и от нее, и от наших подрастающих котят.

От Эндрю Хармел-Лоу

Благодарю свою жену и детей за терпение, а коллег из Thoughtworks — за вдохновение и возможность полностью сформировать этот архитектурный подход.

Без возможности применить все это на практике в компании, которая меня поняла и мне доверилась, рекомендации из этой книги могли бы так и остаться теоретическими. Такой компанией стала Open GI (https://opengi.co.uk), SaaS-провайдер, специализирующийся на рынке страхования в Великобритании и Ирландии. Спасибо всем, с кем я там работал. Отдельная благодарность моему коллеге-разработчику и единомышленнику Питу Хантеру (Pete Hunter). Он на лету вник в суть дела, стал его активным сторонником, постоянно вносил улучшения (мы работали в паре на каждом этапе) и многому научил меня в части практической реализации.

И наконец, спасибо Екатерине и Apiumhub за приглашение к участию, неустанные напоминания и терпеливые ответы на все мои глупые вопросы.

От Майкла Килинга

Я искренне благодарю Анастаса Стояновски (Anastas Stoyanovsky), Колина Дина (Colin Dean), Джорджа Фэйрбэнкса (George Fairbanks), Джо Рунде (Joe Runde) и Рикки Котермански (Ricky Kotermanski) за рецензирование первых черновиков моей главы. Также спасибо всем моим коллегам, нынешним и бывшим, с которыми мне посчастливилось работать. Результаты практической работы, подобные тем, что собраны в этой книге, рождаются только в командах, готовых рисковать и пробовать новое. Продолжайте в том же духе!

Мэри, моя королева, спасибо, что помогала выкраивать время для этого проекта. Оуэн, спасибо. Финну — saaanoot!

От Каролы Лилиенталь

Я выражаю благодарность всем выдающимся ИТ-специалистам и ученым в сфере computer science, с которыми мне посчастливилось работать на протяжении карьеры. Многие из них — мои коллеги из компании WPS (Workplace Solutions) или те, кого я встречала на конференциях и у кого учусь, посещая лекции и участвуя в обсуждениях. Также я благодарна своей семье, которая всегда поддерживает меня, когда мне нужно написать очередную книгу или статью.

От Жоао Розы

Ни один из моих проектов не был бы реализован без поддержки моей жены Кэри. Ты и наша прекрасная кроха — смысл моей жизни. Спасибо! Отдельная благодарность компании Xebia за поддержку на этом пути. Жажда обмена знаниями — у нас в ДНК. Также я хотел бы поблагодарить наших технических рецензентов: Рута Малана (Ruth Malan), Анну Шипман (Anna Shipman), Стива Перейру (Steve Pereira) и Ника Тьюна (Nick Tune); компанию Apiumhub за то, что мотивировали меня написать главу; а также Фая Фунга (Fai Fung), Тейса Весселинка (Thijs Wesselink) и Кенни Баас-Швеглера (Kenny Baas-Schwegler) за рецензирование первых черновиков моей главы. В заключение хочу сказать тем, кого я не упомянул: у меня ужасно дырявая память, и я просто не смог вспомнить все ваши имена. В любом случае я благодарен вам за ваш вклад в мое профессиональное развитие.

От Александра фон Цитцевица

Я благодарю свою жену Шармэн, сыновей и великолепную команду hello2morrow за неустанную поддержку моих проектов, мудрость, ценные советы и огромное терпение. Без вашей неослабевающей помощи моя работа над этой книгой, равно как и другие жизненные достижения, были бы невозможны.

От Рене Вайсса

Для меня участие в создании книги превратилось в нечто особенное. Мне посчастливилось работать со многими людьми, от которых я черпал свое вдохновение. Особо хочу отметить двоих, которые кардинально повлияли на мое понимание архитектуры ПО: Штефана Тота (Stefan Toth) и Штефана Цернера (Stefan Zörner) из компании embarc (Германия) — блестящих архитекторов, тренеров и коучей. Именно они познакомили меня с концепцией эволюционной архитектуры; из этого «зерна» и выросли идеи, изложенные в моей главе. Если вам представится возможность встретиться с ними на конференции или прочитать их книги (пока они доступны только на немецком), очень рекомендую ее использовать.

Отдельная благодарность моей девушке и партнеру Анне. Анна всегда поддерживала мои профессиональные поиски и идеи; без нее я не достиг бы того, что имею сегодня. Спасибо.

От Эойна Вудса

Хотел бы поблагодарить свою семью за постоянную поддержку всех моих профессиональных начинаний, которые требовали очень много времени. Я также благодарен Крису Куперу-Бланду (Chris Cooper-Bland) и Нику Розански (Nick Rozanski) за их развернутый и интереснейший анализ первых версий моей главы — он помог мне значительно ее улучшить. Наши технические рецензенты и превосходная команда издательства O'Reilly сделали книгу намного лучше, поэтому спасибо и всем им тоже. Наконец, спасибо моим коллегам из Endava, которые создали такую дружескую рабочую атмосферу и при этом постоянно бросали мне вызов, побуждая работать над собой и самосовершенствоваться.

ГЛАВА 1

Четыре ключевые метрики в действии

Эндрю Хармел-Лоу

Нет ничего зазорного в том, чтобы считать, что новаторская книга «Accelerate»[1] (изд-во IT Revolution Press, 2018) доктора Николь Форсгрен (Nicole Forsgren), Джеза Хамбла (Jez Humble) и Джина Кима (Gene Kim) — это исчерпывающее руководство по повышению производительности доставки ПО, и эта производительность измеряется четырьмя простыми, но емкими ключевыми метриками.

Кроме того, моя собственная работа над трансформацией во многом основана на рекомендациях из этой книги, и я полностью согласен с ее содержанием. Однако же, на мой взгляд, книга не отменяет потребность в более глубоком изучении, а, наоборот, должна стать предметом дальнейшего обсуждения и анализа. Это позволит обмениваться опытом и объединить сообщество архитекторов, стремящихся к улучшениям. Я надеюсь, что эта глава будет способствовать развитию дискуссии.

Я убедился, что если команда использует четыре ключевые метрики — частоту развертываний (deployment frequency), время выполнения доставки (lead time for changes), долю неуспешных изменений (change failure rate) и время восстановления сервиса (time to restore service)[2] —

[1] Форсгрен Н., Хамбл Д., Ким Дж. «Ускоряйся! Наука DevOps. Как создавать и масштабировать высокопроизводительные цифровые организации».

[2] Перевод дается по книге «Ускоряйся». В исследовании состояния DevOps в России (см. https://express42.com/state-of-devops/) эти метрики называют частотой развертываний, сроками поставки, неуспешными изменениями и временем восстановления соответственно. — *Примеч. науч. ред.*

описанным далее в главе способом, то это помогает ей обучаться и понять, насколько необходима высококачественная, слабосвязанная и модульная архитектура, которую просто развертывать, тестировать, наблюдать и поддерживать. Грамотное применение этих метрик позволяет архитектору ослабить контроль. Вместо диктата и жесткого управления вы можете использовать четыре ключевые метрики, чтобы вовлекать команду в диалог и стимулировать ее улучшать программную архитектуру как общее, а не личное дело. Такой подход позволяет постепенно продвигаться к связной, целостной, модульной и отказоустойчивой архитектуре, которая нативна для облачных сред и которую проще тестировать, запускать и контролировать.

В следующих разделах я покажу, как запустить эти четыре ключевые метрики, и, что важнее, как вам и вашим командам разработчиков эффективнее всего использовать их, чтобы сосредоточиться на непрерывном улучшении процессов и отслеживании прогресса. Основное внимание я уделю практическим вопросам:

- визуализации схемы с четырьмя ключевыми метриками;
- сбору трех необходимых исходных точек данных;
- расчету и отображению самих метрик.

Кроме того, я расскажу о преимуществах архитектуры, работающей на продакшен.

Определение и сбор данных

> *Парадигмы — это источники систем. Из них, из общих соглашений внутри общества о природе реальности, возникают цели системы, потоки информации, обратные связи, запасы, потоки и все остальное, что касается систем.*
>
> *Донелла Медоуз (Donella Meadows), Thinking in Systems: A Primer*[1]

Четыре ключевые метрики вытекают из схемы, лежащей в основе упомянутой выше книги «Accelerate». Я начинаю с этого утверждения, потому что при чтении этой главы базовую схему важно держать в голове. В своей простейшей форме модель представляет собой пайплайн (или «поток»)

[1] Медоуз Д. «Системное мышление. Как создавать и улучшать системы в бизнесе и жизни».

действий, который запускается, когда разработчик отправляет изменения кода в систему контроля версий, и завершается, когда эти изменения поглощаются работающей системой, доставляя работающий сервис пользователям. Эта схема представлена на рис. 1.1.

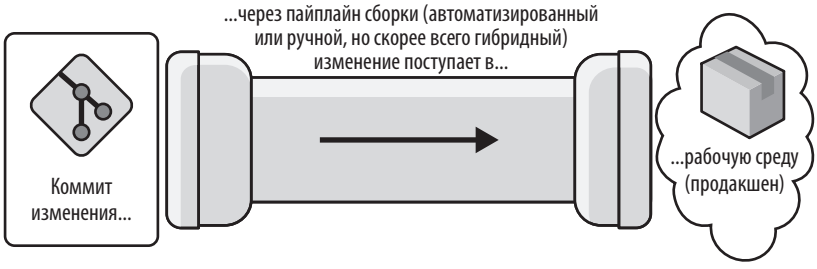


Рис. 1.1. Базовая схема, лежащая в основе четырех ключевых метрик

Для ясности визуализируем, что именно измеряют четыре ключевые метрики в рамках этой модели:

Частота развертываний (Deployment frequency)

Количество отдельных изменений, которые выходят из пайплайна за единицу времени. Эти изменения могут представлять собой «единицы развертывания»: код, конфигурацию или все вместе, в том числе, например, новый функционал или исправление дефекта.

Время выполнения доставки (Lead time for changes)

Время, которое требуется для того, чтобы завершенные разработчиком изменения в коде или конфигурации прошли через весь пайплайн.

В сочетании эта первая пара метрик позволяет измерять *пропускную способность разработки* (development throughput). Ее не следует путать с циклом разработки по методологии Lean или общим временем выполнения (lead time), включающим время на написание кода, отсчет которого иногда начинается с момента, когда продакт-менеджер впервые придумывает идею для новой фичи.

Доля неуспешных изменений (Change failure rate)

Доля изменений, выходящих из пайплайна, которые вызывают сбой в работающем сервисе. (Конкретные критерии, определяющие «сбой», будут рассмотрены далее. Пока что просто считайте сбоем все, что мешает пользователям вашего сервиса выполнять свои задачи.)

Время восстановления сервиса (Time to restore service)

Время, которое требуется после того, как сервис столкнулся со сбоем, чтобы осознать проблему и доставить исправление, которое восстановит работу сервиса для пользователей[1].

Вместе эта вторая пара метрик описывает *стабильность сервиса*.

Сила этих четырех ключевых метрик — в их сочетании. Если вы повышаете пропускную способность разработки, но при этом снижаете стабильность сервиса, то ваше улучшение неравномерно и вряд ли полученное преимущество окажется долгосрочным и устойчивым. Важно следить за всеми четырьмя ключевыми метриками. Трансформации с предсказуемой и долгосрочной ценностью — это такие, которые положительно влияют *на все метрики*.

Итак, мы узнали, откуда берутся наши метрики. Усложним картину, сопоставив общую схему с реальным процессом доставки ПО в организации. В следующем разделе я покажу, как выполнить этот «рефакторинг схемы».

Рефакторинг схемы

Важно дать определение каждой метрики *для конкретных условий*. Как вы, наверное, догадались, первые две метрики измеряют то, что происходит в CI-пайплайнах, а вторая пара отслеживает простои сервиса и его восстановление.

Будьте предельно внимательны, когда будете определять область применения при рефакторинге схемы. Какие изменения вы анализируете — для всего ПО в организации или только те, что относятся к вашей части ПО? Учитываете ли вы изменения всей инфраструктуры или только те, что происходят в ПО и сервисах? Все эти варианты возможны, но следует помнить, что *область применения должна быть одинакова для всех четырех метрик*. Если при расчете времени выполнения изменений и частоты развертываний вы учитываете изменения инфраструктуры, то учитывайте и сбои, вызванные этими изменениями.

---

[1] Исправление не обязательно означает код. Поскольку речь идет о возобновлении работы сервиса в принципе, то для остановки отсчета времени восстановления вполне подходит и, к примеру, автоматическое переключение на резервный узел.

Пайплайны как первая отправная точка

Какие пайплайны следует брать в расчет? Те, которые:

- следят за изменениями кода и конфигурации в репозитории исходного кода в рамках выбранной области применения;
- в ответ на эти изменения выполняют различные действия (например, компиляцию, автоматическое тестирование и сборку);
- развертывают результаты на продакшен.

Не стоит учитывать задачи непрерывной интеграции, которые выполняются для других целей, например резервное копирование баз данных.

Если в вашей организации один репозиторий кода обслуживается единым сквозным пайплайном (например, монолитное приложение в монорепозитории, которое собирается и напрямую развертывается на продакшен одним набором действий), то задача значительно упрощается. Эта модель показана на рис. 1.2.

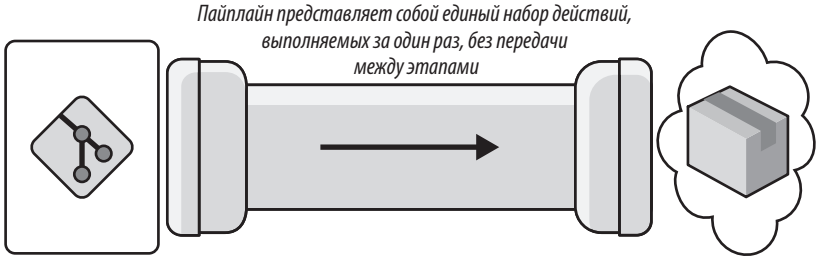


Рис. 1.2. Простейшая модель «система управления версиями — пайплайн — развертывание»

К сожалению, хотя эта модель в точности соответствует нашей базовой схеме, на практике я редко с ней сталкивался. Скорее всего, чтобы схема отражала ваши реалии, потребуется более глубокий ее рефакторинг.

Следующий по простоте измерения и первый значительный шаг в рефакторинге схемы — это набор сквозных пайплайнов, по одному на каждый артефакт или репозиторий (например, на каждый микросервис), где каждый пайплайн выполняет всю необходимую работу и завершается развертыванием на продакшен (рис. 1.3). Например, если вы используете Azure DevOps, создать такие пайплайны несложно[1].

[1] Более того, именно эту модель рекомендует использовать Microsoft.

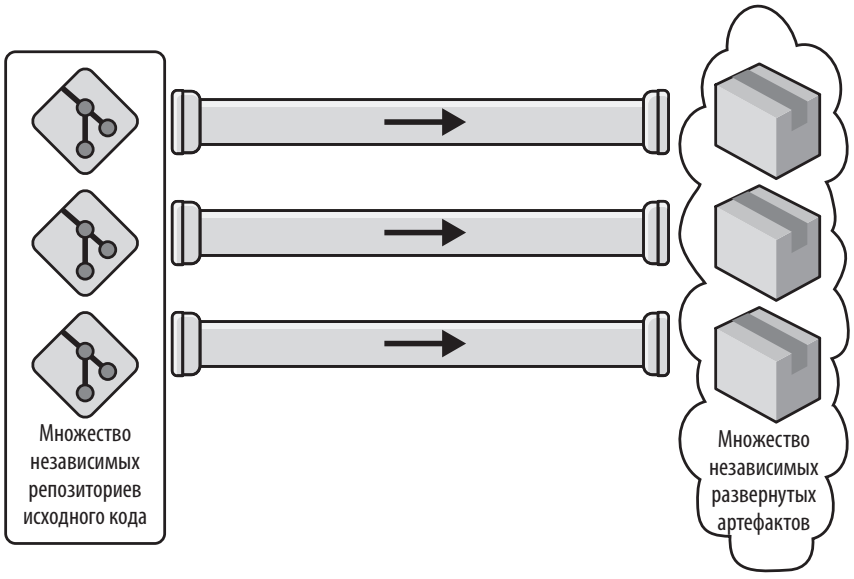


Рис. 1.3. Модель «множества сквозных пайплайнов» идеально подходит для микросервисов

Эти первые две конфигурации пайплайнов, вероятно, *похожи* на ваши, но я полагаю, что ваша картина будет несколько сложнее и потребует еще одного рефакторинга для разделения на серию суб-пайплайнов (рис. 1.4). Рассмотрим пример с тремя такими суб-пайплайнами, которые соединяются в сквозной процесс для доставки изменения на продакшен.

Первый суб-пайплайн может отслеживать пуши в репозитории, выполнять компиляцию, упаковку, модульное и компонентное тестирование, а затем публиковать результат в репозиторий бинарных артефактов. За ним может следовать второй, независимый суб-пайплайн, который развертывает этот новый артефакт в одну или несколько тестовых сред. Третий суб-пайплайн, который может запускаться процессом, подобным CAB[1], в итоге развертывает изменение на продакшен.

[1] Аббревиатура CAB расшифровывается как «change-advisory board» (совет по утверждению изменений). Самый известный пример такой группы, которая регулярно собирается для утверждения релизов кода и конфигурации, описан в ставшей классической книге Джина Кима, Кевина Бера (Kevin Behr) и Джорджа Спаффорда (George Spafford) «The Phoenix Project» (IT Revolution Press, 2018, https://oreil.ly/b5609) («Проект “Феникс”. Как DevOps устраняет хаос и ускоряет развитие компании»).

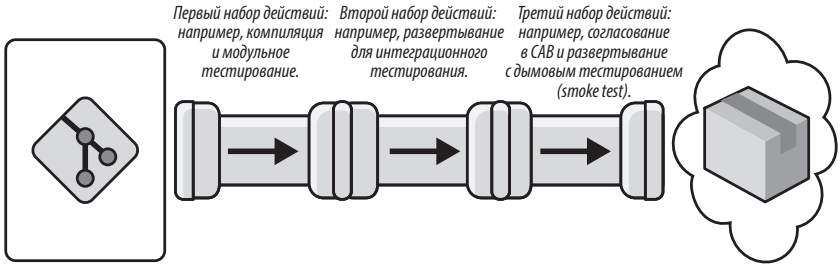


Рис. 1.4. Модель «пайплайн из множества суб-пайплайнов» — это та, которую я встречаю чаще всего

Надеюсь, вы выбрали подходящий для вас вариант. Если же нет, то есть еще и четвертая основная разновидность пайплайнов, к которой мы придем на последнем шаге рефакторинга схемы: многоступенчатый пайплайн слияния (fan-in), показанный на рис. 1.5. Для первой стадии в этой модели обычно используются отдельные суб-пайплайны (по одному на репозиторий), которые затем «сходятся» к общему суб-пайплайну или набору суб-пайплайнов, доводящих изменение до продакшена.

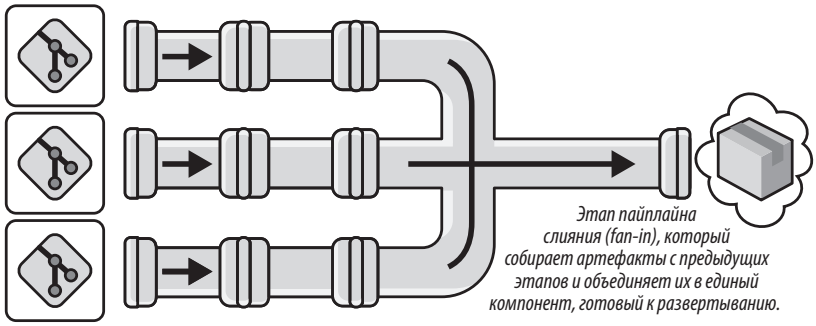


Рис. 1.5. Модель многоступенчатого пайплайна слияния (fan-in)

Определение точек сбора данных

Помимо четырех метрик, у нас есть четыре точки сбора данных. Определим их расположение в нашей схеме, какой бы она ни была. До сих пор мы говорили о пайплайнах, поскольку они обычно предоставляют две из этих точек: метку времени коммита и метку времени развертывания. Третья и четвертая точки сбора данных берутся из меток времени, создаваемых в момент обнаружения деградации сервиса и в момент, когда эта деградация помечается как «устраненная». Теперь обсудим подробнее каждую из них.

Метка времени коммита

При рассмотрении практики работы команд неизбежно возникают тонкие нюансы. Используют ли команды ветвление по фичам? Применяют ли пул-реквесты? Может быть, они используют разнообразные подходы? В идеале (как предлагают авторы «Accelerate») отсчет времени должен начинаться в момент, когда любой набор изменений разработчика считается завершенным и сделан его коммит — независимо от того, где это происходит. Если команды работают именно так, будьте внимательны: удержание изменений в ветках не только удлиняет циклы обратной связи, но и создает дополнительные накладные расходы, а также предъявляет повышенные требования к инфраструктуре. (Об этом я расскажу в следующем подразделе.)

Из-за этой специфики некоторые команды предпочитают использовать запуск пайплайна после слияния в основную ветку (main) в качестве опосредованной точки отсчета или метки времени коммита. Я понимаю, что такая точка отсчета может выглядеть как «костыль»[1], но если вы ее выберете в качестве триггера, будьте, кроме того, готовы к угрызениям совести — за то, что отклоняетесь от лучших принятых практик. Независимо от того, учитываем мы это дополнительное время ожидания или нет, метрики принесут множество других преимуществ, даже если вы позволите себе упростить процесс и начнете отсчет, когда код попадет в основную ветку (main). Если (и когда) эти методы действительно значительно ухудшат процесс доставки, в книге «Accelerate» вы найдете другие рекомендуемые подходы (например, парное программирование и магистральная разработка[2] (TBD, trunk-based development)[3], которые влияют на определение метки времени коммита, предлагая брать за старт таймера время попадания изменения в основную ветку. К тому моменту вы уже начнете видеть пользу от метрик и захотите усовершенствовать процесс их сбора.

Метка времени развертывания

Разобравшись с меткой времени коммита, вы вздохнете с облегчением, узнав, что момент «остановки» часов определить гораздо проще: это за-

---

[1] Особенно если вы используете долгоживущие ветки или пул-реквесты, которые подолгу не закрываются, но я уверен, что вы и так о них знаете, и измерить их влияние по отдельности тоже несложно.

[2] Говорят также «ветвление по модели TBD». — *Примеч. науч. ред.*

[3] См. документацию (https://oreil.ly/L5cs0), чтобы узнать все необходимое о TBD. Исходное определение парного программирования можно найти на ресурсе, посвященном экстремальному программированию (https://oreil.ly/pGAfY).

вершение работы пайплайна, выполняющего финальное развертывание на продакшен. Но нет ли тут некоторой поблажки тем, кто проводит ручное дымовое тестирование уже *после* развертывания? Да, возникает, но, повторюсь, это остается на вашей совести. Если же вам действительно необходимо это финальное действие, вы всегда можете добавить контрольную точку в конце пайплайнов, которую вручную зафиксирует QA-инженер или другой специалист, отвечающий за проверку развертываний, как только убедится в успешности развертывания.

Сложности, характерные для многоэтапных пайплайнов и пайплайнов слияния

Имея эти два источника данных, можно вычислить нужную нам информацию о работе пайплайнов, а именно *общее время выполнения*: интервал между запуском и остановкой таймера. Если у вас реализован один из простейших сценариев работы пайплайна, рассмотренных ранее, — тех, где не используется слияние потоков изменений, — то это относительно легко. Проще всего тем, у кого развернут один или несколько сквозных пайплайнов[1].

Если вам не повезло и у вас есть несколько суб-пайплайнов (как показано на рис. 1.4), вам потребуется собрать дополнительные данные для набора(-ов) изменений, которые были включены в метку времени «старта» и «развертывание» в продакшен. Имея эти данные, вы можете выполнить обработку для расчета общего времени выполнения каждого отдельного изменения.

Если вы используете пайплайн слияния (как на рис. 1.5), эта обработка, вероятно, будет сложнее. Почему? Как видно на рис. 1.6, вам нужен способ определять, откуда берется номер развертывания для изменения 264 — из Репозитория A, Репозитория B или Репозитория C), — чтобы получить метку времени «старта» для этого изменения. Если ваше развертывание объединяет несколько изменений, вам придется отследить каждое из них отдельно, чтобы получить его время «старта».

Очевидно, что в любом случае, независимо от сложности ваших пайплайнов, учитывать следует только те сборки, которые развертывают обнов-

---

[1] Если это соображение навело вас на мысль, что наличие нескольких независимых пайплайнов — по одному на артефакт — это хорошо, примите мои поздравления: вы сами себе напомнили об одном из ключевых принципов микросервисов — независимости развертывания. Если же оно заставляет вас тосковать по монолиту, вспомните о других преимуществах микросервисов, часть из которых мы рассмотрим в конце этой главы.

ления сервисов для пользователей. Убедитесь, что вы измеряете именно такие[1].

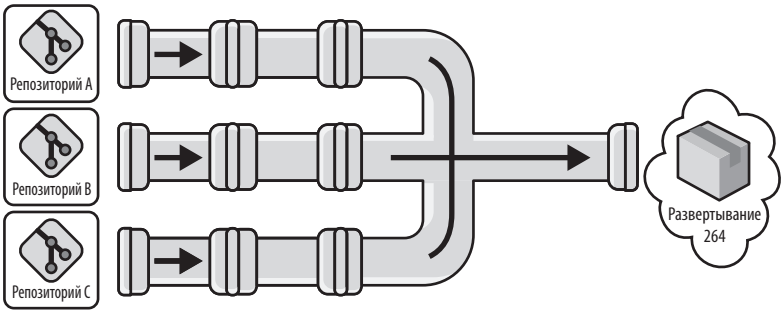


Рис. 1.6. Определение точек сбора данных в модели «пайплайна слияния»

Перед тем как идти дальше, стоит затронуть еще один, заключительный вопрос сбора данных из пайплайнов: какие именно запуски пайплайнов учитывать? «Accelerate» снова не дает четких указаний на этот счет, но для вас должны быть важны только успешные запуски. Если сборка начинается, но падает на этапе компиляции, это искусственно улучшит ваши показатели времени выполнения, поскольку в статистику попадет очень «быстрая» сборка. Если вам захочется манипулировать метриками (а ключевое достоинство четырех основных метрик состоит как раз в том, что ими, насколько мне известно, нельзя манипулировать), то достаточно просто отправлять множество заведомо сломанных сборок, желательно таких, которые завершатся *очень* быстро.

Мониторинг сбоев сервисов

Хотя измерения, связанные с пайплайнами, выполнять относительно просто, третий и последний источник исходных данных можно интерпретировать гораздо шире.

Основная сложность заключается в определении самого понятия «сбой на продакшен». Если сбой произошел, но его никто не обнаружил, можно ли считать, что он вообще был? При работе с четырьмя ключевыми

---

[1] Иногда спрашивают: «А как быть со сборками инфраструктуры?» Их изредка включают в расчет «четырех ключевых метрик», но я бы не стал настаивать на их обязательном учете. А как насчет пайплайнов, запускаемых по расписанию, а не по изменению? Не учитывайте их. Поскольку по итогу их работы ничего не меняется, они не приводят к развертыванию.

метриками я всегда отвечаю на этот вопрос отрицательно. Я определяю «сбои на продакшен» как любые ситуации, когда потребитель сервиса становится неспособным или даже теряет желание оставаться в системе для завершения начатой задачи. Косметические дефекты сбоями сервиса не считаются, но если «работающая система» работает настолько медленно, что вызывает аномальный отток пользователей, то она явно переживает сбой. Здесь присутствует элемент субъективной оценки, и это нормально: выберите определение, которое вас устраивает, и последовательно его придерживайтесь, оставаясь честным перед собой.

Теперь необходимо фиксировать сбои сервисов — это третья и последняя точка сбора данных, которую обычно реализуют с помощью тикета «сбой изменения». Создание такого тикета дает метку времени начала нового отсчета, а его закрытие — соответствующую метку окончания. Время начала, время окончания и количество тикетов — это все точки исходных данных, которые потребуются для дальнейших расчетов. Тикет следует закрывать, когда работа сервиса восстановлена. Восстановление может не означать устранение первопричины сбоя, и это нормально, поскольку важно обеспечить стабильность сервиса. Откат изменений, чтобы вернуться в онлайн и обслуживать клиентов, является приемлемым решением в данном контексте[1].

Но как быть, если у вас нет продакшен-среды? Во-первых, разве вы еще не пытались начать переход к непрерывному развертыванию? Уже явно пора. Во-вторых, продакшен-среда доступна не всем. Хотя это и не очень хорошо, но даже в таких условиях можно использовать четыре ключевые метрики. Для этого необходимо определить наивысший доступный уровень вашей среды, то есть той общей среды, в которую доставляются все изменения команды и которая ближе всего к продакшену. Вероятно, она называется средой системной интеграции, предпродакшен или промежуточной средой (стейджинг). Главное, что, принимая изменения для этой среды, вы исходите из того, что для их переноса на продакшен дополнительно ничего делать не нужно.

---

[1] Кто-то опять укажет, что время открытия тикета не совпадает с моментом возникновения сбоя сервиса. Это верно. Возможно, вам стоит привязать систему мониторинга к созданию таких тикетов, чтобы обойти эту проблему. Если у вас есть такая возможность, поздравляю: вы, вероятно, находитесь на этапе «тонкой настройки» внедрения четырех ключевых метрик. Большинству команд, особенно в начале, такая точность только снится, поэтому для старта будет достаточно и тикетов, заведенных вручную.

С учетом всех перечисленных соображений считайте эту среду эквивалентом продакшена, а тестировщиков и взаимодействующие команды — эквивалентом пользователей. Они сами определяют, что считать сбоем сервиса. Относитесь к сбоям в тестах так же серьезно, как к реальным. Притворяться, что эта среда — продакшен, так себе вариант, но лучше, чем ничего.

Сбор данных и расчет

> *Специалисты по системному моделированию утверждают, что мы меняем парадигмы, когда создаем модель системы. Этот процесс заставляет нас выйти за пределы системы и увидеть целое.*
>
> *Донелла Медоуз. «Thinking in Systems»*

Теперь, когда определения готовы, можно приступать к сбору данных и расчетам. Хотя желательно автоматизировать этот процесс, вполне допустимо выполнять его вручную[1]. Фактически каждый раз, внедряя четыре ключевые метрики, я начинал именно с того, что собирал их вручную, и зачастую не только чтобы получить их базовое исходное значение. Чуть ниже вы увидите, почему сбор и расчет вручную — это нормально.

Сбор метрик может быть простой или сложной задачей, в зависимости от специфики пайплайнов. Независимо от этого четыре ключевые метрики рассчитываются на основе четырех наборов данных, поступающих из четырех точек сбора данных: количества успешных развертываний изменений, общего времени выполнения пайплайна(-ов) для каждого изменения, количества тикетов на сбои изменений и времени, в течение которого эти тикеты остаются открытыми. Просто собрать данные недостаточно для получения метрик; требуется расчет. Рассмотрим каждую метрику по очереди.

Частота развертываний

Это именно частота, а не количество, поэтому необходимо общее количество успешных развертываний *за заданный период* (мне в качестве периода хорошо подошел день). Если у вас несколько пайплайнов, независимо от того, используется ли схема слияния, нужно сложить количество развертываний из них всех.

---

[1] Будьте честны с собой: собирайте данные по всем нужным сборкам, а не по избранным. Также стремитесь к максимальной точности в цифрах, а если приходится делать предположения, оценивайте степень точности.

Если эти данные записывать и подытоживать ежедневно (не забывая включать нулевые значения для дней без развертываний), легко получить первую ключевую метрику. Работа лишь с последним ежедневным значением или значением за последние 24 часа приводит (по моему опыту) к слишком большим колебаниям. Лучше всего отображать среднее значение за более длительный период, например за последние 30 дней.

Время выполнения доставки

Это время, затраченное *на любое отдельное изменение, которое запускает процесс*. Поскольку этот показатель может колебаться, не стоит заносить в отчет только лишь последнее значение из последнего развертывания. Если у вас множество пайплайнов (включая пайплайны слияния), колебания будут еще сильнее, потому что некоторые сборки выполняются *значительно* быстрее других из-за блокировок. Нужен более стабильный показатель, который отразит всю картину, а не последний выброс. Из этих соображений я обычно беру каждое отдельное значение времени выполнения и вычисляю среднее значение всех измерений за день.

Отчетным показателем является среднее значение всех значений времени выполнения за последние 30 дней[1].

Доля неуспешных изменений

Доля неуспешных изменений — это доля разрешенных тикетов по неуспешным изменениям, то есть количество развертываний, которые привели к сбоям, деленное на общее количество развертываний за тот же период. Например, если за день было 36 развертываний и в течение того же дня устранено 2 изменения, приведших ранее к сбоям, то процент неуспешных изменений за этот день составит 2/36, или примерно 5,55555556 %.

Чтобы получить показатель для отчета, нужно рассмотреть эту долю за прошедший период в 30 дней. Для этого сложите количество устраненных сбоев за последние 30 дней и разделите на общее количество развертываний за тот же период.

Здесь, как вы, возможно, заметили, есть элемент допущения. Мы исходим из того, что сбои независимы *и* каждый отдельный сбой вызван одним развертыванием. Почему? Потому что, по моему опыту, связать сбои с конкретными сборками слишком сложно, и в большинстве инци-

[1] Помните! Усреднение средних значений чревато проблемами, поэтому лучше его избегать. Мы используем ежедневные итоги, чтобы получить наглядный график метрик, к которому вернемся позже.

дентов эти два допущения выполняются — по крайней мере, достаточно, чтобы оправдать некоторую потерю точности. Если вы можете более грамотно подойти к этому вопросу — прекрасно!

Внимательный читатель также заметит, что мы говорим только об *устраненных* сбоях. Почему же мы не включаем тикеты, которые все еще открыты? Потому что необходимо сохранить согласованность между всеми четырьмя метриками и потому что для расчета времени восстановления сервиса можно учитывать только устраненные сбои[1]. Если для одного показателя нельзя учитывать нерешенные сбои, то не стоит учитывать их и для другого. Но не стоит беспокоиться: данные об открытых тикетах у нас все равно есть, и мы их не скрываем, как будет видно в следующих разделах.

Время восстановления сервиса

Это время от создания до закрытия тикета о неуспешном изменении. В книге «Accelerate» этот показатель называют *средним временем восстановления сервиса*, в ранних отчетах State of DevOps — просто *временем восстановления*, а в файле `METRICS.md` (https://oreil.ly/VlSgn) проекта Google Four Keys — *медианным временем восстановления*. Я использовал *и медиану, и среднее*: среднее значение чувствительно к выбросам, что иногда полезно для анализа.

Оба показателя легко рассчитать по данным о времени разрешения тикетов о неуспешных изменениях. В любом случае нужно выбрать временной диапазон для данных — я обычно беру последние 120 дней. Затем берем все время устранения сбоев за этот период, вычисляем его среднее значение и используем его как метрику.

Здесь есть нюанс: при создании тикетов о сбоях вручную можно исказить цифры, задержав открытие тикета после обнаружения проблемы. Даже при исключительно добрых намерениях искажение неизбежно. Тем не менее имеющихся данных будет достаточно для ведения мониторинга и внесения улучшений.

Важно обеспечить прозрачность всего процесса сбора и расчета данных. Для этого, во-первых, команды разработки должны изучить четыре ключевые метрики. Не скрывайте ничего.

Во-вторых, исходные данные, расчеты и итоговые показатели должны быть доступны всем. Это пригодится в дальнейшем.

---

[1] Поскольку у неустраненных сбоев, к сожалению, нет меток «решено».

В-третьих, точные определения метрик и правила их применения должны быть доступны *вместе с данными*. Такая прозрачность улучшит понимание процессов участниками команды и повысит их вовлеченность[1].

Уделите особое внимание доступности данных, расчетов и графиков: если ключевые метрики доступны не всем, вы упускаете их главное преимущество.

Отображение и понимание данных

> *[Итак] как изменить парадигмы?.. Продолжать указывать на аномалии и сбои в старой парадигме, а говорить и действовать из новой — громко и уверенно.*
>
> *Донелла Медоуз. «Thinking in Systems»*

Всякий раз, когда я внедрял четыре ключевые метрики, я обычно начинал с *минимально жизнеспособного дашборда* (MVD, minimal viable dashboard)[2] — весьма громкое название для вики-страницы со следующими данными:

- текущие расчетные значения каждой из четырех ключевых метрик;
- определения каждой метрики и временные периоды, по которым мы их рассчитываем;
- исторические значения данных.

Я также указываю источники данных, чтобы все могли с ними работать.

Целевая аудитория

Метрики, как и вся статистика, рассказывают историю, а у историй есть аудитория. Кто же целевая аудитория для четырех ключевых метрик? В первую очередь, это команды, которые разрабатывают и поставляют ПО, — то есть люди, которые фактически будут вносить изменения, если им потребуется улучшить метрики.

---

[1] При таком подходе вы даже можете получать сообщения об ошибках в своих расчетах, если эти расчеты окажутся неверны, — и определенную часть наиболее ценных сведений о четырех ключевых метриках я получил именно таким образом.

[2] Отдельная благодарность Мэтью Скелтону (Matthew Skelton) и Мануэлю Паису (Manuel Pais) за их идею «минимально жизнеспособной платформы», которая вдохновила меня на такое решение.

Таким образом, необходимо обеспечить, чтобы данные — неважно, где и как отображаемые, — находились там, где они будут легко доступны *в первую очередь* именно этим людям и группам. Слово «легко» здесь ключевое. Разработчикам должно быть предельно просто увидеть метрики, углубиться в них и получить более детальную информацию, как правило, о тех точках данных, которые относятся к сервисам, за которые эти команды отвечают.

У четырех ключевых метрик есть и другие аудитории, но они вторичны. Одной из таких вторичных аудиторий может быть высшее руководство. Ему можно показывать метрики, но в агрегированном и доступном только для чтения виде. Если руководству потребуются более детальные данные, ему следует обратиться за ними непосредственно к командам, и это именно тот сценарий, к которому стоит стремиться.

В идеале, как только ваш минимально жизнеспособный дашборд будет запущен, можно приступать к автоматизации сбора и расчета данных. На момент написания книги существует несколько инструментов автоматизации. Возможно, вы остановитесь на Four Keys (https://oreil.ly/BPRaw) от Google, Metrik (https://oreil.ly/1EDTb) от Thoughtworks или на каком-то расширении для платформ, например Azure DevOps (https://oreil.ly/vMSBR). Я лично не использовал ни один из этих инструментов и, хотя уверен, что все они подходят для наших целей, я поделюсь преимуществами опыта создания решения вручную. Это поможет вам оценить, стоит ли использовать готовый продукт или все же потратить время и силы на разработку собственного.

Визуализация

Одним из моих реализованных проектов стал наиболее полнофункциональный дашборд из всех, с которым мне когда-либо доводилось работать. Он был построен на Microsoft PowerBI, поскольку клиентское приложение полностью работало в экосистеме Azure DevOps. Попотев немного над обработкой дат и времени, мы собрали исходные данные, выполнили расчеты и приступили к созданию графиков и других элементов визуализации.

Частота развертываний

Для визуализации этих данных мы выбрали столбчатую диаграмму (рис. 1.7), где по оси X отложены даты, а по оси Y — количество развертываний. Каждый столбец показывает итог за день, а ключевая статистика вынесена в отдельные сводки.

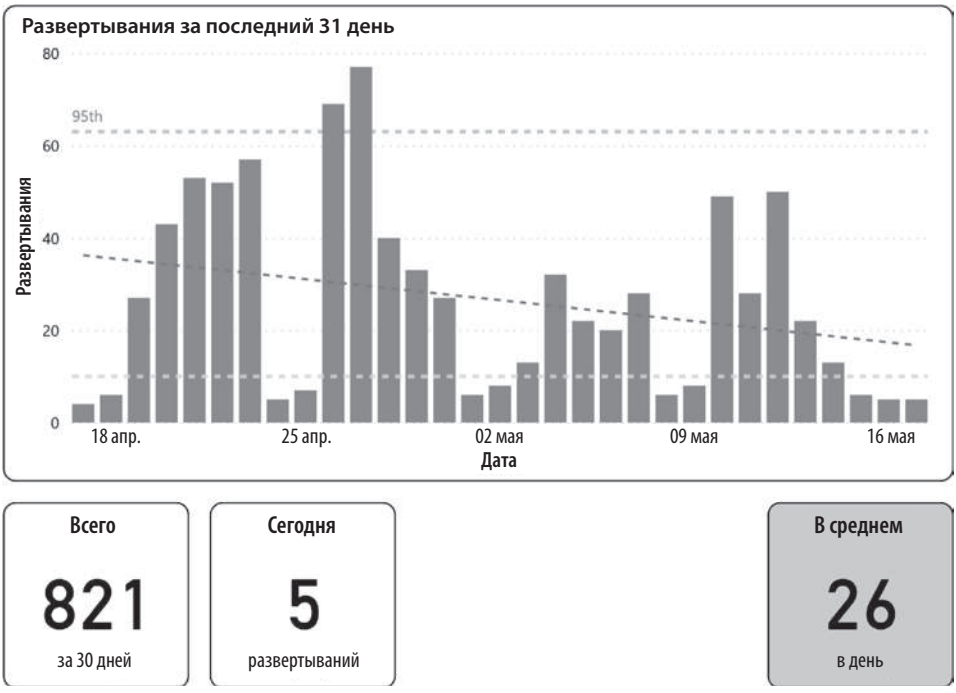


Рис. 1.7. Частота развертываний; данные в блоке в правом нижнем углу указывают на уровень DORA Elite (DevOps Research and Assessment) производительности доставки ПО

Среднее количество развертываний в день представляет собой ключевую метрику частоты, и мы выделили ее зеленым цветом, указывая на уровень Elite по шкале производительности доставки ПО из книги «Accelerate» (от Elite до Low)[1]. Для повышения прозрачности на дашборде отображались развертывания за текущий день и их общее количество за период (31 день). Кроме того, на график были нанесены пунктирными линиями среднее значение, 95-й процентиль и общие тренды изменений данных.

Время выполнения доставки

Столбчатая диаграмма на рис. 1.8 отображает данные о времени выполнения изменений: на оси X снова даты, а столбцы по оси Y показывают *среднее* время выполнения за соответствующий день.

[1] См. рис. 2.2 и 2.3 в книге «Accelerate», а также более актуальные таблицы в последнем отчете DORA State of DevOps (https://oreil.ly/IetZp).

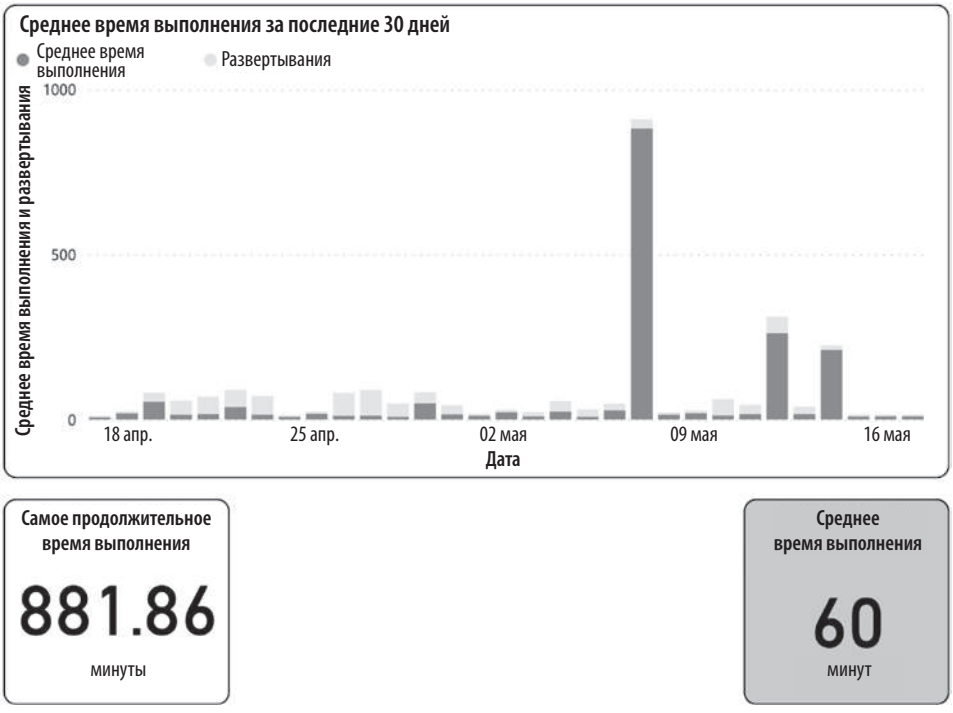


Рис. 1.8. Время выполнения доставки; данные в блоке в правом нижнем углу указывают на уровень DORA High производительности доставки ПО по шкале оценки Accelerate

Ключевая метрика на этом экране — среднее время выполнения за период — выделена, как и на предыдущем, а в сводных показателях отмечен уровень Low по шкале Elite-Low. Также мы сочли полезным выделить самое длинное отдельное время выполнения (блок в левом нижнем углу)[1].

Поскольку мы постоянно задавались вопросом, не было ли в тот день большого количества развертываний, мы вместо множества линий трендов добавили полупрозрачную линию тренда количества развертываний (светло-серого цвета) за основным графиком.

Доля неуспешных изменений

На рис. 1.9 представлена еще одна столбчатая диаграмма для метрики доли неуспешных изменений, но ее данные выглядят иначе. Как видно

[1] Например, когда сборка блокировалась, это было очевидно.

по оси Y, в течение 24-часового периода у нас обычно либо не было отказов, либо был всего один[1]. Соответственно, проблемы были *явно* заметны сразу.

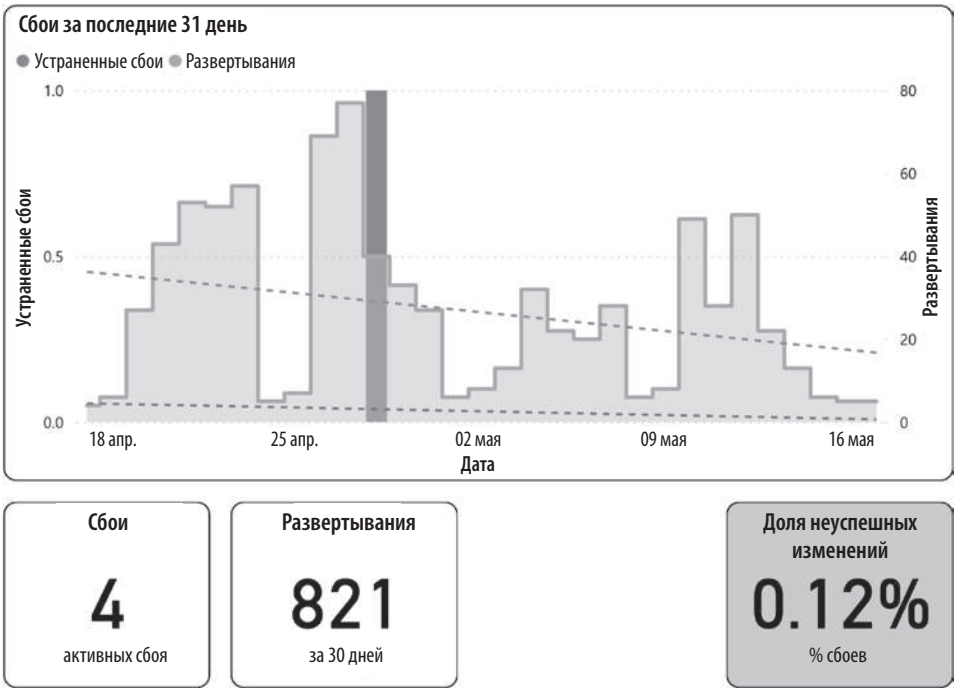


Рис. 1.9. Доля неуспешных изменений; данные в блоке в правом нижнем углу указывают на уровень DORA Elite производительности доставки ПО по шкале оценки Accelerate

Вся дополнительная информация здесь является контекстом. Совместное отображение количества развертываний позволяет быстро ответить на вопрос: «Может ли быть причиной то, что в тот день было много развертываний?»

Наконец, как обычно, внизу показана ключевая метрика: доля сбоев в процентах от общего числа развертываний за период. Рядом приведены и другие важные показатели: количество активных сбоев и общее число развертываний за указанный период.

[1] Иногда отказов было несколько, но это было очень необычно.

Время восстановления сервиса

С последней метрикой — временем восстановления сервиса — мы осваивались дольше всего, но после того как разобрались и стабилизировали метрики частоты развертываний и времени выполнения изменений, эта метрика стала для нас приоритетной[1]. Здесь мы тоже используем столбчатую диаграмму временного ряда (рис. 1.10), но значения на ней отображаются за более длительный период, чем на других (120 дней, для лучшего контекста), чтобы можно было сравнить прогресс по метрике, у которой должно быть значительно меньше точек данных. Для контекста мы снова добавили отображение времени выполнения изменений.

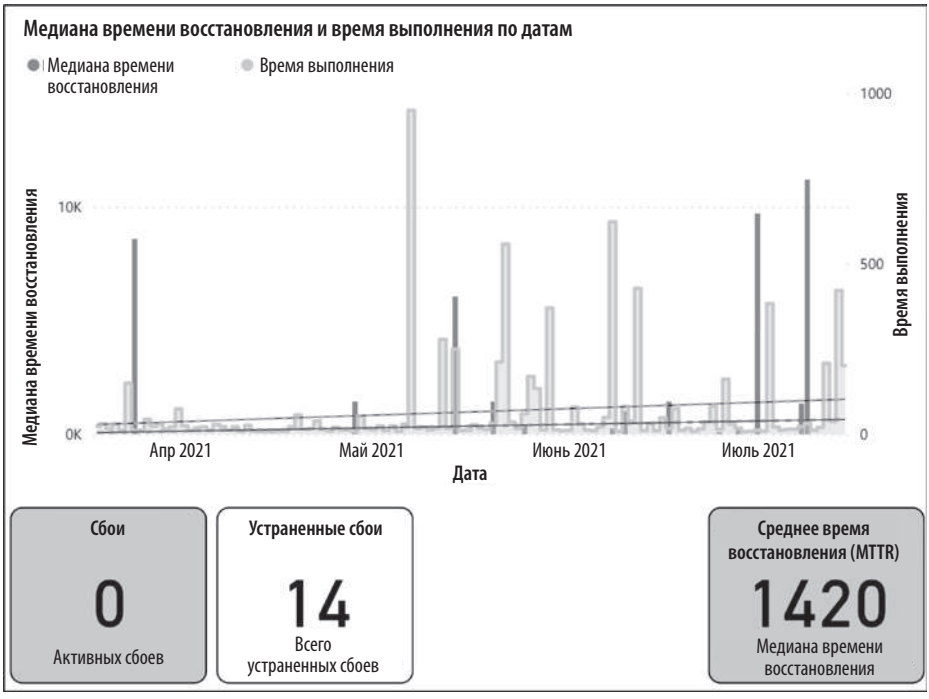


Рис. 1.10. Время восстановления сервиса; данные в блоке в левом нижнем углу указывают на отсутствие открытых сбоев изменений (это не метрика DORA, но ее важно знать), а в блоке в правом нижнем углу — на уровень DORA High производительности доставки ПО

Наконец, как обычно, в правом нижнем углу показана ключевая метрика: медиана времени восстановления для всех устраненных сбоев в заданный

[1] По своему опыту, готов поспорить, что для вас она тоже такой станет.

период. Рядом также приведены другие ключевые показатели: количество активных сбоев и общее число устраненных сбоев за указанный период.

Главная страница

На этом мы не остановились. Наш отчет PowerBI также содержал главную страницу «Четыре ключевые метрики» (Four Key Metrics). На ней были представлены ключевые числовые значения метрик с каждой отдельной страницы статистики, а также графики частоты развертываний и времени выполнения изменений. Цель была в том, чтобы быстро и точно представить статистику *в реальном времени*. По мере изменения приоритетов мы могли выводить на первый план другие графики.

Итак, мы готовы раскрыть реальную силу четырех ключевых метрик. Чтобы получать от этих метрик реальную пользу, принципиально важно предоставить командам доступ к ним и обеспечить понимание разработчиками этих метрик, а также модели и системы, внутри которых эти метрики собираются. Именно такой подход позволяет командам обсуждать, понимать, улучшать поставляемое ПО, а также выступать его владельцем.

Обсуждения и понимание

> *В процессе смены парадигмы нет ничего материального, дорогостоящего или даже медленного. У кого-то такая смена может произойти за миллисекунду. Все, что нужно, — это щелчок в сознании, падение пелены с глаз, новый способ видения.*
>
> *Донелла Медоуз. «Thinking in Systems»*

Как же мы пришли к этим визуализациям и дополнительным деталям и как определили конкретные временные периоды? Мы двигались итеративно, постепенно, внося дополнения и улучшения по мере необходимости.

Каждую неделю мы коллективно обсуждали предстоящие пики нагрузок и записи архитектурных решений (ADR, architectural decision records)[1] и анализировали четыре ключевые метрики. Сначала разговоры шли вокруг того, что означает каждая метрика. Потом, в последующие недели, — вокруг причин текущих значений (например, почему показатели слишком высокие или низкие, почему данные отсутствуют и т. д.). А затем мы перешли к тому, как эти значения улучшить. Медленно, но верно

[1] Впервые термин ADR употребил Майкл Найгард (Michael Nygard).

члены команды освоили схему с четырьмя ключевыми метриками. Очень помогла предоставленная им возможность самостоятельно получать свои данные в реальном времени и видеть только данные из своих пайплайнов (обе функции легко реализовать с помощью дашбордов PowerBI). Также помогло добавление линий трендов, а вскоре после этого — возможность просматривать периоды дольше стандартных 30 дней.

Меня крайне удивило, насколько полезными оказались эти точечно направленные, информативные и кросс-функциональные обсуждения. Прежде мне как архитектору пришлось бы одному выявлять, осмысливать, анализировать и устранять эти проблемы. Теперь же команды сами инициировали и развивали решения.

Ответственность и улучшение

Когда команды начинают брать на себя ответственность, я снова и снова наблюдаю одну и ту же картину. Сначала появляются самые простые запросы на модернизацию процессов и способов работы, например: «Можем ли мы изменить периодичность релизов?» Затем команды начинают уделять больше внимания качеству, предлагая «перенести тестирование на более ранние этапы» и «добавить больше автоматизации»[1]. Следующим шагом становятся запросы на изменение состава команд, например: «Можем ли мы перейти к кросс-функциональной (или ориентированной на потоки задач) организации?»

Компромиссы, неудачи и извлекаемые уроки неизбежны, но *процесс изменений начинает развиваться самостоятельно*. По мере того как вы все больше акцентируете внимание на *сквозном* представлении процесса и лучше понимаете его преимущества, вы будете корректировать и адаптировать свои цели и решения.

Все эти изменения быстро приводят к одному результату — они обнажают архитектурные проблемы. Возможно, эти проблемы кроются в представлении дизайна на доске. Возможно, с доской все было в порядке, но с реализацией, попавшей на продакшен, — нет. В любом случае у вас всегда будут задачи, которые нужно решать. Связанность компонентов, которая оказалась не такой слабой, как вы ожидали; границы доменов — не такими четкими, как изначально казалось; фреймворки, которые больше мешают

[1] К большому удовольствию QA-инженеров и команд эксплуатации. Я нередко наблюдал, как QA-инженеры используют четыре ключевые метрики для инициирования изменений — в той же степени, что и я в роли архитектора.

командам, чем помогают; модули и инфраструктура, которые не так легко тестировать, как хотелось бы; микросервисы, которые при работе с реальным трафиком оказалось невозможно наблюдать, и т. д. Все это типичные проблемы, с которыми приходится иметь дело ответственному архитектору.

Заключение

Перед вами встает выбор: продолжать действовать в одиночку, держать руки на штурвале и управлять архитектурным кораблем в меру своих сил, будучи единоличным капитаном, или воспользоваться расцветом возможностей внутри команд. Убрать руки со штурвала — может быть, не сразу, а постепенно — и использовать беседы и мотивацию, которые открывают четыре ключевые метрики, чтобы двигаться к общей цели: архитектуре, которая менее связана, более отказоустойчива, более нативна для облачных сред, которую проще тестировать, запускать и мониторить.

Именно поэтому четыре ключевые метрики входят в число самых полезных метрик архитектуры. Надеюсь, что вы будете использовать их вместе с коллегами, чтобы создать лучшую архитектуру из тех, что вам когда-либо встречалась.

ГЛАВА 2

Пирамида тестирования с использованием фитнес-функций: аналогия для тестов и метрик архитектуры

Рене Вайсс

Фитнес-функции — концепция, заимствованная из эволюционных вычислений, — это лаконичный метод, который также применяется для вычисления метрик программных систем. В этой главе вы узнаете, как с помощью фитнес-функций находить метрики, адаптированные под конкретную систему, и использовать их для улучшения ее архитектуры, независимо от того, создаете ли вы новую систему или модернизируете существующую. Комбинируя концепции фитнес-функций, метрик и пирамиды тестирования, вы сможете определять и приоритизировать метрики, находить для них баланс компромиссов, а также измерять прогресс в достижении поставленных целей.

Фитнес-функции и метрики

В книге «Building Evolutionary Architectures»[1] Нил Форд, Ребекка Парсонс (Rebecca Parsons) и Патрик Куа (Patrick Kua) определяют фитнес-функцию как «объективную функцию, используемую для оценки того, насколько разрабатываемое решение достигает поставленных целей». Обычно такая

[1] Форд Н., Парсонс Р., Куа П. «Эволюционная архитектура. Автоматизированное управление программным обеспечением», 2-е межд. изд. СПб., издательство «Питер».

функция возвращает дискретное значение — ту самую метрику, которую нужно достичь или улучшить. Чтобы определить, достигнута ли цель, необходим тест или механизм проверки, измеряющий требуемую метрику. В идеале этот процесс следует автоматизировать, но это не обязательное условие для работы с фитнес-функциями!

Будучи очень гибкими, фитнес-функции удобны для создания целевых метрик. С их помощью также можно описывать и интегрировать стандартные метрики (например, покрытие кода или метрики структуры кода, такие как цикломатическая сложность)[1], но их универсальность позволяет адаптировать архитектурные метрики под конкретную систему и контекст.

Связь между фитнес-функцией и целевой метрикой показана на концептуальной карте (рис. 2.1). Фитнес-функция определяет целевую метрику и описывает соответствующий контекст, который я называю *контекстом фитнес-функции*. В него может входить дополнительная информация о среде, определениях и ограничениях, влияющих на тестирование; позже я разобью эту информацию на категории и рассмотрю подробнее.

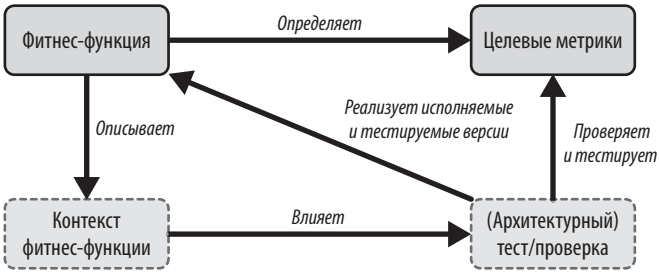


Рис. 2.1. Концептуальная карта фитнес-функции

Архитектурный тест формирует целевую метрику. Зачастую такие тесты также напрямую проверяют, чтобы полученная метрика превышала (или не превышала) заданный порог. Как правило, эти тесты автоматизированы и выполняются в рамках процессов непрерывной интеграции (CI, continuous integration). Некоторые из них можно даже запускать и проверять непрерывно, без специального триггера. Я ис-

[1] Цикломатическая сложность — это широко используемая метрика кода, которую обычно получают с помощью инструментов статического анализа кода. Она была разработана Томасом Дж. Маккейбом (Thomas J. McCabe) в 1976 году. Высокое значение этой метрики указывает на то, что код трудно понять и, возможно, изменить.

пользую термины «*архитектурные тесты*» (tests) и «*архитектурная проверка*» (verification), чтобы четко отделить их от функциональных тестов. Например, функциональный тест может проверять, корректно ли система создает нового клиента; архитектурный же тест может проверять, способна ли система создать 10 клиентов, одновременно достигая целевого показателя с точки зрения ее архитектуры или качества. Например, архитектурный тест в таком сценарии измеряет, насколько быстро были созданы эти 10 учетных записей пользователей и уложилось ли время выполнения в 10 миллисекунд.

Фитнес-функция, ее контекст и фактическая целевая метрика тесно взаимосвязаны. Все эти три элемента мы определяем на *этапе проектирования*. Архитектурный тест создается позже и не входит в определение фитнес-функции и ее метрики.

Если архитектурный тест автоматизирован (что я настоятельно рекомендую), после его реализации метрика будет определяться автоматически. При реализации теста может потребоваться включить инженерное воображение, чтобы внести изменения в систему или архитектуру, а затем оценить новые инструменты и фреймворки. Все это выполняется после того, как вы зададите цели и метрики. Вопросы о том, как и когда определять фитнес-функцию и реализовывать архитектурный тест, рассматриваются в разделе «Разработка собственных фитнес-функций и метрик» на с. 63.

По сути, фитнес-функция — это ваше определение того, что такое «хорошо». Рассмотрим несколько примеров.

Фитнес-функции: покрытие тестами

В этой гипотетической системе критически важно поддерживать покрытие модульными тестами выше порога в 90 %[1]. Целевое значение для автоматизированных интеграционных тестов — покрытие более 50 % строк кода. Посмотрим, как описать эти две цели в виде фитнес-функций с контекстом (примеры 2.1 и 2.2).

---

[1] Покрытие тестами — термин, используемый в программной разработке для измерения степени покрытия исходного кода установленным набором тестов. Например, очень распространено измерение количества строк кода («покрытие строк»), которые анализируются («тестируются») во время выполнения автоматизированных тестов. Существуют и другие способы измерения (например, покрытие ветвей), но для простоты объяснения я остановлюсь на покрытии строк: если программа из 100 строк кода имеет покрытие тестами 80 %, значит, при выполнении тесты анализируют 80 строк этого кода.

Пример 2.1. Фитнес-функция

```
Покрытие модульными тестами > 0,9;
Выполнять при каждой сборке CI; завершаться неудачей, если покрытие ниже
целевого
```

Пример 2.2. Фитнес-функция

```
Покрытие интеграционными тестами > 0,5;
Выполнять при каждой ночной сборке интеграционного теста; завершаться неудачей,
если покрытие ниже целевого
```

Как видно из этих простых примеров, фитнес-функция определяет целевую метрику (покрытие тестами), важный для метрики контекст (тип тестов для выполнения и время их выполнения) и дополнительную контекстную информацию, необходимую для автоматической проверки метрики.

Фактическая реализация (например, измерение покрытого тестами кода во время их выполнения, как описано в примере 2.1; или настройка специальной тестовой среды, выполнение тестов и проверка результата, как в примере 2.2) является частью реализации архитектурного теста.

Фитнес-функции: интеграционные тесты с сетевыми задержками

Тестируемая система интегрируется со сторонней системой через REST/JSON API. Если этот API работает медленно или не отвечает, стабильность и производительность системы ухудшатся, поэтому необходимо убедиться, что система корректно обрабатывает такие ситуации и работает согласно ожиданиям. Пример такой фитнес-функции представлен ниже (пример 2.3).

Пример 2.3. Фитнес-функция

```
Ошибки интеграционных тестов = 0% (при сетевой задержке 10с для вызова
стороннего API); выполнять при каждой ночной сборке интеграционного теста;
завершаться неудачей при сбое интеграционного теста
```

В этом примере состав фитнес-функции может быть не столь очевиден, как в предыдущих двух примерах. Целевой показатель — 0 % ошибок при выполнении тестов (то есть полное отсутствие ошибок) в условиях имитации сетевой задержки в 10 секунд для вызовов стороннего API во время интеграционного тестирования.

Непосредственная реализация теста включает настройку тестовой среды, имитацию 10-секундной сетевой задержки для вызова стороннего API, ежевечерний запуск сборки интеграционных тестов и завершение с ошибкой при возникновении любой ошибки в ходе этих тестов.

Вариацией примера 2.3 является объединение того же контекста с дополнительной метрикой, например общей пропускной способностью системы при наличии сетевой задержки в 10 секунд, что позволяет проверить корректность работы резервного механизма (пример 2.4).

Пример 2.4. Фитнес-функция

```
Ошибки интеграционных тестов = 0% (при сетевой задержке 10 секунд для вызова
стороннего API); выполнять при каждой ночной сборке интеграционного теста;
завершаться неудачей при сбое интеграционного теста; завершаться неудачей, если
время выполнения теста > 10 минут (стандартное время выполнения без сетевой
задержки составляет менее 5 минут)
```

В этой вариации добавлена дополнительная метрика: должен быть достигнут определенный целевой уровень производительности. При условии 10-секундной сетевой задержки мы проверяем, что резервный механизм системы работает и вся система продолжает функционировать в заданных временных рамках (максимум за двойное время выполнения по сравнению со стандартной сетевой задержкой).

Категории фитнес-функций

Фитнес-функции распадаются на множество категорий (я также называю их *измерениями* (dimensions) и буду использовать оба термина в этой главе взаимозаменяемо). Эти измерения помогают разработчикам определять наиболее полезные фитнес-функции для своих программных систем. Фитнес-функции всегда представляют собой комбинацию описанных здесь измерений; однако не все случайные комбинации возможны или имеют смысл.

Н. Форд, Р. Парсонс и П. Куа приводят отличный список описаний этих категорий, и большинство из этих описаний я и буду использовать. Я также дополню их список теми категориями, которые посчитаю важными.

На эти измерения полезно опираться как на руководство и исходные данные, чтобы учесть все значимые параметры при создании и определении фитнес-функции и целевой метрики, а также при последующей реализации итогового теста, который формирует и проверяет целевую метрику. Измерения можно использовать как каталог, выбирая их и собирая из них подходящую комбинацию для своей системы и контекста (краткий обзор я приведу далее). Как и всегда в программной разработке, использовать следует только те категории, которые дают командам конкретную информацию, направление и практический смысл.

Далее рассмотрим шесть категорий, которые я считаю обязательными, а затем четыре необязательные.

Обязательные категории фитнес-функций

Следующие шесть категорий я считаю обязательными, поскольку их использование всегда оправданно в проектах программной разработки. Если не учесть их при разработке фитнес-функции, то метрика и ее тестирование будут искажены, то есть в определении будет упущено что-то важное.

Атомарная или комплексная?

Какая часть системы задействована в тестировании при создании метрики? На практике этот показатель скорее непрерывный, чем бинарный, но для упрощения понимания мы рассмотрим два крайних случая как отдельные категории.

Атомарные (atomic) фитнес-функции проверяют лишь отдельные, ограниченные параметры системы. Следовательно, их успешное выполнение не обязательно говорит о корректной работе всей системы, а только о проверенной ее части. Типичные примеры: выполнение статического анализа кода (например, измерение цикломатической сложности как показателя сопровождаемости) или измерение покрытия модульными тестами как показателя сопровождаемости или тестируемости.

Комплексные (holistic) фитнес-функции, напротив, предоставляют более полную обратную связь. Их успешное выполнение означает, что бóльшая часть системы работает, как ожидалось, и конечные пользователи могут применять ее по назначению. Комплексные функции обычно сложнее разрабатывать и поддерживать.

Какой триггер запускает выполнение теста?

Помимо тестов, выполняемых вручную, обычно разрабатывают тесты для автоматического выполнения по определенному триггеру — например, рабочий процесс CI, который запускается действием разработчика или по расписанию (скажем, ночью). Непрерывные (continual) фитнес-функции, в отличие от триггерных (triggered), выполняют оценку постоянно, независимо от активности разработки (например, постоянная проверка метрик и их пороговых значений). Непрерывное предоставление информации обычно связано с реальными показателями на продакшен и оценкой показателей метрик, собираемых во время работы системы. Непрерывные фитнес-функции также часто относятся к области мониторинга системы (инструментам); например, мониторинг времени отклика определенного сервиса может быть такой фитнес-функцией.

Где выполняется тест?

Рассматриваемый тест можно запускать в тестовой системе или на продакшен. Другой вариант — запускать его в пайплайне CI/CD (например, можно

измерять покрытие кода модульными тестами прямо на хосте системы CI/CD). Функцию можно оценивать в тестовой системе (тестами производительности или нагрузочными тестами). В некоторых случаях тесты могут даже выполняться напрямую в продакшен-системе. Категории тестов могут пересекаться: например, если тест производительности выполняется в тестовой среде, а его запуск инициируется пайплайном CI/CD. В этой категории определяется, где выполняется тест, требуется ли дополнительное оборудование и повлияет ли тест на работающую продакшен-систему.

Тип метрики

Тип метрики — довольно очевидный параметр для анализа. Значение какого типа выдает архитектурный тест? Простое значение true/false (например, «все тесты пройдены») или числовое? Кроме того, стоит подумать, следует ли сохранять и отображать значения метрики в виде временно́го ряда (если в нем и заключается основная ценность).

Автоматизированные и ручные тесты

Иногда тесты целесообразно выполнять вручную. Обычно это происходит, когда автоматизация требует значительных усилий, стоит слишком дорого или просто неосуществима. Например, проверку юридических требований можно описать как фитнес-функцию, но автоматизировать ее бессмысленно. Тем не менее архитекторы ПО, как правило, стремятся к автоматизации, чтобы иметь возможность легко и часто запускать тесты.

Атрибуты качества

Для меня самая важная категория — это определение *атрибутов качества* (также известных как атрибуты *требований к качеству* или *цели по качеству*) для программной системы. В общем случае при разработке архитектуры ПО есть три ключевых драйвера: функциональные требования, атрибуты качества и ограничения. Цели по качеству определяют, насколько хорошо должна работать система. Они описывают, насколько хорошо функциональные требования всего продукта должны взаимодействовать друг с другом, и формулируют дополнительные требования к качеству системы (например, насколько должно быть легко адаптировать определенную ее часть). Следовательно, согласно Бассу, Клементсу и Казману, цели по качеству очень важны при разработке архитектур ПО[1].

---

[1] Лен Басс (Len Bass), Пол Клементс (Paul Clements) и Рик Казман (Rick Kazman), «Software Architecture in Practice», 3-е изд. (Вестфорд, Массачусетс: Addison-Wesley, 2015), с. 64.

В стандарте Международной организации по стандартизации ISO-25010 приведен приблизительный каталог атрибутов качества. В нем перечислены восемь основных атрибутов (характеристик) качества продукта, которые затем дробятся на более специфические податрибуты (подхарактиристики) (табл. 2.1).

Таблица 2.1. Атрибуты качества и их составляющие[1]

Атрибут	Составляющий атрибут
Функциональная пригодность (functional suitability)	Функциональная полнота (completeness)
	Функциональная корректность (correctness)
	Функциональная целесообразность (appropriateness)
Уровень производительности (performance efficiency)	Временные характеристики (time-behaviour)
	Использование ресурсов (resource utilization)
	Потенциальные возможности (capacity)
Совместимость (compatibility)	Сосуществование (coexistence)
	Интероперабельность (interoperability)
Удобство использования (usability)	Определимость пригодности (appropriateness recognizability)
	Изучаемость (learnability)
	Управляемость (operability)
	Защищенность от ошибок пользователя (user error protection)
	Эстетика пользовательского интерфейса (user interface aesthetics)
	Доступность (accessibility)
Надежность (reliability)	Завершенность (maturity)
	Готовность (availability)
	Отказоустойчивость (fault tolerance)
	Восстанавливаемость (recoverability)

[1] Перевод характеристик и подхарактеристик дается в соответствии с ГОСТ Р ИСО/МЭК 25010-2015. — *Примеч. ред.*

Таблица 2.1 (*окончание*)

Атрибут	Составляющий атрибут
Защищенность (security)	Конфиденциальность (confidentiality)
	Целостность (integrity)
	Неподдельность (nonrepudiation)
	Отслеживаемость (accountability)
	Подлинность (authenticity)
Сопровождаемость (maintainability)	Модульность (modularity)
	Возможность многократного использования (reusability)
	Анализируемость (analyzability)
	Модифицируемость (modifiability)
	Тестируемость (testability)
Переносимость (portability)	Адаптируемость (adaptability)
	Устанавливаемость (installability)
	Взаимозаменяемость (replaceability)

Стандарт ISO — не единственный способ систематизировать характеристики качества ПО; компания Hewlett-Packard разработала другую систему под названием FURPS (https://oreil.ly/hAWgp)[1]. Однако в этой главе я буду использовать атрибуты ISO.

Какой бы шаблон, каталог или стандарт вы ни использовали при обсуждении целей по качеству со стейкхолдерами, помните, что эти цели — один из главных драйверов при разработке архитектуры системы, поэтому они же должны быть главным драйвером при определении фитнес-функций и метрик.

Время и силы стоит тратить только на те атрибуты, которые существенно влияют на ваши общие цели. Исходя из этого, в разделе «Разработка собственных фитнес-функций и метрик» на с. 63 первым этапом создания соответствующих фитнес-функций и их метрик является определение целей по качеству и их согласование с ключевыми стейкхолдерами.

[1] Аббревиатура от functionality, usability, reliability, performance, supportability — функциональность, удобство использования, надежность, производительность, сопровождаемость. — *Примеч. ред.*

Необязательные категории фитнес-функций

Следующие категории могут служить дополнительным ориентиром и быть полезными в вашем контексте. Я считаю их необязательными, поскольку они не всегда применимы. Некоторые из них связаны с дополнительными требованиями к коммуникации и документации, которые обычно возникают только в крупных проектах.

Временная или постоянная?

Если использование и срок действия фитнес-функции изначально ограничены, ее можно явно пометить как *временную* (temporary); остальные фитнес-функции относятся к категории *постоянных* (permanent). «Постоянная» здесь означает, что для функции не планировалась конкретная дата окончания, а не то, что функция будет работать «вечно»: ее, как и любой другой элемент программной разработки, можно изменить или убрать.

Хороший пример временной фитнес-функции — целенаправленная и длительная работа по изменению или рефакторингу ПО. Пока рефакторинг продолжается, его помимо прочих поддерживают временная фитнес-функция и соответствующая ей метрика. Как только рефакторинг завершается, эту функцию убирают.

Статическая или динамическая?

У *статической* фитнес-функции, или, точнее, у метрики статической фитнес-функции, целевая метрика определена статическим значением. Соответственно, проверка выполняется относительно значения этой статической метрики. Такую фитнес-функцию вы видели, в том числе, в примерах 2.1 и 2.2, где мы проверяли, всегда ли покрытие кода выше определенного статического значения.

Динамическая фитнес-функция, напротив, определяет значение целевой метрики в заданном диапазоне в зависимости от другого значения. Например, целевой диапазон времени отклика можно задать в зависимости от количества активных на данный момент пользователей. В частности, установить целевой диапазон времени отклика от 50 до 100 миллисекунд, если количество онлайн-пользователей составляет от 10 000 до 100 000 человек.

Разумеется, для такого динамического определения создать (автоматизированный) тест сложнее. Зато функция с динамическим определением может лучше адаптироваться к реальным сценариям использования, чем функция со статическим определением, и поэтому давать более ценный результат, зависящий от вашего сценария.

Кто является целевой аудиторией?

В целевую аудиторию фитнес-функций и метрик могут входить разработчики ПО, инженеры по эксплуатации, продакт-менеджеры и другие стейкхолдеры. Определение аудитории полезно при использовании фитнес-функций в крупных организациях, с потребностью в дополнительной документации и коммуникации. Предварительное знание целевой аудитории может быть важно при принятии решения о том, как и где визуализировать результаты и предоставлять к ним доступ.

Где будет применяться функция и метрика?

Если у вас большая система или несколько систем, то может потребоваться ограничить область действия и выполнение фитнес-функции и ее метрики только одной системой, подсистемой или сервисом. Обычно это сопровождается дополнительными потребностями в документации и коммуникации, которые существуют в более крупных проектах программной разработки. Как вариант, вы можете ограничить данную фитнес-функцию определенной технологией в рамках системы или подсистемы, например, требуя один тип покрытия кода для компонентов фронтенда, написанных на JavaScript, и другой тип для компонентов бэкенда, написанных на Java.

Категории фитнес-функций: обзор каталога

Таблицы 2.2 и 2.3 содержат краткий обзор категорий и их возможных значений, который можно использовать в качестве справочного материала при создании ваших первых фитнес-функций.

Таблица 2.2. Обязательные категории фитнес-функций

Категория	Возможные значения
Обратная связь	Атомарная или комплексная
Выполнение теста	По триггеру или непрерывно
Место выполнения	CI/CD, тестовая среда, продакшен-система и т. д.
Тип метрики	True/False, дискретное значение, временные ряды/исторические значения
Автоматизация	Автоматизированные или ручные тесты
Атрибуты качества	Атрибуты ISO: функциональная пригодность, уровень производительности, совместимость, удобство использования, надежность, защищенность, сопровождаемость, переносимость

Таблица 2.3. Необязательные категории фитнес-функций

Категория	Возможные значения
Временная или постоянная	Временная или постоянная
Статическая или динамическая	Статическая или динамическая
Целевая аудитория	Определяется вами; например, разработчики и владельцы продукта
Область применимости	Определяется вами; например, определенные технологии (только JavaScript) или определенные области системы (сервис A или сервис B)

Теперь, когда у вас есть представление о категориях, применим их к фреймворку пирамиды тестирования.

Пирамида тестирования

Пирамида тестирования — это широко известная и общепринятая концепция, используемая для классификации различных видов автоматизированных функциональных тестов по трем уровням[1]. Мартин Фаулер (Martin Fowler) описывает такую пирамиду как «способ представления того, каким образом следует использовать различные виды автоматизированных тестов, чтобы создать сбалансированный портфель». *«Сбалансированный»* в данном контексте означает портфель автоматизированных функциональных тестов, который позволяет найти компромисс между временем выполнения, а также затратами на запуск и сопровождение, с одной стороны, и надежностью, которую могут обеспечивать автоматизированные тесты, — с другой. Как правило, чем больше тестов у вас есть, тем больше вы должны быть уверены в том, что приложение работает как ожидалось. Но большое количество тестов подразумевает и более высокие затраты на их запуск и сопровождение.

На рис. 2.2 показана трехслойная базовая пирамида тестирования. Каждый слой обеспечивает соответствующий уровень обратной связи.

---

[1] См., например, Эшли Дэвис (Ashley Davis), «Bootstrapping Microservices with Docker, Kubernetes, and Terraform» (Manning, 2021); Лиза Криспин (Lisa Crispin) и Джанет Грегори (Janet Gregory), «Agile Testing: A Practical Guide for Testers and Agile Team» (Addison-Wesley, 2008); и Мартин Фаулер (Martin Fowler), «TestPyramid», MartinFowler.com, 1 мая 2012 г., https://oreil.ly/9o1DV.

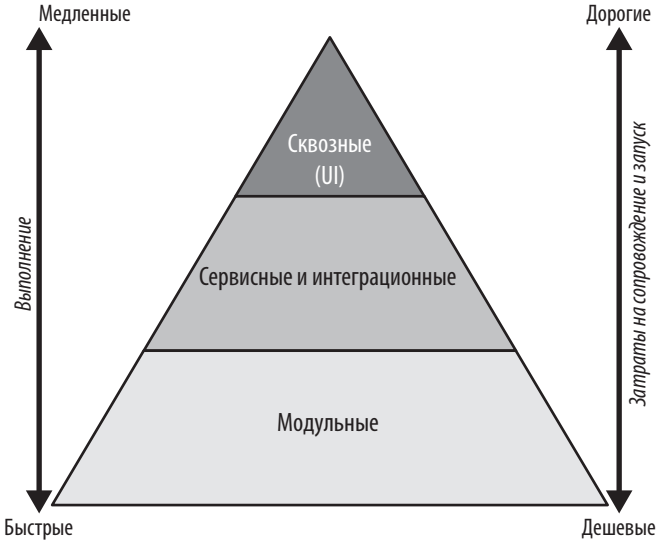


Рис. 2.2. Пирамида тестирования

Базовый слой

Базовый слой состоит из самых простых тестов, которые мы выполняем: модульных, или юнит-тестов. Если модульный тест не проходит, сразу понятно, где находится проблема: она должна быть в тестируемом модуле. В целом, если модульный тест не прошел, довольно трудно привести реалистичный пример ситуации, когда в модуле возникли проблемы и конечный пользователь их обнаружил.

Средний слой

Средний слой описывает сервисные и интеграционные тесты. (Иногда здесь также размещают компонентные и API-тесты.)

Верхний слой

Наверху находятся сквозные (E2E) тесты, часто выполняемые непосредственно через пользовательский интерфейс (UI) приложения. Если эти тесты не проходят, легко увидеть, к каким негативным последствиям это может привести в реальных ситуациях; и в то же время обнаружить компонент с ошибкой может быть сложно, поскольку на этом уровне совместно работают многие компоненты.

Тесты внизу пирамиды обычно выполняются быстро; кроме того, их просто и недорого сопровождать и запускать. Чем выше вы поднимаетесь, тем

медленнее выполняются тесты и тем дороже их разрабатывать и сопровождать. Из этих соображений крайне важно найти баланс с точки зрения количества тестов на трех слоях для достижения наилучшего результата — то есть иметь такой набор тестов, который просто поддерживать и который обеспечивает максимально высокую уверенность в том, что система работает как задумано.

Конечно, эту модель можно использовать, чтобы определиться с тем, куда направить усилия по тестированию и автоматизации, но она идеализирована и не всегда верна. Например, в одной системе может потребоваться много интеграционных тестов на среднем слое; в другой системе, без пользовательского интерфейса, могут отсутствовать тесты на верхнем слое.

Прежде чем я покажу, как адаптировать эту структуру пирамиды для систематизации фитнес-функций, бегло пройдемся по категориям, которые помогают с определением фитнес-функций.

Пирамида фитнес-функций

Концепция *пирамиды фитнес-функций* тесно связана с концепцией функциональной пирамиды тестирования. Я адаптировал основную концепцию пирамиды для фитнес-функций и архитектурных метрик, чтобы повторно использовать ее в архитектурных тестах для баланса (эта идея важна также в архитектурной верификации).

Статистик Джордж Бокс однажды написал, что «все модели ошибочны», но добавил, что «все же некоторые из них полезны»[1]. Я надеюсь, что представленная здесь модель относится к категории полезных.

Как и в случае с исходной пирамидой, самые простые и недорогие тесты находятся в основании пирамиды, более продвинутые — в середине, а самые сложные, которые должны обеспечивать «максимальную» обратную связь в реальных условиях, находятся на вершине пирамиды тестирования фитнес-функций.

Фитнес-функции всегда создаются по нескольким категориям, которые я представил в предыдущем разделе. Хотя все обязательные категории важны для описания полезной фитнес-функции, лишь немногие влияют на классификацию фитнес-функций по слоям пирамиды. Таким образом, я рекомендую использовать концепцию пирамиды, чтобы сформировать

[1] Джордж Э. П. Бокс (George E. P. Box), Science and Statistics, «Journal of the American Statistical Association» 71, no. 356 (декабрь 1976 г.): 791-799.

такой набор архитектурных тестов, в котором будут сбалансированы, с одной стороны, время выполнения, затраты на запуск и поддержку, а с другой — уверенность, которую эти архитектурные тесты (преимущественно автоматизированные) обеспечивают.

Отмечу, что далеко не для каждой среды я бы рассматривал время выполнения (короткое или длительное) как значимый фактор для отнесения фитнес-функции на тот или иной слой пирамиды. Две наиболее значимые категории для классификации фитнес-функции и ее тестовой реализации по слоям — это ее обратная связь (атомарная или комплексная) и триггер выполнения, как показано на рис. 2.3[1].

На верхнем слое пирамиды находятся только комплексные фитнес-функции. Что касается триггера выполнения, то сделать так, чтобы тесты и проверки, особенно предоставляющие комплексную информацию, выполнялись непрерывно, довольно трудно. Таким образом, то, какая фитнес-функция появится на том или ином слое, определяется взаимодействием этих категорий.



Рис. 2.3. Категории, определяющие, к какому слою пирамиды следует отнести фитнес-функцию

[1] Н. Форд, Р. Парсонс и П. Куа считают, что эти две категории смешаны по определению, поскольку они тесно связаны при определении реальных фитнес-функций, метрик и тестов; конечно, другие категории могут оказывать влияние, но это очень сильно зависит от конкретного сценария использования. Для справки: эти же авторы предложили считать эти две категории «смешанными» по определению.

Верхний слой

Тесты верхнего слоя являются комплексными и обеспечивают наиболее полную информацию о состоянии системы и ее функциональности для конечных пользователей. Следовательно, эти метрики и проверки наиболее близки к реальным сценариям использования.

Обычно эти тесты сложнее и дороже всего создавать и поддерживать. Также в определенных обстоятельствах они, скорее всего, будут вести себя необычно, поскольку включают множество компонентов и тестируют большую область системы. Безусловно, мы стараемся получить комплексную обратную связь по всей системе, но мы также должны учитывать непредвиденные ошибки, локализовать которые сложнее обычного.

Таким образом, тесты верхнего уровня сложно создавать и сопровождать, а проблемы, которые они выявляют, иногда трудно отследить до первопричины. Из этих соображений мы выбираем только несколько «хороших» тестов верхнего уровня, поскольку пытаемся сбалансировать усилия и результат (меньше тестов на вершине пирамиды).

В качестве примера комплексного теста верхнего слоя: для интернет-магазина можно постоянно измерять ключевые показатели, такие как количество оформленных заказов в минуту, выручка в минуту или количество входов в систему в минуту, если они попадают в заданный целевой коридор значений. Отклонение может указывать на скрытую техническую проблему, которую необходимо устранить (или которая, возможно, появилась при последнем развертывании). Верхний слой также включает в себя тесты из области *хаос-инжиниринга*[1] — практики добавления ошибок на продакшен, чтобы проверить устойчивость системы при измерении общей работоспособности и ее готовности к обслуживанию конечных пользователей.

Средний слой

Средний слой, как показано на рис. 2.3, состоит из запускаемых по триггеру комплексных или непрерывных атомарных фитнес-функций. Эти фитнес-функции предоставляют обширную обратную связь о работоспособности всей системы, но не выполняются непрерывно; они запускаются по специальному действию разработчика.

---

[1] См., например, «Principles of Chaos Engineering» (https://principlesofchaos.org) и Netflix's Chaos Monkey (https://oreil.ly/4i0Z1).

Триггерную комплексную метрику можно тестировать, выполнять и оценивать в составе сборки интеграционных тестов, внутри тестовой системы или в качестве этапа автоматизированного пайплайна развертывания. Прогон интеграционных тестов с несколькими тест-кейсами также может обеспечить качественную обратную связь о производительности всей системы, ее поведении при выполнении транзакций или ее устойчивости (если при таком тестировании будет смоделирован отказ других частей системы или сторонних систем). Примеры фитнес-функций 2.3 и 2.4 можно рассматривать как тесты в среднем слое.

Простая атомарная фитнес-функция, которая непрерывно оценивается в продакшен-системе, также будет относиться к этому слою: например, мониторинг и измерение атомарных значений в реальном времени, таких как длительность транзакции или производительность на уровне конечного пользователя (скажем, время загрузки веб-приложений в браузере).

Нижний слой

В основании пирамиды фитнес-функций находятся триггерные атомарные фитнес-функции. Обычно их легко и недорого реализовывать и запускать. Благодаря своей простоте они хорошо устоялись и зачастую уже интегрированы в пайплайн CI/CD. Именно они составляют основу для определения полезных метрик. Этот слой может включать в себя метрики покрытия кода, статического анализа кода (например, цикломатическую сложность) или простые тесты производительности.

Я рекомендую сделать нижний слой широким, а к верхним слоям уже подходить сбалансированно. Использовать фитнес-функции без тестов верхнего или даже среднего слоя вполне нормально! И наоборот, могут быть ситуации, когда нужно перевернуть пирамиду вверх дном, чтобы она содержала много непрерывных комплексных тестов и лишь несколько триггерных атомарных. Этот выбор всегда сильно зависит от контекста и целей, но обычно я рекомендую следовать общепринятой форме пирамиды, когда больше всего тестов находится в основании и меньше — в слоях выше.

Однако аналогия между пирамидами тестов и фитнес-функций рушится, когда мы задумываемся, сколько тестов должно быть в основании пирамиды. Обычно никто не ограничивает количество модульных тестов, если они полезны для детализации. Но я не рекомендовал бы создавать в основании пирамиды фитнес-функций как можно больше тестов, поскольку все они подразумевают дополнительные накладные расходы.

Примеры и их полная категоризация

Итак, мы оценили широту возможностей фитнес-функций, теперь вернемся к предыдущим примерам и выведем их соответствующие категории, чтобы несколько оживить теорию.

В примере 2.1 мы определили первую версию фитнес-функции для проверки покрытия модульными тестами.

Вот разбивка этой фитнес-функции по категориям:

- Уровень обратной связи: атомарный
  - Покрытие модульными тестами дает лишь ограниченную информацию о функционировании всей системы.
- Триггер выполнения: по событию
- Место выполнения: CI/CD
  - Выполнение запускается при каждом пуше в систему контроля версий, выполняются модульные тесты, измеряется покрытие модульными тестами.
- Тип значения метрики: конкретное значение (> 90 %)
- Автоматизация
  - Фитнес-функция будет оцениваться автоматически.
- Атрибут требования к качеству: удобство сопровождения
  - Фитнес-функция отслеживает, удобно ли сопровождать систему на определенных уровнях; хорошее покрытие тестами служит индикатором того, что систему проще сопровождать (адаптировать, изменять, улучшать).
- Статическая или динамическая: статическая

Разбивка примера 2.2 по категориям (покрытие интеграционными тестами) такая же. Фитнес-функция находилась бы на нижнем слое пирамиды.

В примере 2.3 мы определили первую версию фитнес-функции для проверки конкретной функциональности в случае сетевой задержки.

Вот разбивка этой фитнес-функции по категориям:

- Уровень обратной связи: атомарный или комплексный
  - Довольно сложно однозначно решить, относится ли данная фитнес-функция к комплексным. В основном это зависит от выполняемых тестов и важности сторонней системы для нашей системы.

Если сторонняя система используется во многих сценариях для нашей системы, то фитнес-функцию можно отнести к комплексным; в противном случае я бы отнес ее к атомарным.

- Триггер выполнения: по событию
- Место выполнения: CI/CD и тестовая среда
  - Функция запускается ежедневно в рамках пайплайна CI, но тесты выполняются в тестовой среде.
- Тип значения метрики: 0/1 (если все тесты прошли)
  - Можно возразить, что нам интересен процент успешного прохождения всех тестов, но поскольку мы требуем, чтобы все тесты проходили, то решение — 0 или 1.
- Автоматизация
  - Фитнес-функция будет оцениваться автоматически и запускаться по ночам.
- Атрибут требования к качеству: надежность
  - Фитнес-функция помогает поддерживать высокую надежность системы, даже если сторонний интерфейс отвечает медленно.
- Статическая или динамическая: статическая

Вот разбивка примера 2.4 по категориям:

Уровень обратной связи (атомарный или комплексный): может быть любым

Решение, является ли эта конкретная фитнес-функция атомарной или комплексной, в основном зависит от выполняемых тестов, важности сторонней системы для нашей системы и общей производительности системы. Если ваша система использует стороннюю систему в нескольких сценариях, то это влияет на производительность всей системы, поэтому вы можете классифицировать фитнес-функцию как комплексную.

Триггер выполнения и место выполнения: по событию; CI/CD и тестовая среда

Функция запускается ежедневно в рамках пайплайна CI, но тесты выполняются в тестовой среде.

Тип значения метрики: два типа

Первый тип, если все тесты проходят, — это 0/1. Можно возразить, что нам интересен процент успешного прохождения всех тестов, но по-

скольку мы требуем, чтобы все тесты проходили, то решение — 0 или 1. Второй — конкретное значение для производительности: в данном случае быстрее 10 минут.

Автоматизация: да

Фитнес-функция будет оцениваться автоматически и запускаться по ночам.

Атрибуты требований к качеству: надежность, уровень производительности

Фитнес-функция помогает поддерживать в системе высокую надежность и адекватную производительность, даже если сторонний интерфейс отвечает медленно.

Статическая или динамическая: статическая

Это статическая функция, поскольку две метрики определены статически и целевые значения не зависят от результата другой фитнес-функции.

Как я уже говорил, я бы поместил примеры 2.3 и 2.4 на средний слой пирамиды фитнес-функций.

Полная категоризация примеров верхнего уровня

В этом разделе представлено полное определение двух более сложных фитнес-функций верхнего уровня (примеры 2.5 и 2.6). Мы проведем их полную разбивку по категориям. В пирамиде фитнес-функций они занимают место на вершине или близко к ней.

Пример 2.5. Фитнес-функция (интернет-магазин)[1]

```
Измерять выручку в минуту в течение дня. Считать неудачей, когда выручка
в минуту, рассчитанная на основе текущего времени, выходит за пределы коридора,
заданного следующей таблицей:

Временной интервал       Минимальная выручка (в минуту)
01:00 – 05:00            € 200
05:01 – 07:00    400     € 400
07:01 – 09:00    600     € 600
09:01 – 11:30    900     € 900
```

[1] Это очень упрощенная таблица, которая приведена только для примера; реальная версия требует более детализированной структуры или более сложной связи между временем и ожидаемой выручкой.

```
11:31 – 13:30    1100    € 1100
13:31 – 17:30    950     € 950
17:31 – 19:30    1500    € 1500
19:31 – 21:00    750     € 750
21:01 – 00:59    300     € 300
```

Разберем фитнес-функцию из примера 2.5 по категориям:

Уровень обратной связи (атомарный или комплексный): комплексный

Эта метрика является прямым показателем производительности всей системы.

Триггер выполнения и место выполнения: непрерывно, продакшен-среда

Оценка фитнес-функции выполняется непрерывно в продакшен-среде.

Тип значения метрики: дискретный.

Значением является выручка. Требуется проверка, превышает ли значение порог.

Автоматизированная: да

Фитнес-функция будет оцениваться автоматически.

Атрибуты требований к качеству: несколько

Включают надежность, уровень производительности, удобство использования и др. Поскольку мы измеряем всю систему, на примерах показана проверка нескольких атрибутов качества напрямую.

Статическая или динамическая: динамическая

Фитнес-функция находилась бы на верхнем слое пирамиды.

Пример 2.6. Фитнес-функция (надежность интернет-магазина)[1]

```
Развернуть новый релиз на продакшен (ночью, в 01:00).
Во время развертывания релиза постоянно выполнять набор регрессионных тестов,
содержащий пять основных сценариев использования конечным пользователем (вход
в систему, добавление товара в корзину, удаление товара из корзины, просмотр
корзины, оформление заказа). Система выполняет все действия и отвечает в течение
100 мс. Считать неудачей, когда тестовый сценарий завершается неуспешно.
Считать неудачей, когда система не выполняет действия и не отвечает менее чем за
100 мс.
```

[1] Это тест на надежность, который проверяет работоспособность системы во время постепенного обновления. Тест проверяет, чтобы при развертываниях система обеспечивала нулевое время простоя; кроме того, он может показать, что даже если узел по какой-то причине недоступен, то система в целом отзывчива и работает согласно ожиданиям.

Разобьем фитнес-функцию из примера 2.6 по категориям:

Уровень обратной связи (атомарный или комплексный): комплексный

Эта метрика напрямую измеряет работу всей системы (а не всех нод в онлайне) во время развертывания.

Триггер выполнения и место выполнения: по событию, продакшен-среда

Оценка фитнес-функции выполняется с использованием дискретного триггера (ежедневно в 01:00) в продакшен-среде.

Тип значения метрики: два типа

Первый тип — дискретное значение (производительность), с проверкой, превышает ли значение порог.

Второе значение — 0/1: система доступна во время развертываний, и тесты не проваливаются.

Автоматизированная: да

Фитнес-функция будет оцениваться автоматически.

Атрибуты требований к качеству: несколько

Включают надежность и уровень производительности.

Статическая или динамическая: статическая

Эта фитнес-функция находилась бы на верхнем слое пирамиды.

Как видим, при разбивке реальных примеров категорию не всегда можно определить однозначно. С другой стороны, категории нужны не для этого. В первую очередь они должны служить каталогом важных параметров фитнес-функций и их метрик. Второе их назначение заключается в том, что они помогают архитекторам ПО выявлять неохваченные области, где могут потребоваться дополнительные усилия.

Разработка собственных фитнес-функций и метрик

Сейчас я покажу, как я разрабатываю фитнес-функции и продолжаю постоянно и итеративно вносить изменения после формирования первоначального набора функций. Для меня основная отправная точка для всей работы, связанной с программной архитектурой, — это цели системы по качеству. Ее же я рекомендовал бы и вам при работе над фитнес-функциями и метриками для своей системы.

Если вы еще не согласовали основные цели по качеству системы со стейкхолдерами, самое время начать. Сделайте подборку релевантных целей по качеству и сформируйте их общее представление, согласованное всеми стейкхолдерами системы.

Н. Форд, Р. Парсонс и П. Куа утверждают: «Команды должны определять фитнес-функции как часть первоначального понимания общей архитектуры своего проекта. Им также следует заранее определиться с фитнес-функцией системы, чтобы понять, какие изменения им необходимы»[1]. Хотя в целом я поддерживаю эту идею, определить все фитнес-функции с самого начала так же сложно, как и знать все требования на старте. Я рекомендую начинать с малого и простого и обучаться уже по ходу реализации системы. Используйте эти знания для улучшения, изменения или добавления новых фитнес-функций и метрик по мере необходимости. Начните с соответствующих тестов на нижнем слое пирамиды фитнес-функций.

Следующий порядок действий, который я бы полностью интегрировал в итеративную разработку, например Scrum, поможет вам определить первые фитнес-функции и внедрить архитектурные тесты:

1. Совместно с ключевыми стейкхолдерами определите наиболее важные атрибуты качества, установите цели архитектуры и задокументируйте их.

   Постановка целей помогает избежать создания бесполезных фитнес-функций и автоматизированных тестов. Я часто видел, как автоматизируют тесты, не направленные на ключевые цели качества, просто потому что это легко. Но ценности это приносит мало или не приносит никакой. Фокус на основных архитектурных целях создает осмысленность цели.

2. Сформулируйте черновые определения фитнес-функций и их целевые метрики.

   Подумайте о важных для вас параметрах. Составьте их черновой список, доступный всей команде (можно использовать бэклог). Документируйте категории, которые вы точно считаете релевантными.

   Зачем нужен общий список или бэклог? Чтобы определить правильные фитнес-функции и создать автоматизированный тест, который выводит целевую метрику, требуются усилия. С помощью бэклога вы можете собирать идеи, пока не будете готовы к их реализации.

---

[1] Форд Н., Парсонс Р., Куа П. «Эволюционная архитектура». Глава 2 «Фитнес-функции».

Документирование категорий для каждого черновика фитнес-функции также может пригодиться позже, чтобы выбрать фитнес-функции в новых областях, например для охвата еще не учтенных целей качества системы, или чтобы сбалансировать портфель тестов, перенося тесты с другого слоя.

3. Расставьте приоритеты и выберите фитнес-функции, которые важны, полезны и которые можно тестировать в данный момент.

   Учтите области и измерения, которые не покрыты существующими тестами, а также слои пирамиды. Есть ли у вас уже сбалансированный портфель архитектурных тестов и метрик? Если вы проделываете эту работу впервые, начните с чего-то простого из нижней части пирамиды.

4. Завершите определения любых незавершенных фитнес-функций среди выбранных.

   Сохраните полное определение и его классификацию в соответствующем слое пирамиды. Во время следующей итерации процесса эти наработки дадут ценное представление об областях, которые вы уже охватили.

5. Разработайте автоматизированный тест, который может генерировать метрику.

   В идеале эти тесты нужно проверять часто. В зависимости от типа фитнес-функции и ее определения обычно также следует убедиться, что тест непосредственно проверяет целевую метрику. Я настоятельно рекомендую автоматизировать его по умолчанию; только несколько очень специфичных метрик стоит создавать и проверять вручную.

6. Визуализируйте результаты.

   Используйте дашборд или другой способ визуализации, чтобы делиться результатами со всей командой и при необходимости с соответствующими стейкхолдерами.

7. Регулярно пересматривайте пирамиду.

   Возможно, некоторые тесты ненадежны, какие-то метрики уже стали бесполезными или затраты на их сопровождение слишком высоки. Выводите из эксплуатации тесты и фитнес-функции, которые не приносят достаточной ценности или больше не нужны.

   Также корректируйте существующие фитнес-функции и метрики по мере необходимости: например, делайте их более жесткими, чтобы видеть улучшения в системе, или более мягкими — они не так важны для достижения общих целей системы.

Заключение

Произвольное тестирование системы может привести к тому, что вы упустите из виду самое главное: общее качество в ключевых областях, таких как производительность, безопасность и модифицируемость. Начните с создания надежной основы архитектурных тестов, которые относительно легко создать и можно развивать и расширять по мере необходимости. Используя фитнес-функции как метод создания метрик, адаптированных под конкретную систему, вы сможете проводить ее кастомизацию, а соблюдая описанный здесь процесс — сократить ненужные накладные расходы. Пирамида фитнес-функций предоставляет дополнительный уровень, который служит для классификации тестов и балансировки усилий. Конечная цель — метрики: неискаженные измерения, которые помогают команде сосредоточиться на согласованных целях программной системы.

ГЛАВА 3

Эволюционная архитектура: управление архитектурой через тестируемость и развертываемость

Дэвид Фарли

Программная архитектура иллюзорна, но в то же время важна. Например, она может определять такие важные характеристики систем, как их масштабируемость, производительность и отказоустойчивость. А ее иллюзорность проявляется в том, что оценка этих качеств часто бывает размытой и субъективной.

Архитектурные описания и документы — это своего рода карты местности для создаваемых нами систем. Они позволяют ориентироваться в пространстве и при этом не содержат излишне подробных деталей, которые, скорее всего, все равно будут меняться. По мере того как мы будем лучше понимать потребности и запросы пользователей и заказчиков, мы будем изменять свое видение того, какое место на шкале должны занимать архитектурные свойства наших систем.

Работая с программной архитектурой, вы можете задаться вопросом: если я начинаю с простой системы, как мне справляться со всплесками запросов? А если запросов больше не становится? Как справляться с растущими требованиями к производительности, безопасности и высокой доступности? Как поддерживать систему в состоянии, которое легко эволюционирует, чтобы удовлетворять новые или непредвиденные требования? Как оставить возможность вносить изменения, которые не планировались, и при этом не тормозить процесс разработки, поскольку изменения требуют обеспечения будущей устойчивости?

В этой главе мы покажем, что такие вопросы решаются через защитный подход: архитекторам ПО необходимо проектировать и изучать методы управления сложностью систем, которые они создают.

Важность обучения и открытий

Сложные системы никогда не формируются полностью готовыми в умах своих создателей; напротив, они являются результатом постепенного развития и изучения условий, в которых они используются. Занимаясь программной разработкой, вы всегда находитесь в процессе непрерывного обучения и совершения открытий. Это означает, что если вы хотите делать свое дело хорошо, то нужно отказаться от мысли, что вы должны или сможете заранее предугадать, как будет использоваться и развиваться ваша система со временем.

В реальности новые системы являются частью сложных адаптивных систем, объединяющих разработчиков, пользователей и заказчиков, а также их окружение и контексты организации. Эта реальность требует более динамичного, почти органичного подхода к архитектуре и проектированию, который позволяет учиться по ходу работы и адаптировать ПО к постоянно меняющимся представлениям о том, что оно делает и как.

Надо уметь начинать работу, не имея всех ответов. Также нужно защищать и поддерживать способность вносить изменения в систему по мере углубления ее изучения. Следовательно, необходимо быть всегда готовыми к переменам.

Инструменты для устойчивых изменений

Какие архитектурные решения позволяют всегда быть готовыми к переменам? Существует пять атрибутов проектирования ПО, которые помогают достичь устойчивого эволюционного подхода:

Модульность (modularity)
: Разделение систем на части, которые могут изменяться, не вызывая принудительного изменения в других частях.

Связность (cohesion)
: Сохранение совместно изменяемых частей кода ближе друг к другу.

Разделение ответственности (separation of concerns)
: Нацеливание каждой части кода и системы на решение одной проблемы.

Абстракция/сокрытие информации (abstraction/information hiding)

Формирование такого разграничения в системе, которое позволяет использовать поведение одной части системы без необходимости понимать, как работают другие части.

Связанность (coupling)

Степень, в которой отдельные части системы должны изменяться вместе.

Эти атрибуты универсальны для информации в целом, а не только для программ. Они ничего не говорят о том, как работают системы или о природе какой-либо конкретной технологии.

Независимо от назначения системы и используемых в ней технологий, если она модульна и связна, эффективно разделяет ответственность, использует хорошую абстракцию для разграничения своих частей и корректно управляет связанностью между частями, с ней будет легче работать, ее будет легче изменять, она будет понятнее и ее легче будет тестировать, чем аналогичную систему, которая в меньшей степени соответствует этим условиям.

Такой эволюционный подход к проектированию и развитию архитектуры и системы означает, что по мере накопления знаний вам будет легче адаптировать систему к тому новому, что вы для себя открываете.

Например, как быть, если вы вдруг узнаете, что ваша система будет работать эффективнее с графовой СУБД, чем с реляционной? Если вы с самого начала решили отделить ядро логики предметной области от подробностей сохранения данных, то вопрос с миграцией будет решаться проще. В такой системе будут отлично реализованы разделение ответственности, модульность, связность и абстракция. Вам будет понятно, как относительно легко переключить репозиторий на основе реляционной СУБД на репозиторий с графической БД. Если же у вас логическое ядро и механизм сохранения данных переплетены, то даже представить такой переход будет очень сложно.

Тестируемость: создание высококачественных систем

Как внедрить эти пять атрибутов качества в создаваемые системы? Ответ, как обычно, «в зависимости от обстоятельств». Такими обстоятельствами могут быть один или несколько разработчиков, работающих над кодом, а точнее их навыки, опыт и степень вовлеченности.

Если у вовлеченных людей недостаточно навыков, то вне зависимости от их стараний хорошего результата у них не будет. Если им не хватает опыта, то даже имея навыки в какой-то специфике проблемы, они упустят некоторые нюансы, например «дырявые» абстракции или разные виды связанности, и это может поставить под угрозу разработку сложной системы. Если у них недостаточно мотивации делать свою работу отлично, то результат может оказаться плохим при любых навыках или опыте.

И это еще не все. Вы должны быть уверены, что ваше ПО работает. Следовательно, его надо тестировать, чтобы убедиться, что система делает то, что от нее ожидалось. Еще важнее то, что у вас должны быть развязаны руки, чтобы безопасно и уверенно изменять систему. Если вы пишете код или системы, не тестируя их, то вам, скорее всего, лучше оторваться от клавиатуры и подумать над тем, что вы делаете, еще раз. Если вы разрабатываете только лишь бы что-то разрабатывать, то вы никогда не получите ничего качественного, если это не простой, одноразовый код.

Раз тестирование необходимо, обсуждать можно только то, каким оно должно быть: ручным или автоматизированным. Ручное тестирование — это медленно, неэффективно, дорого и ненадежно; автоматизированное тестирование — это, безусловно, эффективнее и, как правило, выше по качеству. Так как же предельно упростить тестирование кода? Как добиться *тестируемости*?

Чтобы протестировать компонент в системе, нужен доступ к этому компоненту, а сам компонент должен быть готов к оценке и находиться в определенном состоянии. Во время тестирования вы будете вызывать некое поведение системы, фиксировать ее отклики, оценивать их и определять, соответствуют ли они вашим ожиданиям.

Итак, какие характеристики системы будут этому способствовать? Будет прекрасно, если вам удастся сосредоточиться только на ее поведении. Модульные системы тестировать легче, чем немодульные.

Изучать поведение конкретного модуля удобнее всего, если этот модуль легко подготовить к тестированию. Если тестируемый модуль не обладает внутренней связностью и при этом тесно сцеплен с другими модулями, тестировать его будет сложно. Всегда легче тестировать ПО с высокой внутренней связностью и слабыми связями с другими компонентами.

Чтобы привести систему в конкретное состояние, нужно контролировать ее переменные, а значит, для выполнения оценки нужно ограничивать ее сложность. При правильном разделении ответственности мы можем переключаться на поведение одного интересующего нас компонента, а не совокупности компонентов, что явно усложняет картину.

Наконец, если тесты в определенной степени отделены от тестируемого кода, то код можно изменять без обязательного изменения тестов. Это означает, что в удобном для тестирования коде должны быть правильные абстракции, скрывающие от тестов детали реализации. Такие абстракции, в свою очередь, также упрощают настройку тестирования, поскольку их границы формируют границы тестирования и, как следствие, ограничивают объем и сложность тестов.

Таким образом, если какой-то код легко тестировать, значит, с ним легко работать и его легко изменять. Интересное наблюдение!

Если вы хотите спроектировать код так, чтобы его было удобно тестировать, то при проектировании следует руководствоваться тестами. Если вы учитываете тестируемость кода, когда выстраиваете процесс разработки и проектирования систем, то вы повышаете качество проектирования.

Проектирование на основе тестируемости также повышает уровень профессионализма разработчиков и помогает создавать системы, в которых соблюдены пять вышеперечисленных атрибутов, гораздо увереннее, чем если бы вы просто использовали навыки, опыт и вовлеченность разработчиков.

В конечном счете, проектирование на основе тестируемости развязывает вам руки, позволяя в будущем модифицировать системы как угодно и когда угодно.

Развертываемость: масштабирование разработки систем

Тестируемость — это свойство, которое можно использовать, чтобы эффективнее управлять разработкой для более грамотного проектирования систем. Это свойство работает в любом приближении, но есть еще одно, действующее на уровне всей системы: *развертываемость*.

Подход к программной разработке, который мне ближе всего, — это *непрерывная доставка*. Ее суть в том, чтобы ПО всегда было готово к релизу. Готовность определяется *пайплайнами развертывания*. Такие пайплайны автоматизируют большую часть процесса развертывания и дают представление о готовности системы к развертыванию как таковому. Если пайплайн развертывания успешно проходит все проверки, то ПО по определению готово к релизу, и все, что позволяет определять такую готовность, находится внутри этого пайплайна.

Если пайплайн развертывания определяет готовность ПО к релизу, то его область действия — «от коммита до готовности к релизу». Если релиз-кан-

дидат успешно проходит через пайплайн, он уже *безусловно* готов к развертыванию. Если же в конце пайплайна нужно дополнительно тестировать компоненты или подсистемы на взаимодействие с другими компонентами или подсистемами, то такой пайплайн совершенно бесполезен для определения «готовности компонента к релизу».

Чтобы обеспечить высокое качество системы, требуется ее исчерпывающая оценка. То есть необходимо оценивать именно те изменения, которые в случае успеха и будут развернуты на продакшен. Более того, для повышения надежности оценки требуется повысить степень ее однозначности, то есть *детерминированность*. Таким образом, задача состоит в том, чтобы оценивать именно тот код, который пойдет на продакшен, и обеспечить максимально детерминированное поведение этого кода. В таком пайплайне любое тестирование будет приводить к одинаковому результату после каждого запуска данной версии кода.

Из этих соображений можно заключить, что область действия пайплайна развертывания имеет свои ограничения — в частности, она ограничена *одной независимо развертываемой единицей ПО*. Такой единицей может быть хоть отдельный микросервис, хоть целая корпоративная система, но ее развертываемость — единственный однозначный критерий ее оценки.

Заключение

Когда разработчики обдумывают программную архитектуру, они часто вспоминают об иных, нежели упомянутые выше, важных свойствах системы — о безопасности, масштабируемости и отказоустойчивости. Однако в итоге степень важности каждого из этих свойств зависит от степени успешности бизнеса, в котором функционирует система и работают ее разработчики, а также от общего уровня зрелости системы.

Если зациклиться на масштабируемости и отказоустойчивости, а получить всего трех пользователей, то вся работа будет проделана впустую. Если сосредоточиться на общей безопасности, но при этом программа никогда не будет использоваться за пределами корпоративного интранета (то есть закрытой среды), то эти усилия тоже будут бессмысленными. Такой подход равносилен перепроектированию системы, когда разработчики стараются удовлетворить все возможные потребности, которые, впрочем, могут и не возникнуть.

Если же, напротив, вооружиться инженерным подходом к решению проблем и эволюционным подходом к архитектуре и проектированию, то можно начать разработку раньше, а систему адаптировать под текущие потребности точнее — и это будет гораздо эффективнее.

Выстраивание работы вокруг принципов тестируемости и развертываемости помогает сохранять гибкость системы по мере ее развития. Если система модульна, в ней продуманы абстракции и она в целом хорошо структурирована, то, когда понадобится улучшить безопасность, это сделать будет проще. Когда понадобится сделать код более отказоустойчивым, можно внедрить тестирование по принципу хаос-инжиниринга и посмотреть, как этого добиться.

С более технической точки зрения можно сказать, что улучшение тестируемости и развертываемости (особенно тестируемости) подталкивает инженеров проектировать системы так, чтобы в них были более эффективно разделены необходимая и непреднамеренная сложность. Такой подход, в свою очередь, сохраняет больше пространства для маневров — даже кардинальных — в отношении других архитектурных свойств системы и, как следствие, расширяет возможности для создания высококачественной и продуманной архитектуры.

ГЛАВА 4

Повышение качества архитектуры с помощью индекса зрелости модульности (MMI)

Доктор Карола Лилиенталь

За последние 20 лет было вложено немало времени и средств в программные системы, реализованные на современных языках программирования, таких как Java, C#, PHP и т. д. В проектах разработки приоритеты часто смещались в сторону быстрой реализации функций, а не качества архитектуры. Такая практика привела к росту объема технического долга — излишней сложности, которая со временем только накапливается и требует дополнительных затрат. Сегодня такие системы справедливо называют унаследованными (legacy), поскольку их сопровождение и развитие обходятся дорого, они трудоемки и нестабильны.

В этой главе мы обсудим, как измерить объем технического долга в программной системе с помощью MMI — индекса зрелости модульности (modularity maturity index). Индекс MMI для кода или приложений в ИТ-ландшафте служит руководством для менеджмента и команд при принятии решения о том, каким программным системам нужен рефакторинг или замена, а с какими и так все хорошо. Цель — определить, какой технический долг нужно устранить, чтобы архитектура стала устойчивой, а обслуживание — экономичнее.

Технический долг

Термин *технический долг* был введен Уордом Каннингемом (Ward Cunningham) в 1992 году: «Технический долг возникает, когда сознательно или бессознательно принимаются неверные или слабые технические решения.

Эти неверные или слабые решения впоследствии добавляют объем работы, из-за чего обслуживание и развитие становятся дороже»[1]. Как только вы принимаете плохое решение, вы сразу же начинаете накапливать технический долг, который придется гасить с процентами, чтобы ликвидировать.

В этом разделе я обозначу два типа технического долга, но сосредоточусь на том, который выявляется в ходе ревью архитектуры, — долге реализации.

Долг реализации

Исходный код содержит признаки так называемого запаха кода, например длинные методы и пустые catch-блоки. По большей части долг реализации теперь можно обнаружить в исходном коде автоматически с помощью различных инструментов. Каждая команда разработчиков должна постепенно устранять этот долг в повседневной работе, не требуя на это дополнительного бюджета.

Долг проектирования и архитектуры

Проектирование классов, пакетов, подсистем, слоев и модулей, а также зависимостей между ними часто реализуется непоследовательно, сложно и с отклонениями от запланированной архитектуры. Такой долг нельзя просто посчитать или измерить; для него требуется обширный анализ архитектуры, который представлен в разделе «Ревью архитектуры для определения MMI» на с. 88.

Прочие проблемные области, которые также можно рассматривать как долг по программным проектам (например, отсутствие документации, низкое покрытие тестами, неудобство использования или неадекватное оборудование), здесь не рассматриваются, поскольку к техническому долгу они не относятся.

Возникновение технического долга

Рассмотрим возникновение технического долга и его возможные эффекты. Если в начале проекта разработки ПО архитектура этого ПО была спроектирована на высоком качественном уровне, то можно предположить, что сопровождать такую программную систему на начальном этапе будет несложно. На нем программная система находится в коридоре низкого технического долга при тех же усилиях по ее сопровождению (см. стрелки, ведущие вниз, на рис. 4.1).

---

[1] Уорд Каннингем, «The WyCash Portfolio Management System: Experience Report», OOPSLA '92, Ванкувер, Британская Колумбия, 1992.

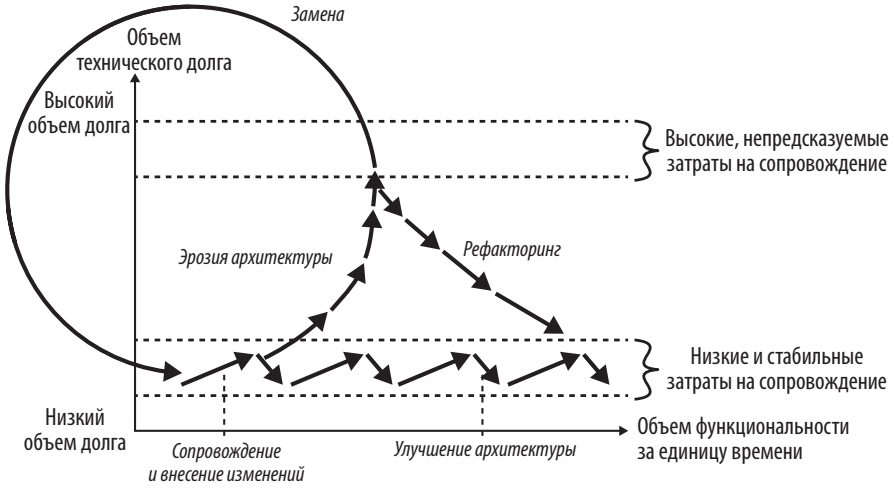


Рис. 4.1. Возникновение технического долга и его возможные эффекты

Если в процессе обслуживания и интеграции изменений система постоянно расширяется, в ней неизбежно возникает технический долг (на рис. 4.1 стрелки, направленные вверх). Программная разработка — это постоянный процесс изучения условий использования системы, где первая ее итерация редко является окончательной. Пересмотр архитектуры (улучшение, показанное стрелками вниз) должен проводиться регулярно. Это обеспечивает постоянное чередование процессов сопровождения/внесения изменений и улучшения архитектуры.

Если команда может постоянно чередовать процессы расширения и улучшения архитектуры, система будет оставаться в коридоре низких и стабильных затрат на сопровождение. К сожалению, этот аспект появился в повестке дня для многих руководителей, отвечающих за финансовые вопросы, лишь в последние годы — слишком поздно для большинства систем, запущенных в начале 2000-х годов.

Если команде разработчиков не разрешается неуклонно сокращать технический долг, то со временем неизбежно начинается процесс *эрозии архитектуры*, как показано стрелками, выходящими из коридора низких и стабильных затрат на сопровождение, на рис. 4.1. Как только технический долг накапливается, сопровождение и изменение ПО становится все дороже, а последующие ошибки — все труднее для понимания, и это доходит до такой степени, что внесение каждого изменения становится мукой. Рисунок 4.1 показывает, как постепенно ухудшается ситуация, — стрелки, направленные вверх, становятся все короче. По мере увеличения

долга все меньше и меньше функциональности удается реализовать за единицу времени.

Существует два способа выйти из дилеммы технического долга.

Рефакторинг

Можно провести рефакторинг унаследованной системы «изнутри», чтобы снова повысить скорость разработки и уровень стабильности. На этом, как правило, трудном пути систему необходимо шаг за шагом вернуть в коридор низких и стабильных затрат на сопровождение (см. стрелки с пометкой «Рефакторинг» на рис. 4.1).

Замена

Также можно заменить унаследованную систему другим ПО с меньшим техническим долгом (см. круг на рис. 4.1).

Конечно, может оказаться так, что в начале проекта сильной команды не было. В этом случае технический долг возникает с самого старта разработки и постоянно растет. Про такие программные системы можно сказать, что они развивались в неблагоприятных условиях. В долгосрочной перспективе система в таком состоянии не будет устраивать ни разработчиков, ни руководство.

Такой взгляд на технический долг понятен большинству руководителей, отвечающих за финансовые вопросы. Ни у кого нет желания накапливать технический долг и постепенно вязнуть в разработке, чтобы в итоге каждое изменение превращалось в статью непредсказуемых расходов. Несложно понять, что для поддержания низкого уровня технического долга на протяжении всего жизненного цикла ПО работать нужно постоянно. Проблему хорошо осознает даже большинство далеких от ИТ людей, но есть вопрос: как же все-таки оценить размер технического долга в программной системе?

Оценка через MMI

Моя докторская диссертация по архитектуре и когнитивистике, а также результаты более чем трехсот архитектурных оценок позволили нам с командой создать единую схему оценки под названием «MMI» (modularity maturity index, индекс зрелости модульности) для сравнения технического долга, накопленного в архитектуре различных систем.

Согласно когнитивистике, в ходе эволюции человеческий мозг выработал отличные механизмы, которые помогают ему справляться со сложными

структурами, такими как организация систем, планировка городов и стран, визуализация генеалогии и т. д. Программные системы, несомненно, также являются сложными структурами из-за своего размера и количества элементов. В своей диссертации я связала выводы когнитивной психологии о трех механизмах, которые мозг использует для работы со сложностью (группирование, построение иерархий и создание схем), с важными принципами программной архитектуры и проектирования (модульность, иерархичность и соответствие паттернам). В этой главе я могу лишь кратко объяснить эти взаимосвязи. Все подробности, особенно касающиеся когнитивной психологии, можно найти в моей книге[1], которую я писала на основе докторской диссертации.

У упомянутых принципов программных систем есть одно важнейшее свойство: для работы со сложными структурами они опираются на механизмы мозга. Архитектуры и проекты, построенные по этим принципам, люди воспринимают как целостные и понятные, такие проекты проще обслуживать и развивать. Соответственно, эти принципы следует использовать в программных системах, чтобы их обслуживание и развитие можно было выполнять быстро и практически безошибочно. Цель такого использования в том, чтобы программные системы можно было развивать в течение длительного времени без снижения качества и скорости разработки даже при смене команд разработчиков.

Модульность

В разработке ПО *модульность* — это принцип, который ввел Дэвид Парнас (David Parnas) в 1970-х годах. Д. Парнас утверждал, что модуль должен содержать только одно проектное решение (инкапсуляция), а структура данных для этого решения должна быть инкапсулирована в пределах того же модуля (локальность)[2].

В современных языках программирования модули — это единицы внутри программной системы, такие как классы, компоненты или слои. Наш мозг предпочитает рассматривать системы с разных уровней абстракции — это позволяет ему таким образом увеличивать объем рабочей памяти. Ключевой момент здесь в том, что мозг выигрывает от использования модулей только тогда, когда информацию о них можно представить в виде

---

[1] Карола Лилиенталь (Carola Lilienthal), «Sustainable Software Architecture», докторская диссертация, Dpunkt.verlag, 2019.

[2] Дэвид Парнас (David Parnas), «On the Criteria to be Used in Decomposing Systems into Modules», Communications of the ACM 15, № 12 (1972): 1053–1058.

целостной и осмысленной сущности. Исходя из этого модули, в которых собраны произвольные, не связанные между собой элементы, являются бессмысленными, а потому восприниматься не будут. Соответственно, у системы с целостными и осмысленными модулями будет низкий уровень технического долга и избыточной сложности.

Насколько модули представляют собой целостные и осмысленные элементы в архитектуре ПО, можно оценить только качественно, и для такой оценки имеются различные проверки, о которых мы поговорим ниже.

Связность через связанность

Модули должны содержать логически взаимосвязанные подмодули, то есть уровень их внутренних взаимосвязей (связность) должен быть высоким, а внешних (связанность) — низким. Например, если подмодули одного модуля имеют более высокий уровень связанности с другими модулями, чем со своими «сестрами и братьями», то уровень их взаимной связности низок, следовательно, модульность реализована плохо. В отличие от связности, связанность можно измерить.

Имена

Если архитектура программной системы модульная, то для каждого модуля необходимо понимать, какова его задача. Важно, чтобы у программного модуля в действительности была только одна задача. Хороший индикатор четкости зон ответственности модулей — это имена, из которых должны считываться их задачи. Если имя сформулировано расплывчато, на модуль стоит обратить внимание. К сожалению, этот признак не поддается измерению.

Пропорциональность

Модули одного уровня, такие как слои, компоненты, пакеты, классы и методы, должны быть пропорциональны относительно друг друга. С этой точки зрения стоит изучить очень большие модули, чтобы определить, нужно ли запланировать их декомпозицию. Яркий пример приведен на рис. 4.2. На круговой диаграмме слева показаны размеры девяти модулей сборки: команда сообщила нам, что каждая единица — это запланированный в сборку модуль. Одна единица сборки представляет собой самую большую часть диаграммы, что составляет 950 860 строк кода. Остальные восемь частей в сумме составляют всего 84 808 строк кода — явная диспропорция. Справа на рис. 4.2 показана архитектура системы: 9 частей, или единиц, сборки, схематично изображенные в виде квадратов, и дугообразные связи между ними. «Монолит» — это та самая большая часть с круговой диаграммы. Она использует восемь малых

единиц сборки, которые мы здесь называем «Сателлит №...». Темный оттенок квадрата «Монолита» по сравнению со светлым оттенком сателлитов, как и круговая диаграмма, указывает на то, что именно в нем находится бóльшая часть исходного кода. Модульность этой системы диспропорциональна, и этот показатель поддается измерению.

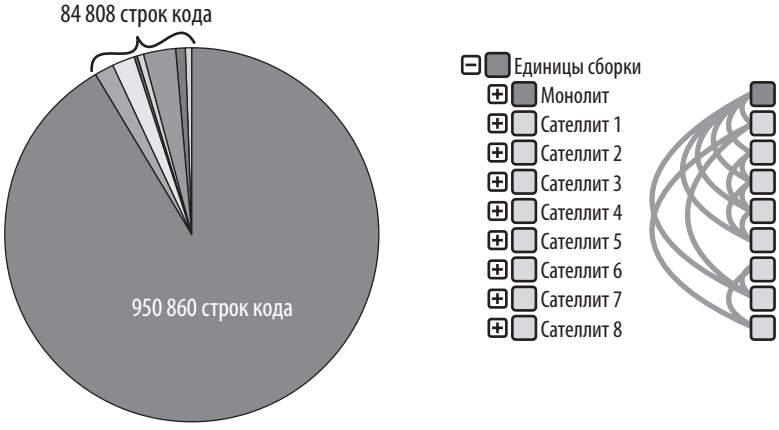


Рис. 4.2. Сильная диспропорция

Подобные оценки можно проводить на всех уровнях, чтобы проверить модульность системы. Степень влияния каждого пункта на расчет MMI описана в разделах «Иерархичность» и «Соответствие паттернам».

Иерархичность

Иерархии играют важную роль в восприятии, понимании сложных структур и хранении знаний. Людям проще усваивать знания, воспроизводить их и ориентироваться в них, если информация представлена в виде иерархических структур. Иерархичность реализуется в языках программирования через *отношения вложенности*: классы находятся в пакетах, пакеты — в других пакетах и, наконец, в проектах или артефактах сборки. Такая иерархичность соответствует ожиданиям наших когнитивных механизмов.

В отличие от отношений вложенности, связи типа *«использовать»* (use) и *«наследовать»* (inherit) можно применять так, что иерархичность не формируется: можно связать любые классы и интерфейсы в коде с помощью таких отношений. Таким образом создаются переплетенные, совершенно не иерархические структуры. Профессионалы в таких случаях говорят о циклических зависимостях классов, пакетов, модулей и восходящих

связей между слоями архитектуры. При проведении ревью архитектуры я встречаю самые разные случаи: от циклических зависимостей на уровне единичных структур до огромных «циклических монстров».

На рис. 4.3 показано облако циклических зависимостей, состоящее из 242 классов. Каждый прямоугольник представляет класс, а линии между ними — связи. Эта циклическая зависимость распределена по 18 каталогам, и для выполнения своих задач все эти классы зависят друг от друга.

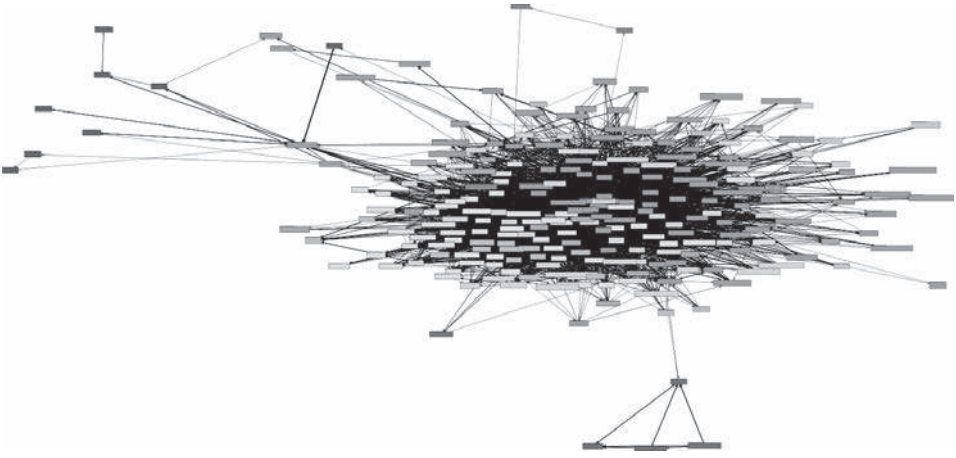


Рис. 4.3. Облако циклических зависимостей из 242 классов

Система, из которой взято облако циклических зависимостей на рис. 4.3, содержит всего 479 классов. Таким образом, более половины всех классов (242) прямо или косвенно зависят друг от друга. Кроме того, у этого облака есть ядро зависимостей и лишь несколько сателлитов. Провести декомпозицию этого облака практически невозможно — для нее требуется масштабное перепроектирование всех классов. Гораздо лучше с самого начала проектировать архитектуру так, чтобы подобные большие циклические зависимости не возникали. К счастью, в большинстве систем ядра циклических зависимостей менее плотные и масштабные, поэтому их можно устранить за несколько сессий рефакторинга.

На рис. 4.4 показана архитектура с неиерархической структурой. Четыре технических слоя небольшой прикладной системы (80 000 строк кода) — App, Services, Entities и Util — расположены друг над другом и, как и положено, взаимодействуют в основном сверху вниз (показано нисходящими стрелками слева). Однако между слоями возникли некоторые обратные связи (показаны восходящими стрелками справа), что приводит к цик-

лическим зависимостям между слоями и, как следствие, к нарушениям архитектуры.

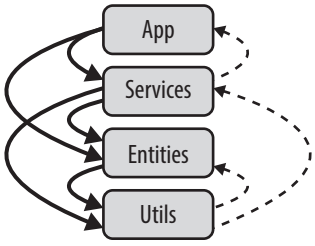


Рис. 4.4. Циклические зависимости на уровне архитектуры

Нарушения в этих слоях архитектуры вызваны всего 16 классами, и их было легко устранить. Таким образом, повторюсь: чем раньше вы обнаружите в слоях своей архитектуры подобные циклические зависимости и сделаете их рефакторинг, тем лучше.

К счастью, циклические зависимости легко вычислить на любых уровнях, а значит, можно точно оценить степень иерархичности системы. Степень влияния каждого пункта на расчет MMI описана в разделе, посвященном соответствию паттернам.

Соответствие паттернам

Схемы — самый эффективный механизм, который люди используют для структурирования сложных взаимосвязей. Схема как абстракция обобщает типичные свойства схожих объектов или связей. Например, если сказать, что некий человек является учителем, то на уровне абстракции в вашей умозрительной схеме возникнут различные предположения и идеи о роде его деятельности: учителя работают в школе, их рабочий день отличается от привычного восьмичасового, они должны проверять домашние задания и контрольные. В частности, в памяти всплывут образы учителей из вашей школы, которые сохранились как прототипы схемы «учитель».

Когда у вас есть схема для того или иного жизненного контекста, вы можете воспринимать и решать вопросы и проблемы гораздо быстрее. Например, паттерны проектирования, широко распространенные в программной разработке, задействуют способность человеческого мозга работать со схемами. Если разработчик уже работал с паттерном проектирования и составил по нему схему, он сможет быстрее распознавать и понимать фрагменты кода и структуры, где этот шаблон применяется.

Использование схем дает в повседневной жизни решающее преимущество в скорости понимания сложных структур. Именно поэтому много лет назад паттерны программной разработки получили широкое распространение. Для разработчиков и архитекторов оказалось важным наличие паттернов, которые можно было бы распознавать и систематически применять в исходном коде. Следовательно, систематическое применение паттернов помогает справляться со сложностью исходного кода.

На рис. 4.5 слева показана диаграмма, разработанная командой для фиксации паттернов проектирования в инструменте, который позволяет объединять классы одного шаблона в слой. Справа исходный код разделен на эти паттерны проектирования, и слева от оси видно множество нисходящих дуг, соответствующих диаграмме. Дуги слева от оси нисходящие, а справа (их немного) — восходящие, против иерархии слоев; и это нормально, поскольку паттерны проектирования образуют иерархические структуры. Паттерны проектирования в этой системе реализованы хорошо, поскольку в ней преобладают нисходящие связи (слева) и очень мало восходящих связей (справа).

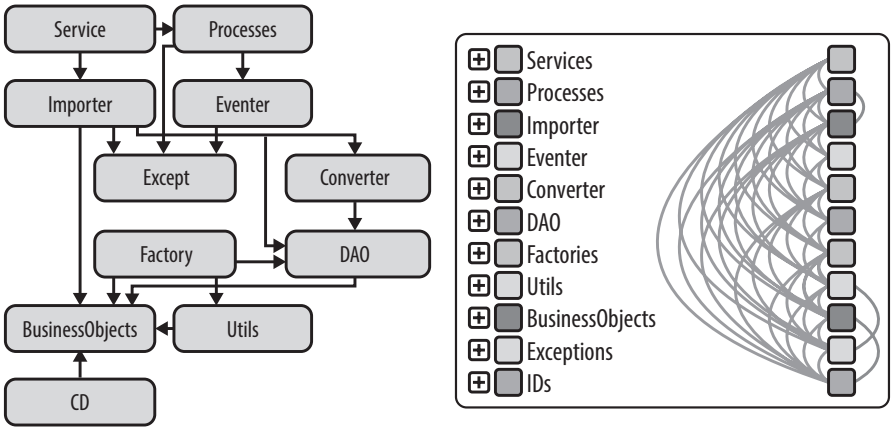


Рис. 4.5. Паттерны на уровне классов = язык паттернов

Анализ паттернов в исходном коде — обычно самая интересная часть анализа архитектуры. В его ходе каждый раз важно понимать уровень архитектуры, на котором в данный момент работает команда разработчиков. Классы, реализующие отдельные паттерны, часто распределены по пакетам или каталогам. При моделировании паттерна, показанного справа на рис. 4.5, этот уровень архитектуры можно визуализировать и сделать доступным для анализа.

Измерить напрямую соответствие паттернам (подобно иерархичности) нельзя, но в разделе ниже будут приведены некоторые метрики, которые используются для его оценки в MMI.

Расчет MMI

MMI вычисляется на основе комплекса критериев и метрик, с помощью которых оцениваются три принципа: модульность, иерархичность и соответствие паттернам. В итоговый расчет MMI эти три принципа входят с разным процентным весом, и для каждого из них рассчитываются свои критерии согласно табл. 4.1. Наибольшее влияние на MMI оказывает модульность (45 %), поскольку она служит основой для иерархичности и соответствия паттернам; и по этой же причине индекс и получил такое название[1].

Критерии из табл. 4.1 определяются с помощью инструментов сбора метрик, инструментов анализа архитектуры или на основе экспертного заключения ревьюера (см. столбец «Способ определения»). Есть масса инструментов, позволяющих получать точные значения метрик, но для того, чтобы эти значения можно было использовать в расчете индекса MMI, необходимо учитывать, как именно реализована та или иная метрика в каждом конкретном инструменте. Заключение ревьюера измерить невозможно, поэтому оно остается на его же усмотрение. Конечно, показатели инструментов анализа архитектуры измеримы, но они также сильно зависят от выводов ревьюера.

В моей практике для оценки таких не поддающихся измерению критериев аналитики проводятся обсуждения с разработчиками и архитекторами. Эти обсуждения проходят в формате очных или удаленных воркшопов. Мы также изучаем систему с разных точек зрения на архитектуру, используя инструменты анализа архитектуры[2]. Для максимальной согласованности результатов заключений анализ всегда проводится вдвоем, а затем его результаты обсуждаются в более широком кругу ревьюеров архитектуры.

MMI рассчитывается так: для каждого критерия по табл. 4.2 определяется число от 0 до 10. Полученные для каждого раздела значения суммируются и делятся на количество критериев из табл. 4.1. Затем результат по каждому принципу умножается на его процентный вес, и итоговый MMI также представляет собой число от 0 до 10.

---

[1] Lilienthal, Sustainable Software Architecture.

[2] В своем анализе я использую Sotograph, Sonargraph, Lattix, Structure101 и TeamScale.

Таблица 4.1. Индекс зрелости модульности (MMI)

Категория	Подкатегория	Критерии	Способ определения
1. Модульность (45 %)			
	1.1. Модуляризация домена и технических слоев (25 %)		
		1.1.1. Распределение исходного кода по модулям домена, % от общего объема кода	Инструменты анализа архитектуры
		1.1.2. Распределение исходного кода по техническим слоям, % от общего объема кода	Инструменты анализа архитектуры
		1.1.3. Соотношение размеров модулей домена [(LoC макс./ LoC мин.)/кол-во]	Инструменты сбора метрик
		1.1.4. Соотношение размеров технических слоев [(LoC макс./ LoC мин.)/кол-во]	Инструменты сбора метрик
		1.1.5. Четко определенные зоны ответственности у модулей доменов, технических слоев, пакетов и классов	Ревьюеры
		1.1.6. Отображение технических слоев и модулей доменов через пакеты/пространства имен или проекты	Ревьюеры
	1.2. Внутренние интерфейсы (10 %)		
		1.2.1. Наличие интерфейсов у доменных или технических модулей (% нарушений)	Инструменты анализа архитектуры
		1.2.2. Отображение внутренних интерфейсов через пакеты/пространства имен или проекты	Ревьюеры
	1.3. Пропорции (10 %)		
		1.3.1. % исходного кода в крупных классах	Инструменты сбора метрик
		1.3.2. % исходного кода в крупных методах	Инструменты сбора метрик
		1.3.3. % классов в крупных пакетах	Инструменты сбора метрик

Таблица 4.1 (*окончание*)

Категория	Подкатегория	Критерии	Способ определения
		1.3.4. % методов системы с высокой цикломатической сложностью	Инструменты сбора метрик
2. Иерархичность (30 %)			
	2.1. Разделение на технические слои и слои домена (15 %)		
		2.1.1. Количество архитектурных нарушений при разделении на технические слои (%)	Инструменты анализа архитектуры
		2.1.2. Количество архитектурных нарушений при разделении модулей домена на слои (%)	Инструменты анализа архитектуры
	2.2. Циклические зависимости классов и пакетов (15 %)		
		2.2.1. Количество классов в циклических зависимостях (%)	Инструменты сбора метрик
		2.2.2. Количество пакетов в циклических зависимостях (%)	Инструменты сбора метрик
		2.2.3. Количество классов на циклическую зависимость	Инструменты сбора метрик
		2.2.4. Количество пакетов на циклическую зависимость	Инструменты сбора метрик
3. Соответствие паттернам (25 %)			
		3.1. Распределение исходного кода по паттернам, % от общего объема	Инструменты анализа архитектуры
		3.2. Отсутствие циклических зависимостей в связях между паттернами (% нарушений)	Инструменты анализа архитектуры
		3.3. Явное отображение паттернов (через имена классов, наследование, аннотации)	Ревьюеры
		3.4. Разделение исходного кода между доменом и техническими слоями (DDD, Quasar, Hexagonal)	Ревьюеры

Таблица 4.2. Подробная инструкция по расчету MMI

Разд.	0	1	2	3	4	5	6	7	8	9	10
1.1.1	<=54 %	>54 %	>58 %	>62 %	>66 %	>70 %	>74 %	>78 %	>82 %	>86 %	>90 %
1.1.2	<=75 %	>75 %	>77,5 %	>80 %	>82,5 %	>85 %	>87,5 %	>90 %	>92,5 %	>95 %	>97,5 %
1.1.3	>=7,5	<7,5	<5	<3,5	<2,5	<2	<1,5	<1,1	<0,85	<0,65	<0,5
1.1.4	>=16,5	<16,5	<11	<7,5	<5	<3,5	<2,5	<2	<1,5	<1,1	<0,85
1.1.5	Нет					Частично					Да, все
1.1.6	Нет					Частично					Да
1.2.1	>=6,5 %	<6,5 %	<4 %	<2,5 %	<1,5 %	<1 %	<0,65 %	<0,4 %	<0,25 %	<0,15 %	<0,1 %
1.2.2	Нет					Частично					Да
1.3.1	>=23 %	<23 %	<18 %	<13,5 %	<10,5 %	<8 %	<6 %	<4,75 %	<3,5 %	<2,75 %	<2 %
1.3.2	>=23 %	<23 %	<18 %	<13,5 %	<10,5 %	<8 %	<6 %	<4,75 %	<3,5 %	<2,75 %	<2 %
1.3.3	>=23 %	<23 %	<18 %	<13,5 %	<10,5 %	<8 %	<6 %	<4,75 %	<3,5 %	<2,75 %	<2 %
1.3.4	>=3,6 %	<3,6 %	<2,6 %	<1,9 %	<1,4 %	<1 %	<0,75 %	<0,5 %	<0,4 %	<0,3 %	<0,2 %
2.1.1	>=6,5 %	<6,5 %	<4 %	<2,5 %	<1,5 %	<1 %	<0,65 %	<0,4 %	<0,25 %	<0,15 %	<0,1 %
2.1.2	>=14 %	<14 %	<9,6 %	<6,5 %	<4,5 %	<3,2 %	<2,25 %	<1,5 %	<1,1 %	<0,75 %	<0,5 %
2.2.1	>=25 %	<25 %	<22,5 %	<20 %	<17,5 %	<15 %	<12,5 %	<10 %	<7,5 %	<5 %	<2,5 %
2.2.2	>=50 %	<50 %	<45 %	<40 %	<35 %	<30 %	<25 %	<20 %	<15 %	<10 %	<5 %
2.2.3	>=106	<106	<82	<62	<48	<37	<29	<22	<17	<13	<10
2.2.4	>=37	<37	<30	<24	<19	<15	<12	<10	<8	<6	<5
3.1	<=54,5 %	>54,5 %	>59 %	>63,5 %	>68 %	>72,5 %	>77 %	>81,5 %	>86 %	>90,5 %	>95 %
3.2	>=7,5 %	<7,5 %	<5 %	<3,5 %	<2,5 %	<2 %	<1,5 %	<1,1 %	<0,85 %	<0,65 %	<0,5 %
3.3	Нет					Частично					Да
3.4	Нет					Частично					Да

На рис. 4.6 показана выборка из 18 программных систем, которые оценивались в течение 5 лет (ось X). Для каждой системы размер показан в LoC, то есть в строках кода (размер точки), а MMI — по шкале от 0 до 10 (ось Y).

Если система имеет оценку от 8 до 10, доля технического долга в ней невелика. Система находится в коридоре низких и стабильных затрат на сопровождение (см. рис. 4.1). Системы на рис. 4.6 с оценкой от 4 до 8 уже накопили довольно много технического долга. Чтобы улучшить их качество, необходимо провести несколько сессий соответствующего рефакторинга. Системы с оценкой ниже 4 можно сопровождать и расширять, только приложив большие усилия (см. рис. 4.1, коридор высоких и непредсказуемых затрат на сопровождение). В отношении этих систем необходимо тщательно взвесить, стоит ли проводить их модернизацию через рефакторинг или их проще заменить.

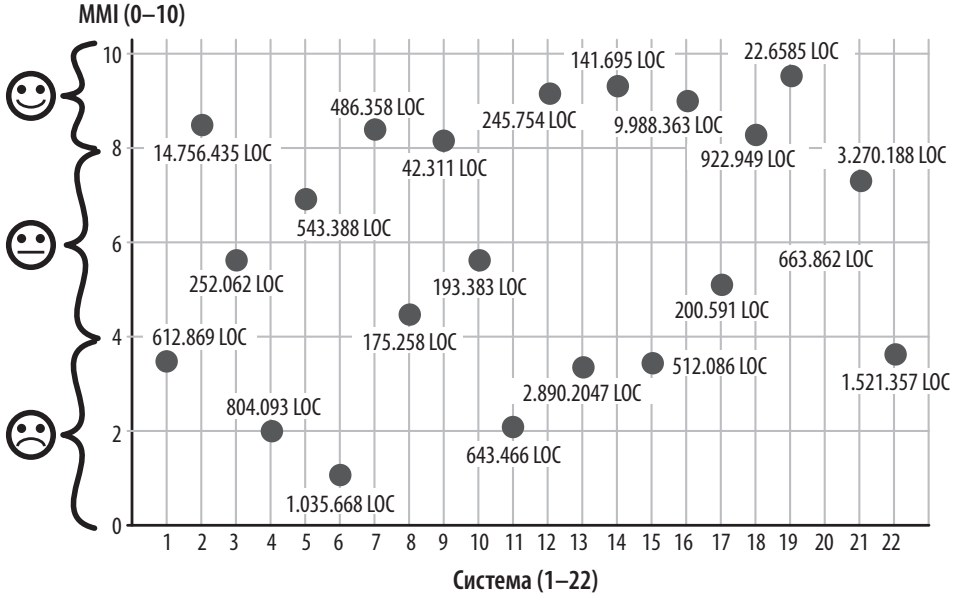


Рис. 4.6. MMI для различных систем

Ревью архитектуры для определения MMI

Большинство команд разработчиков могут без труда перечислить список долгов по архитектуре и дизайну системы, над которой они работают. Этот список — хорошая отправная точка для анализа технического долга. Чтобы разобраться в причинах долгов по архитектуре и дизайну,

рекомендуется провести ревью архитектуры. Оно позволяет сопоставить, насколько запланированная целевая архитектура соответствует фактической, которая, по сути, реализована в исходном коде (рис. 4.7). Целевая архитектура — это план, который существует на бумаге или в сознании архитектора и разработчиков. Сегодня для такого сопоставления целевой и фактической архитектур доступно несколько хороших инструментов, включая Lattix, Sotograph/SotoArc, Sonargraph, Structure101 и TeamScale.

Как правило, фактическая архитектура, в том виде как она реализована в исходном коде, отличается от целевой архитектуры, зафиксированной в плане, — и тому есть множество причин. Зачастую отклонения возникают незаметно, поскольку среды разработки дают лишь локальное представление об исходном коде, который подвергается обработке, и не формируют общую картину. К отклонениям приводит и недостаток знаний об архитектуре у разработчиков. В некоторых случаях отклонения вносятся намеренно, поскольку команда работает в условиях цейтнота и ей нужно быстрое решение, при этом необходимый рефакторинг откладывается на неопределенное время.

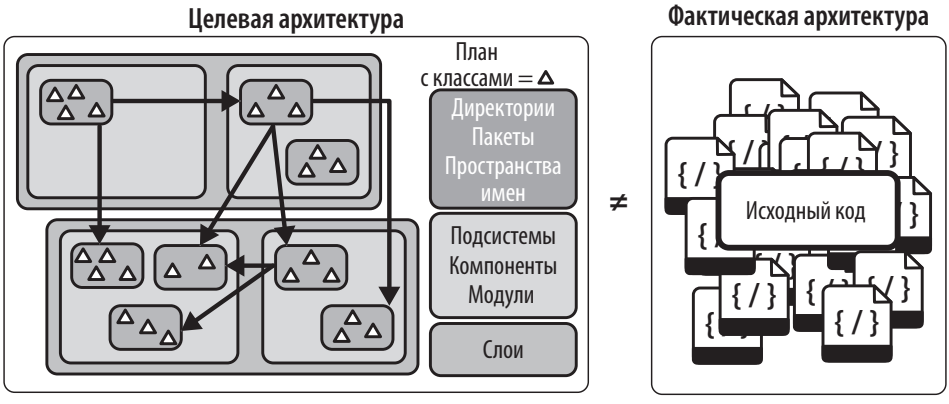


Рис. 4.7. Сравнительный анализ целевой и фактической архитектуры

На рис. 4.8 показана последовательность ревью архитектуры для выявления размера технического долга. Ревью архитектуры проводят ревьюер совместно с архитекторами и разработчиками системы в формате воркшопа. В начале воркшопа проводят парсинг исходного кода системы с помощью инструмента анализа (1) и определяют ее фактическую архитектуру. Затем на основе фактической архитектуры моделируется целевая так, чтобы их можно было сравнить (2).

Таким образом проясняется размер технического долга, и ревьюер вместе с командой разработчиков ищут простые решения, как с помощью рефакторинга привести фактическую архитектуру в соответствие с целевой (3). Бывает и так, что в ходе обсуждения ревьюер и разработчики понимают, что фактически реализованное в исходном коде решение оказывается лучше запланированного.

Иногда ни целевая, ни отличная от нее фактическая архитектура лучшими решениями не являются, и тогда ревьюеру с командой приходится совместно проектировать новую целевую архитектуру. Далее в ходе ревью они составляют список задач технического долга и возможного рефакторинга (4). И наконец, изучают прочие метрики (5), чтобы найти оставшийся технический долг, например большие классы, слишком сильную связанность, циклические зависимости и т. д.

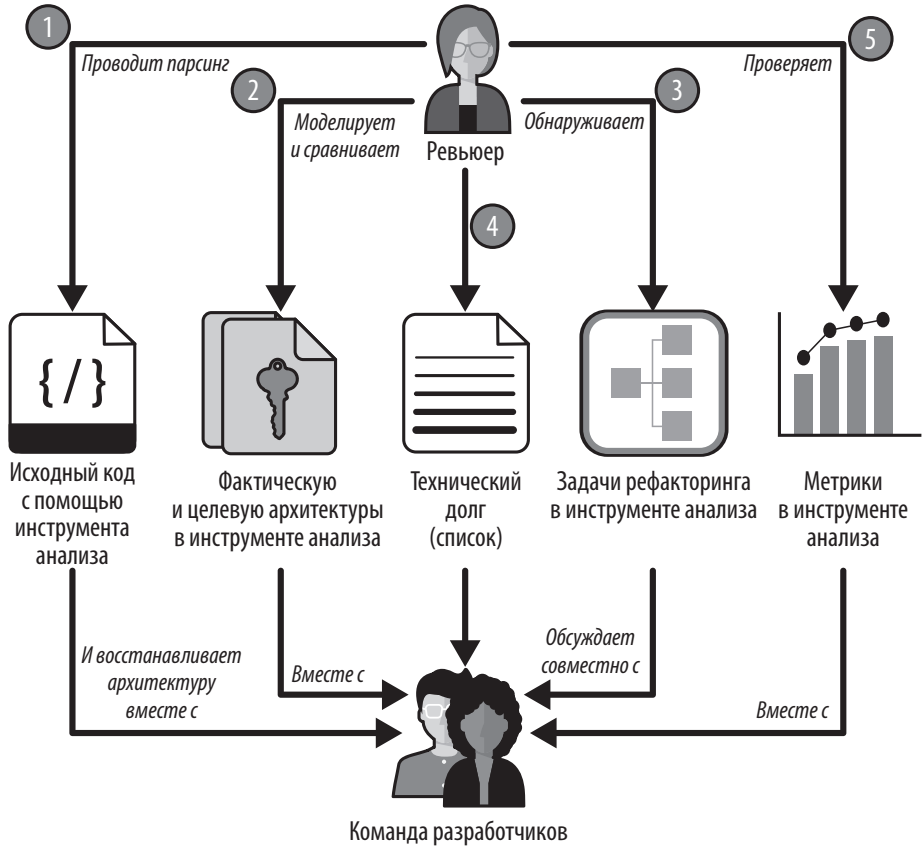


Рис. 4.8. Анализ архитектуры для определения MMI

Заключение

MMI определяет размер технического долга в унаследованной системе. По результатам оценки модульности, иерархичности и соответствия паттернам определяется необходимость рефакторинга или даже замены системы. Если MMI системы меньше 4, следует рассмотреть вариант замены текущей системы на другую, с меньшим техническим долгом. Если система получает MMI от 4 до 8, ее обновление, как правило, обходится дешевле замены. В этом случае команда разработчиков вместе с ревьюером определяет и приоритизирует задачи рефакторинга для сокращения технического долга. Эти задачи нужно поэтапно включать в план обслуживания или развития системы, а результаты — регулярно проверять. Таким образом система может постепенно сместиться в область низкозатратного постоянного текущего сопровождения.

Приятнее всего ревьюерам работать с системой, MMI которой выше 8. Как правило, это означает, что команда и ее архитекторы хорошо поработали и могут заслуженно гордиться своей архитектурой. Мы же в таких случаях очень рады, что можем дать положительную оценку их работе через MMI.

ГЛАВА 5

Локальные сборки и метрики: инструменты для выживания в переходный период к DevOps

Кристиан Чичери

Многие считают архитектуру ПО ремеслом, но я рассматриваю ее скорее как науку. Ученые, как правило, измеряют все, что можно, и на основе этих измерений строят дальнейшие рассуждения. Даже если получить точные цифры невозможно, математические подходы к архитектуре ПО все равно опираются на те величины, которые можно измерить, — метрики и показатели. Иногда применение таких подходов зависит от того, какие метрики в конкретной ситуации целесообразны, а какие — нет. Как убедиться, что ключевые показатели эффективности (KPI) дают именно ту информацию, которая позволит организации решить, стоит ли вкладывать время и усилия в рассматриваемый проект?

Чтобы получить качественные метрики, нужно грамотно выстроить систему и проделать большую работу. Но система вашей организации может быть не лучшего качества или в ней могут быть не внедрены передовые метрики на основе лучших практик DevOps. DevOps — это культурный сдвиг. Риск понять эту концепцию неправильно велик, да и компании не всегда готовы полноценно внедрять лучшие DevOps-практики. И даже если такая цель поставлена, изучение и внедрение лучших практик требует времени. Реальность не всегда идеальна, и стандартные метрики не всегда отражают истинную проблему.

Что же делать, если вы хотите внедрить лучшие практики, а организация к этому еще не готова? В таких не самых идеальных условиях, я считаю, все равно полезно и важно иметь набор практик и метрик, которые помогут

«пережить» переходный период и сохранить продуктивность. Именно об этом и пойдет речь в данной главе.

Я покажу вам несколько примеров из реальных проектов и обосную пользу локальных сборок и метрик, которые вам помогут. Вы увидите, как метрики и локальные сборки могут быть эффективны в следующих ситуациях:

- слабое взаимодействие между командами DevOps и QA-инженеров;
- непродуктивные циклы обратной связи;
- чрезмерная зависимость от автоматизации без ее глубокого понимания;
- отсутствие чувства ответственности за проверки и автоматизацию.

Как консультант, я постоянно сталкивался с этими и другими распространенными антипаттернами. В этой главе представлены метрики, которые помогут командам определить приоритеты при возникновении подобных сценариев и составить дорожную карту для безболезненного улучшения процесса разработки.

Ключевые термины

Распространение Agile-движения, и, в частности, экстремального программирования (XP), сместило центр внимания разработки на автоматизацию. Идея, которую в 2011 году четко изложил Мартин Фаулер[1], заключается в том, что действия, которые «причиняют боль» (то есть отнимают много времени или усилий), следует выполнять как можно чаще. Это позволяет собирать больше обратной связи, нарабатывать практику и дробить процессы на более мелкие этапы. Такие действия следует автоматизировать[2].

CI/CD

Кроме того, в 2006 году Фаулер высказался в пользу непрерывной интеграции (CI, constant integration), которую он определяет как *практику* разработки: «Непрерывная интеграция — это практика разработки ПО, при которой члены команды интегрируют результаты своей работы в систему с высокой частотой. Как правило, каждый интегрирует результаты минимум раз в день, в итоге получается несколько интеграций в день.

---

[1] Мартин Фаулер. «Frequency Reduces Difficulty», MartinFowler.com, 28 июля 2011 г.

[2] Конечно, не только эти действия, но с них целесообразно начинать.

Каждая интеграция проверяется механизмами автоматизированной сборки (включая механизмы тестирования), чтобы как можно быстрее обнаружить ошибки интеграции»[1].

Далее Фаулер развивает идею, рассматривая методологию CI уже как набор практик, где автоматизация — лишь одна из них. Автоматизация также *способствует* повседневной разработке. Ключевой момент здесь (о нем подробнее далее) — это необходимость избегать поломки общей основной ветки кода.

Концепция непрерывной интеграции (CI) была расширена через непрерывную доставку (CD, constant delivery); часто их упоминают вместе: CI/CD[2]/CI — это лишь *первая* часть процесса, в которой обычно участвует только команда разработчиков. Доставка ПО — это сложный процесс с участием различных стейкхолдеров, включая разные технические команды (например, эксплуатации и QA). CD объединяет их всех через автоматизацию. И все же автоматизация лишь *поддерживает* процессы.

Чтобы вы могли полноценно разобрать примеры, важно четко понять, как работает пайплайн CI/CD, поэтому я, с вашего позволения, немного углублюсь в детали. Современное ПО обычно разделено на компоненты, поэтому поздние этапы пайплайна доставки, как правило, включают запуск сложных тестов взаимодействия, чтобы убедиться, что ничего не сломано. Эти тесты выполняются автоматически или вручную. В обоих случаях чем позже обнаруживается дефект, тем дороже, как правило, обходится его исправление. Крайний случай — когда конечные пользователи сообщают о дефекте на продакшен. Если это происходит, к работе подключается служба поддержки: дефект нужно зарегистрировать и, возможно, классифицировать, а необходимость и сроки его исправления определяются в зависимости от срочности.

Дефект на продакшен может нанести ущерб репутации организации. Он негативно влияет на настроение конечных пользователей, особенно если влечет за собой серьезные последствия для них или их бизнеса. Весь процесс устранения дефекта на столь позднем этапе неэффективен по сравнению с другим крайним случаем — обнаружением и исправлением

---

[1] Мартин Фаулер. «Continuous Integration», MartinFowler.com, 1 мая 2006 г.

[2] В CI/CD аббревиатура CD может означать как непрерывную доставку (continuous delivery), так и непрерывное развертывание (continuous deployment) (https://oreil.ly/URy9a). Здесь под CD подразумевается *непрерывная доставка*. Подробнее о различиях между этими двумя концепциями можно узнать в блоге М. Фаулера (https://oreil.ly/8Isic).

на ранней стадии, в локальной среде разработки, еще до его фиксации в общей основной ветке.

DevOps

DevOps — это неологизм, образованный слиянием английских слов development и operations (разработка и эксплуатация). Термин был придуман в 2009 году Патриком Дебуа (Patrick Debois) и стал популярным благодаря хэштегу в Twitter. Концепция DevOps была задумана, чтобы устранить традиционную разобщенность между разработчиками и инженерами по эксплуатации (системными администраторами). Также она расширила концепцию Agile, направленную на сокращение времени между появлением проблемы и ее обнаружением.

Ключевые идеи DevOps

Процесс

Команда системных администраторов и инженеров по эксплуатации должна быть полностью интегрирована с командой разработчиков. Принципы CI/CD предполагают, что код всегда должен находиться в состоянии, готовом к доставке.

Инструменты

Команда, ответственная за готовность приложения к доставке, также выбирает подходящие для работы инструменты. Это означает, что разработчики и инженеры по эксплуатации совместно принимают решения и используют общий набор инструментов.

Культура

Подходы к работе у команд эксплуатации и разработки должны быть согласованы. Например, операционную часть, так же как и код, необходимо поместить под контроль версий (см. концепцию «*инфраструктура как код*» и все, что с ней связано). Средства автоматизации, которые использует команда разработчиков, например автоматические тесты или локальные сборки, необходимо вписать в более широкий контекст, автоматизируя процессы внутреннего контроля сред, развертываний и выполнения кода.

Культура — это главное изменение, которое DevOps привнесло в индустрию ПО. Такой подход к работе означает, что и разработчики, и инженеры по эксплуатации должны выходить за рамки своих дисциплин: разработчикам приходится изучать среду выполнения ПО, а также то, как выявлять

и устранять проблемы в средствах автоматизации и скриптах системы. Инженеры по эксплуатации и системные администраторы должны понимать, как написан код, какова архитектура решения и даже какие у него есть уязвимые места. Они должны уметь писать модульные тесты и отлаживать код. Они должны отвечать за удобство поддержки, требуя от команды разработчиков применять политики логирования и внутреннего контроля.

Таким образом, DevOps — это в первую очередь *культура*, а инструменты и автоматизация являются следствием ее принятия.

Сдвиг ответственности

Идеальный сценарий — когда команды полностью контролируют свои пайплайны и культура DevOps полностью соблюдается. В организации больше нет изолированных подразделений, препятствующих коммуникации, а команды самостоятельно создают и запускают свой код.

Реальность же зачастую далека от идеала. Во многих организациях концепция DevOps никакого отношения к культуре не имеет — этот термин используют для обозначения своего рода «современного системного администратора». От кандидатов на подобную должность требуются навыки эксплуатации, например владение инструментами автоматизации и написание системных скриптов, но редко — столь же глубокие знания в области разработки. Эти сотрудники тратят столько времени на создание и обслуживание средств автоматизации, что термин «DevOps» часто превращается в название команды автоматизации со своими рабочими процессами и системой тикетов. В то же время разработчики почти не знают, где и как происходит развертывание. Они оторваны от процесса автоматизации, но проверки делегируются именно им. Команда QA-инженеров также работает обособленно. Это лишь возрождает изолированность команд, вновь создавая «несоответствие сред» между локальными средами разработки и продакшеном. Я называю этот антипаттерн *сдвигом ответственности* (ownership shift).

Контроль над локальной средой

Широко известно высказывание Мартина Фаулера о важности непрерывной интеграции (CI): «Исправляйте сломанные сборки немедленно»[1]. Это, конечно, замечательная рекомендация, но для идеальных условий

[1] Фаулер М. «Continuous Integration», https://oreil.ly/WXFrv.

DevOps. В реальности же одно из последствий сдвига ответственности заключается в том, что команда разработчиков больше не контролирует механику пайплайна сборки. Вообще рекомендация Фаулера исходит из нескольких допущений, ни одно из которых не выполняется автоматически в неидеальной культуре DevOps. Например:

- Единственный способ сломать сборку — внести изменения в код, то есть в исходный код приложения в основной ветке (в системе контроля версий), а не в создаваемую скриптами инфраструктуру. И следовательно, исправить сборку можно, просто изменив код.
- Члены команды знают, как самостоятельно отлаживать проблемы со сборкой, и имеют необходимые доступы и разрешения, чтобы их устранить.

В той реальности, которую я здесь описываю, эти допущения соблюдаются редко. Когда сборка ломается, а причина при анализе кода неочевидна, разработчики практически не контролируют автоматизацию. Они могут передать проблему «команде DevOps», которая реагирует на тикет в соответствии со своим графиком и приоритетами. Процесс доставки ПО блокируется, и выделенная команда QA-инженеров не может проверить последнюю версию ПО. Таким образом, ПО находится в состоянии, когда его невозможно доставить.

Концептуальная ошибка здесь — рассматривать пайплайн CI/CD (рис. 5.1) или автоматизацию в целом как ядро процесса проверки. Из-за сдвига ответственности, если делегировать все проверки исключительно автоматизации, скорее всего, возникнут все те же проблемы с эффективностью и те же риски.

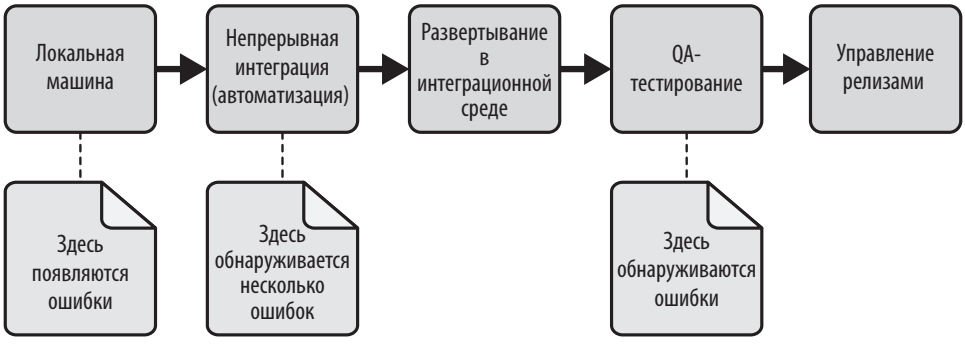


Рис. 5.1. Пайплайн CI/CD

Автоматизация — необходимый шаг, который делает проверку дешевым и повторяющимся процессом. Она *не* должна быть чем-то внешним по отношению к процессу разработки. И я бы предложил перенести центр внимания к проверке туда, где возникают проблемы, — в локальную среду разработки. (Под *локальной* я подразумеваю среду, в которой обычно вносятся изменения, — не обязательно компьютер разработчика, хотя это привычный и удобный вариант.)

Приватная сборка

В своей книге «Continuous Integration: Improving Software Quality and Reducing Risk» [1] авторы Пол М. Дюваль (Paul M. Duvall), Стив Матиас (Steve Matyas) и Эндрю Гловер (Andrew Glover) вводят понятие «приватная сборка». Они утверждают: «Чтобы исключить поломку сборок, разработчики должны эмулировать интеграционную сборку в своей локальной IDE после завершения модульных тестов». Далее они отмечают: «Такая сборка позволяет интегрировать новый рабочий продукт с рабочим продуктом остальных разработчиков, получать изменения из репозитория системы контроля версий и успешно выполнять локальную сборку с учетом последних изменений. Таким образом, код, который каждый разработчик отправляет в репозиторий, вносит вклад в общее дело, и вероятность возникновения сбоя в этом коде на сервере интеграционной сборки снижается».

Приватные сборки *подлежат автоматизации*, но они встречаются реже обычных сборок, поскольку за автоматизацию последних отвечает «команда DevOps», а у нее есть своя среда и график. Кроме того, для приватных сборок обычно нет таких общедоступных сценариев интеграционного тестирования, которые можно было бы легко запустить на локальной машине.

Не очень спасают ситуацию и такие технологии, как контейнеры, и даже те инструменты сборки, которые легко поддаются стандартизации, — автоматизация сохраняет свою сложность, потому что на каждой платформе достаточно своих специфических проблем. К примеру:

- Набор данных для выполнения интеграционного тестирования нужно тщательно подбирать, возможно, потребуется даже постоянное взаимодействие с командой QA.

---

[1] Дюваль Пол М., Матиас С., Гловер Э. «Непрерывная интеграция. Улучшение качества программного обеспечения и снижение риска».

- Механизм контроля данных еще не стандартизирован.
- Миграции базы данных, если они есть, должны запускаться автоматически в процессе сборки, часто способами, отличными от тех, что используются в продакшене.
- Приложения часто состоят из компонентов, как в современной архитектуре микросервисов. Компонентам требуются сложные каналы связи, например брокеры сообщений.
- Некоторые тесты, например сквозные, сложно писать и выполнять на локальной машине (и тут может потребоваться явный «сдвиг ответственности» в сторону команд QA).

Отмечу, что это лишь немногие из причин, по которым полностью автоматизированные приватные сборки часто бывает сложно реализовать, однако сама концепция ценна. Важно не упускать из виду *цель* приватных сборок: предотвратить попадание проблем в общую кодовую базу или на последующие этапы проверки (автоматической или ручной). (Визуализация запуска пайплайна CI/CD с приватной сборкой представлена на рис. 5.2.)

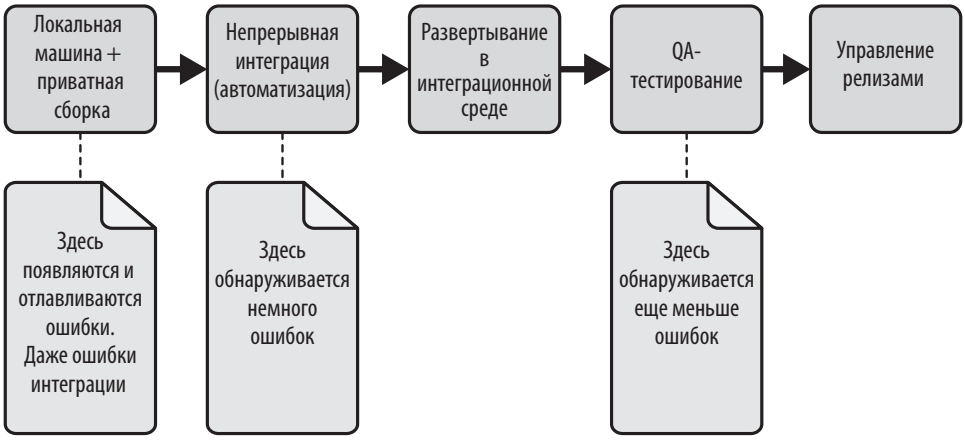


Рис. 5.2. Пайплайн CI/CD, который начинается с приватной сборки

Если какие-то шаги приватной сборки автоматизировать не получится, их придется выполнять вручную, по крайней мере временно. (Не скажу, конечно, что это лучший способ, но лучше уж вручную, чем совсем никак!) Независимо от того, автоматизированы приватные сборки или нет, они остаются необходимым этапом жизненного цикла разработки ПО.

Такие сборки можно выполнять *локально* на машине разработчика или в некой выделенной инфраструктуре, например в выделенной облачной среде. Среда разработки не обязательно должна быть физической машиной или даже полноценной операционной системой. *Удобная* локальная среда — та, что находится под полным контролем разработчика (часто это его собственная машина), чтобы он мог быстро и легко проверять изменения.

Главное, что нужно помнить о приватных сборках, — это то, что они *приватные*[1], то есть выполняются в выделенной среде до того, как вносимые ими изменения попадут в общую основную ветку. Они нужны для того, чтобы не публиковать изменения, которые могут сломать общую сборку или ветку в целом.

Кейс: нестабильная основная ветка

В компании А есть так называемая мультидисциплинарная команда. Но Agile-практики в компании еще не устоялись, поэтому команда не является мультидисциплинарной в том смысле, какой в это понятие вкладывают специалисты по Agile. Команда работает итерациями переменной длины и состоит из представителей отдельных команд: фронтенда, бэкенда, администрирования баз данных (DBA), DevOps и QA.

QA-инженер сообщает, что каждый раз, когда команда QA приступает к тестированию последней версии, ПО не работает. Более того, по его словам, на внесение исправлений в новые версии ПО уходит много времени. Команда решает провести анализ первопричин, рассмотрев все процессы каждой из подкоманд. Чтобы зафиксировать наиболее важные аспекты жалобы QA-инженера, они составляют таблицу (табл. 5.1).

Команда принимает решение сосредоточиться на так называемых *неизбежных ошибках*.

---

[1] Я не использую слово «личный» (personal) вместо «приватный» (private), хотя оно могло бы показаться более понятным. Дело в том, что «личный» по определению относится только к одному человеку, тогда как в разработке может применяться парное или моб-программирование. Как бы то ни было, можно утверждать, что приватные сборки можно называть личными, если рассматривать личную среду как ту, из которой код публикуется в общую основную ветку.

Таблица 5.1. Анализ ошибок

Ошибка	Компонент	Можно ли избежать?	Легко ли автоматизировать?	Чем вызвано?
A1	Фронтенд	Нет	Нет	Непроверенное неопределенное значение, полученное от сервера
A2	Бэкенд	Да (отсутствует тест)	Да	Несоответствие типа данных в API
A3	Интеграция с реальными данными	Нет	Да	Формат реальных данных как пограничный случай
A4	Интеграция	Нет	Нет	Проблема с обратной совместимостью

Ошибка A1

Команда выясняет, что ошибка A1 попала в основную ветку незамеченной, потому что фронтенд-разработчик тестировал свой код на локальной машине, обращаясь к «стабильному» API бэкенда. Это вполне разумный способ избежать проблем с изменениями в API; к тому же фронтенд-разработчик не знает, как управлять API бэкенда и тестовыми данными. Однако если бы он перед отправкой кода запустил последнюю версию API бэкенда, то легко обнаружил бы проблему.

Команда решает научить разработчиков фронтенда, администраторов баз данных и DevOps-инженеров управлять *всей* средой бэкенда на локальных машинах, в том числе управлять обновлениями и проверять их (получать и интегрировать изменения кода от других членов команды из текущей основной ветки, что похоже на команду `git pull`). Разработчики жалуются, что локальной средой сложно управлять: по их словам, отсутствие эталонных данных и неработающая локальная база данных создают большие неудобства. И хотя команда фронтенда находит нетривиальные способы сгладить это недовольство, жалобы на ошибки в бэкенде и на стабильность API не прекращаются.

Ошибка A2

Выясняется, что у команды бэкенда нет автоматизированных тестов для проверки контрактов API. Их обычный цикл разработки построен на основе методологии TDD (test-driven development, разработка через тестирование): они пишут модульные и интеграционные тесты, компилируют

их, запускают и проверяют работу. После этого они уверенно отправляют код в репозиторий. Разработчики бэкенда отмечают, что на данном этапе проекта нет ни времени, ни возможности добавить тесты для контрактов API. Они утверждают, что локальное тестирование фронтенд- и бэкенд-компонентов упростит обнаружение ошибок, так как используемые для этого ручные сценарии немногочисленны и просты в исполнении.

Они просят команду фронтенда показать, как настроить их рабочую среду на локальной машине. Они обещают выполнять приватные сборки с использованием общих ручных тест-кейсов, чтобы обеспечить стабильность API — по крайней мере до тех пор, пока не появится возможность дальнейшей автоматизации.

Ошибка А3

К этому времени разработчики фронтенда и бэкенда уже выработали методику ручного тестирования приложения в локальной среде при каждом обновлении, используя новую версию тестовых данных. Однако вскоре они начинают жаловаться на нестабильность тестовых данных и сообщать о частых, запутанных и на первый взгляд случайных ошибках. Им приходится тратить время на сложную отладку, чтобы в итоге прийти к выводу: причина в новых данных, которые содержат ошибки.

Они просят администраторов БД настроить локальную среду и выполнять в ней набор ручных тестов перед отправкой изменений в репозиторий — по сути, делать ручную приватную сборку. Администраторы БД прогоняют тесты при каждом изменении данных: до и после получения изменений из репозитория. Обнаружив несовместимость данных, администраторы БД и разработчики бэкенда связываются друг с другом напрямую. Разработчики бэкенда пишут интеграционный тест, который выявляет несовместимость, а затем устраняют ее. Когда они отправляют данные и исправления в репозиторий, основная ветка остается стабильной.

Ошибка А4

После внедрения ручных приватных сборок *все* члены команды начинают искать ошибки интеграции. QA-инженер занимается функциональной проверкой и управлением версиями приложения. Теперь развертывания проходят гораздо стабильнее, а исправлений требуется значительно меньше. Тем не менее тестирование исправлений по-прежнему занимает много времени, поскольку процессы автоматизации еще не отлажены. QA-инженер просит DevOps-инженеров сократить время развертывания. Одновременно он просит команду предоставить ему локальную среду разработки, чтобы он мог про-

должать функциональную проверку даже при отсутствии финальной развернутой версии. В локальной среде ему также проще разобраться в возможных проблемах, и он может точнее сообщать об ошибках. Контроль над локальной средой дает QA-инженеру больше возможностей действовать проактивно, и теперь он может начать писать автоматизированные сквозные тесты.

Команда обновляет таблицу анализа ошибок (табл. 5.2), отражая в ней полученные результаты.

Таблица 5.2. Скорректированный анализ ошибок

Ошибка	Компонент	Предотвращается приватной сборкой?	Легко ли автоматизировать?	Чем вызвано?
A1	Фронтенд	Да	В процессе	Непроверенное неопределенное значение, полученное от сервера
A2	Бэкенд	Да	Да	Несоответствие типа данных в API
A3	Интеграция с реальными данными	Да	Да	Формат реальных данных как пограничный случай
A4	Интеграция	Да	Нет	Проблема с обратной совместимостью

Как видно, в системе периодически возникают ошибки. Их количество можно сократить, выполняя приватные сборки перед слиянием с основной веткой. Это гарантирует, что добавляемый код не нарушит ее стабильность. По моему опыту, в любой команде эффективнее выполнять собственную сборку, чтобы убедиться, что как минимум ваши функции работают и не нарушают работу всей системы.

Кейс: замкнутый круг в консалтинговой компании

Консалтинговая компания сотрудничает с Компанией B — крупным, давно работающим на рынке разработчиком ПО, который внедряет новые методологии и стремится к автоматизации. Внутренняя DevOps-команда Компании B занимается автоматизацией довольно сложного пайплайна доставки, который все еще находится в разработке.

Разработчики из консалтинговой компании (далее — «консультанты») начинают поставлять функционал, но средства автоматизации, которые они

используют для развертывания тестовой среды, работают нестабильно. Часто после отправки кода итоговое развертывание в тестовой среде не выполняется. К тому же у них нет инструментов для отладки возможных проблем интеграции.

В ходе обсуждения с командой Компании B консультанты выявляют несколько проблем, например:

- изменились тестовые данные;
- механизм выполнения миграций базы данных оказался временно отключен;
- контейнер, в котором работает бэкенд, постоянно перезапускается, возможно, из-за системной ошибки;
- один из компонентов, от которого зависят другие, оказался временно отключен из-за миграции в другую облачную систему.

На первой итерации разработки консультанты не могут выпустить почти ни одного функционального инкремента, поскольку анализ и решение упомянутых проблем сильно замедляют работу. На второй итерации они подают официальный запрос на предоставление выделенной тестовой среды.

Как это часто бывает, запрос принимают, но его реализация не вписывается в график работы выделенной DevOps-команды. По ее оценкам, на создание новой среды уйдет две недели; в качестве причины задержки они называют внутреннюю бюрократию. Тем временем консультанты не понимают, как разрешить сложившуюся тупиковую ситуацию. Они начинают поставлять функционал, полагаясь только на модульное и интеграционное тестирование.

После того как консультанты поставляют первую итерацию функционала, внутренние стейкхолдеры Компании B жалуются на его реализацию, в частности на то, что не учтены пограничные случаи. Также обнаруживаются несколько критических ошибок, которые влияют на интеграцию с другими сервисами. Команда разработчиков (состоящая из консультантов и специалистов Компании B) тем временем начала проводить тестирование на уровне компонентов и контракта API, чтобы исключить хотя бы эти проблемы, но внутренняя DevOps-команда слишком занята, чтобы встроить эти тесты в свой пайплайн.

Во время второй итерации API-тесты уже написаны, но DevOps-команда так и не встроила их в пайплайн. Процесс остается по-прежнему неэффективным с точки зрения скорости и качества доставки, ничего не меняется; не гарантируется даже доставка исправлений. Тогда команда разработчиков

решает провести комплексный анализ, чтобы выявить истинные проблемы и изучить возможные решения.

Поскольку ситуация в Компании B вряд ли быстро изменится, команде требуется временное решение. Они приходят к выводу, что API-тесты никто не запускает и поэтому за ними никто не следит. Тогда эта команда решает начать *вручную* выполнять все API-тесты *до и после* каждого обновления на локальных машинах.

Такое решение позволяет не зависеть от никем не контролируемой автоматизации и возвращает к локальной среде — приватной сборке. Опять же, они сами себе обещают, что скоро автоматизируют запуск тестов, но по факту полагаются на сам *процесс автоматизации*. Между тем ошибки интеграции и функциональные пограничные случаи в область тестирования в рамках этого процесса не попадают.

И вот, убедившись, что приватная сборка — это лучший вариант, команда решает применить тот же принцип и к остальным проверкам. Проблема, однако, в том, что приложение зависит от внешних сервисов. Тогда принимается решение создать потенциально ненадежные *псевдореплики* внешних сервисов, чтобы, используя их сбои, обеспечить надежное тестирование приложения.

На третьей итерации команда разработчиков проверяет все свои изменения локально, независимо от средств внутренней автоматизации Компании B. Процесс явно ускоряется, исправленные версии компонентов работают эффективнее, и разработчики могут заниматься автоматизацией проверок, а не поиском выхода из тупика.

В итоге эта команда добилась высокой скорости и надежности доставки благодаря тому, что стала проводить проверки в локальной среде разработки.

Метрики

В обоих рассмотренных примерах команды реагировали на проблемы, расширяя возможности локальных сред. Они шли на многие компромиссы, например, занимались ручным тестированием, хотя могли бы заявить, что это «не входит в круг их обязанностей». Они оценивали ситуацию *неявно* — на основе метрик.

Метрики могут быть качественными или количественными. Кроме того, по способу достижения цели они могут быть прямыми или косвенными. В этом разделе я предлагаю рассмотреть несколько метрик для оценки процесса разработки.

Время до получения обратной связи

Группа: качественные метрики
Тип: косвенная
Измеряет: стоимость и время вывода на рынок

`Time to Feedback` (время до получения обратной связи) — это метрика, которая измеряет время и усилия, требуемые для получения отклика на реализацию нового функционала. Эта метрика считается качественной главным образом потому, что ее количественная оценка зависит от конкретной ситуации. На самом деле, в одних случаях обратная связь занимает слишком много времени (из-за медленной автоматизации), а в других — проблема в том, чтобы найти нужного человека для получения официальной обратной связи.

Существуют разные виды функциональной обратной связи. Обратная связь может касаться проблем интеграции или чисто функциональных проверок. Ее можно получать от стейкхолдеров, потребителей API и конечных пользователей. Как правило, команде разработчиков следует в первую очередь обрабатывать ту обратную связь (например, по интеграции и QA), которая блокирует ее собственную работу.

`Time to Feedback` — косвенная метрика. Она может предупредить о проблеме, но не помогает глубоко разобраться в ее первопричине. Длительный цикл функциональной обратной связи означает, что в процессе есть сбой, но не указывает, в чем именно проблема.

В первом примере (Компания A) QA-инженер, который отслеживал обратную связь по интеграции, был перегружен задачами из-за множества проблем, которых можно было избежать. В тот момент команда не могла оперативно получать обратную связь о своей работе, что привело к нестабильности основной ветки. В примере Компании B цикл обратной связи был полностью делегирован, что мешало повседневной работе.

Количество предотвратимых проблем интеграции в развернутом приложении за итерацию

Группа: количественные метрики
Тип: косвенная
Измеряет: внутренний процесс контроля качества

Эта метрика подсчитывает количество предотвратимых (устранимых) проблем, которые обнаруживаются в каждой итерации развернутого ПО.

К ним относятся те проблемы, которые можно было легко выявить во время приватной сборки с помощью автоматической или ручной проверки. Низкое или снижающееся значение указывает на более зрелый процесс, позволяющий успешнее находить интеграционные ошибки до развертывания приложения.

Более классическая метрика — это `number of bugs found in QA` (количество ошибок, выявленных при QA), но в данном случае разница между ними заключается в критерии классификации: проблема должна быть легко предотвратимой на предыдущем этапе проверки. В целом приватную сборку не следует считать идеальной и абсолютно полной проверкой.

Хорошим правилом считается выполнять проверки на основе тестов контракта API, а также happy-path тестов (то есть по позитивному сценарию) для глобально важных (например, для входа в систему) или затрагиваемых изменениями функциональных областей. Метрика является косвенной, поскольку требует анализа того, затрагивает ли проблема отдельную часть команды или всю команду в целом.

Время на восстановление стабильности основной ветки за итерацию

Группа: количественные метрики

Тип: прямая

Измеряет: стабильность кода и способность команды ее поддерживать

Под *стабильностью основной ветки* здесь понимается стабильное (или приемлемо рабочее) состояние основной ветки кода. Когда основная ветка нестабильна, обновление (git pull) часто нарушает работу какой-либо функции приложения в локальной среде.

Эта метрика измеряет время отладки, потраченное на устранение ошибок, которые были внесены в основную ветку и обнаружены при запуске приватной сборки после обновления на локальной машине. Она также включает время на устранение ошибок, которые нарушают работу существующих функций. Обеспечить стабильность основной ветки можно, предотвращая попадание в нее дефектов, а восстановить — постоянно их исправляя. Эта метрика измеряет усилия, необходимые для *восстановления* стабильности.

Это *прямая* метрика, поскольку она измеряет время (а значит, и затраты), ушедшее на действия, не связанные с разработкой новых функций. В отличие от других метрик, например *общего времени отладки* или *количества ошибок*, она измеряет именно эффективность устранения регрессионных ошибок в локальной приватной сборке. В отличие от метрик, которые ка-

саются регрессионных ошибок в целом, она учитывает время, затраченное на стабилизацию основной ветки в процессе разработки.

Результаты этой метрики могут указывать на недостаточную дисциплину выполнения локальных приватных сборок. Такое случается, если локальная среда разработки не стандартизирована или разработчики не до конца ее понимают. Результаты также могут указывать на недостаточную автоматизацию тестирования, что в описываемых обстоятельствах вполне ожидаемо.

Наконец, стабильность основной ветки напрямую и в огромной степени влияет на объем работы, необходимой для получения стабильных версий и релизов. В конечном счете стабильность основной ветки — это один из факторов, который делает возможным непрерывное развертывание ПО.

Затраты на приватные сборки

Главное возражение разработчиков против того, чтобы вернуть валидацию в локальную среду, заключается в том, что у них снижается производительность и частота коммитов, поскольку им приходится тратить больше времени на поддержку локальной среды.

Это веский довод. Однако я считаю, что усилия, которые уходят на выполнение приватных сборок, *все равно* несоизмеримо меньше затрат на постоянную стабилизацию основной ветки, итерации с QA-инженерами и управление тикетами. Как убедительно доказывают многочисленные источники, чем больше времени проходит с момента появления дефекта до его обнаружения, тем дороже в экспоненциальной прогрессии обходится его исправление. Кроме того, ручные интеграционные проверки должны быть *временной мерой*. Следует как можно скорее автоматизировать все действия, которые команда вынуждена повторять в ходе приватной сборки. (Временный) этап ручного тестирования должен быть похож на простое и целенаправленное «дымовое тестирование» и оставаться минимально затратным.

Метрики на практике

В заключительном разделе этой главы приведу несколько примеров интерпретации метрик из предыдущего раздела.

Длительный срок получения обратной связи, много предотвратимых проблем интеграции, короткий срок стабилизации основной ветки

Это наиболее распространенное сочетание. Метрика `Time to Feedback` (время до получения обратной связи) в связке с метрикой `Evitable Integration`

Issues (количество предотвратимых проблем интеграции) часто указывает на возможную первопричину. На самом деле, высокое значение Evitable Integration Issues подтверждает гипотезу о том, что жизненный цикл обратной связи затягивается из-за переноса большинства проверок на средства автоматизации и QA. Низкое значение Time for Trunk Stability (время до стабилизации основной ветки) коррелирует с высоким значением Evitable Integration Issues, что, вероятно, указывает на попадание в основную ветку ошибок, не обнаруженных в локальной среде.

Чтобы подтвердить или опровергнуть эту гипотезу, можно проделать следующее:

- проанализировать «предотвратимость» заявленной проблемы интеграции, например на ретроспективах;
- убедиться, что все члены команды могут запускать полные тесты в локальной среде;
- организовать в команде обмен наборами минимальных функциональных тестов, которые нужно запускать (вручную или автоматически) перед коммитом;
- потребовать от команды дисциплины в выполнении приватных сборок;
- работать итеративно.

Короткий срок получения обратной связи, много предотвратимых проблем интеграции, короткий срок стабилизации основной ветки

Time to Feedback — качественная метрика, поэтому она может быть субъективной, и высокое значение Evitable Integration Issues эту гипотезу может подтвердить. В таком случае мы имеем сценарий, аналогичный предыдущему.

Такая ситуация может также возникнуть, если QA-инженеры очень эффективно справляются с большой нагрузкой, компенсируя неэффективность локальных проверок, которые выполняет остальная часть команды. В этом случае следует потребовать большей дисциплины от остальных членов команды.

Длительный срок получения обратной связи, мало предотвратимых проблем интеграции, короткий срок стабилизации основной ветки

Поскольку Time to Feedback — косвенная метрика, ее требуется анализировать дополнительно. Низкие значения двух других метрик могут указывать на два совершенно противоположных сценария.

Первый — неудовлетворительный общий процесс QA, при котором ошибки обнаруживаются на очень поздних этапах доставки ПО. Причиной может быть неэффективный процесс обнаружения в целом, медленная автоматизация (или отсутствие таковой) или и то и другое вместе. Чтобы опровергнуть эту гипотезу, необходимо проанализировать другие метрики, например удовлетворенность клиентов и количество ошибок, которые появляются в каждой итерации.

Второй возможный сценарий: стабильность основной ветки высока и процессы налажены, но бюрократия, загруженность команды DevOps и перегруженность QA-инженеров приводят к задержкам в получении обратной связи от средств автоматизации и QA. В этом случае следует оперативно устранить барьеры между командами разработки, DevOps и QA, поскольку команда разработчиков и так эффективно поддерживает стабильность основной ветки.

Мало предотвратимых проблем интеграции и длительный срок стабилизации основной ветки

Такая ситуация часто возникает, когда в основную ветку попадает много ошибок, которые не проявляются в среде интеграции и QA. Возможно, одни члены команды тратят много времени на исправление ошибок, а другие — недисциплинированны и неохотно запускают приватные сборки. Если вы заметите такой значительный дисбаланс в усилиях по поддержанию стабильности основной ветки, проведите внутренние ретроспективы процессов.

У вас может возникнуть вопрос, почему в этой комбинации нет метрики `Time to Feedback`. Если значение `Time to Feedback` высокое, проанализируйте ситуацию с точки зрения неэффективности процесса доставки так же, как вы это делали в предыдущем пункте.

Заключение

Мир несовершенен, и компании-разработчики ПО не всегда применяют все передовые практики. Причины тому могут быть разными: непонимание того, как эти практики внедрять, незавершенный процесс их внедрения или вообще другие приоритеты организации.

Эта глава посвящена разбору обстоятельств, при которых культура DevOps, будучи передовой практикой, слабо проявляется в команде разработчиков. Подобные условия негативно сказываются на производительности команды и в конечном счете на обеспечении качества. В таких случаях важно

компенсировать неэффективность и риски, которые вызваны разделением команд автоматизации и разработки.

Многие команды терпят неудачу при внедрении новых методологий или передовых практик, потому что пытаются изменить вещи, с принципом работы которых они еще сами не разобрались, — а без такого понимания не получится вносить улучшения и добиваться результатов.

Цель изучения метрик в этой главе — *выявить* проблему смещения ответственности и *точечно* воздействовать на слабые места процесса. Если программная архитектура обычно касается «статических» свойств системы или ее свойств во время выполнения ПО, то метрики из этой главы охватывают *процесс* создания продукта (в противовес процессам проектирования или измерения). Если мы будем оценивать архитектурные свойства часто ломающейся системы, наши выводы, скорее всего, будут неверны. Именно поэтому контроль стабильности основной ветки должен быть приоритетом программной разработки.

Приватные сборки — важнейший инструмент, который помогает «пережить» внедрение более совершенного способа интегрировать команду и сохранить приемлемую эффективность и качество. Приватные сборки сводят к минимуму появление в системе контроля версий ошибок и дефектов, которые обнаруживались бы на поздних стадиях, что влекло бы за собой снижение эффективности, рост затрат и рисков. Это особенно актуально для ошибок, связанных с интеграцией различных программных компонентов.

Подготовка приватных сборок сама по себе — передовая и знакомая многим практика, но она становится критически важна, когда процессы автоматизации и валидации оторваны от команды разработчиков. Если расширять возможности локальной среды, то ответственность за доставку работоспособного ПО снова вернется к команде разработчиков. Таким образом, локальный запуск приватных сборок и повышенное внимание к метрикам позволяют ускорить цикл доставки, удешевить процесс разработки и повысить качество ПО даже в неблагоприятных условиях.

ГЛАВА 6

Масштабирование организации: центральная роль программной архитектуры

Жоао Роза

Я изучаю социотехнические системы и теорию сложности. Хотя в душе я инженер-разработчик, мне нравятся сложные задачи, которые возникают на стыке взаимодействия людей, технологий и базовых процессов, то есть в *социотехнических системах*. Каждый день меня вдохновляет возможность улучшать системы, частью которых я являюсь. Я считаю, что технические директора (CTO, chier technical officer) и директора по продуктам и технологиям (CPTO, chief product and technology officer) должны понимать идею социотехнических систем и вносить в них свой вклад. Это поможет окружающим их людям и командам достигать совершенства в своих областях знаний.

Я сформировал свои подходы к программной архитектуре, побывав в роли инженера-разработчика, менеджера, архитектора ПО, консультанта и технического директора. Моя специализация — цифровые компании, которые меняют жизнь людей к лучшему. В основном я работаю со скейл-апами — организациями, которые уже доказали востребованность своих продуктов на рынке и теперь стремятся выйти на новые рынки или вывести новые продукты. Я пишу эту главу с позиции технического директора, рассматривая вопросы с точки зрения реализации стратегии. Такой подход позволяет мне связать концепции программной архитектуры и метрик — способов измерения прогресса в достижении целей — с остальными частями организации, которые они затрагивают. Подобная целостность обеспечивает слаженную работу всей организации, в которой сотрудники понимают принимаемые решения и в своей работе опираются на постоянно меняющуюся программную архитектуру (спойлер: да-да, архитектура не статична).

Надеюсь, эта глава послужит для вас источником вдохновения: в ней описана вымышленная история, в которой собирательный персонаж по имени Анна связывает программную архитектуру с метриками. Я не ставлю цель дать вам готовый рецепт, поскольку каждый контекст уникален. Я скорее предлагаю подход, который поможет сформулировать метрики для своих собственных архитектурных решений.

> Представьте себе Анну, сеньор-разработчика. Недавно она устроилась в отдел разработки продуктов вымышленного финтех-скейлапа YourFinFreedom. Ее отдел стремится ускорить работу и повысить качество поставляемого программного продукта. По своим ощущениям, Анна хорошо вписалась в коллектив, и здесь ценят ее профессиональные навыки.
>
> Цель YourFinFreedom — использовать недавнее законодательство Европейского союза об открытом банкинге (https://oreil.ly/0FJKr), известное как Вторая директива об оказании платежных услуг (Second Payment Services Directive, PSD2), для создания сервиса, который позволит людям пользоваться финансовыми услугами по самым выгодным тарифам. Компания находится в Бельгии, ее клиенты — в Бельгии, Нидерландах и Люксембурге. Компания планирует выйти на рынок остальных стран Евросоюза, поэтому ее стратегия — предложить свой сервис в крупнейших странах Европы: Франции, Германии и Италии. Сейчас сервис YourFinFreedom работает нестабильно и иногда бывает недоступен, что вызывает жалобы конечных пользователей. Эти проблемы — следствие быстрого развития компании.
>
> Это означает, что сервис должен стать более отказоустойчивым, взаимодействовать с бóльшим числом поставщиков финансовых услуг и масштабироваться для удовлетворения прогнозируемого спроса. Одновременно с этим необходимо повышать функциональность продукта и скорость доставки ценности пользователям.

Здесь действуют несколько бизнес-факторов, и все они влияют на архитектуру ПО компании YourFinFreedom. Конечно, это лишь упрощенная история. Каждый контекст уникален, и в нем всегда действуют разные силы (см. рис. 6.1).

За время своей профессиональной деятельности я заметил, что программная архитектура обычно отражает два аспекта: организационную структуру компании и способы коммуникации внутри этой структуры. Первый аспект очевиден, обычно он отражается в организационной схеме компании. Второй не очевиден, и люди часто не осознают его влияния на свою работу, особенно на архитектуру ПО. Если программная архитектура не создается целенаправленно для решения конкретных задач, она начинает повторять организационную и коммуникационную структуру компании (то есть ее команд и отделов). Это явление известно как закон Конвея, по имени его исследователя Мела Конвея. В своей статье 1968 года «How Do

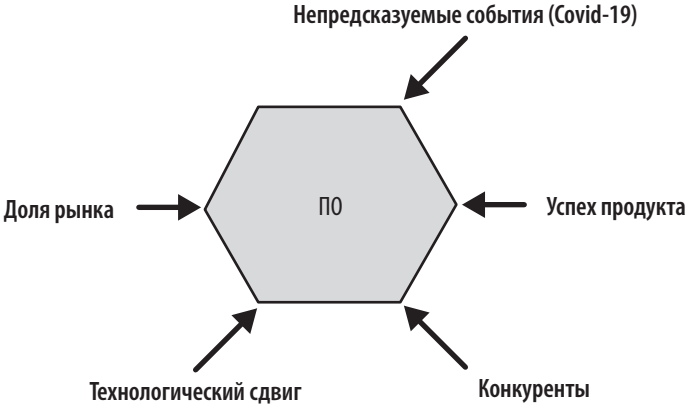


Рис. 6.1. Примеры бизнес-факторов, влияющих на архитектуру ПО

Committees Invent?» М. Конвей пишет: «Организации, проектирующие системы... вынуждены создавать проекты, структура которых копирует структуру коммуникаций в этих организациях»[1].

YourFinFreedom: отказ от монолита

Кейша, руководитель отдела продуктовой разработки, приходит к решению, что лучший способ ускорить работу и повысить качество продуктов — перейти на микросервисную архитектуру. Существующая архитектура представляет собой монолит, который годами обрастал различными функциями и интеграциями с поставщиками финансовых услуг. По мере перехода отдела на новый архитектурный стиль количество команд постепенно вырастет с 3 до 10, и каждая из них будет работать в своей зоне ответственности. Кейша уверена, что такой подход обеспечит масштабируемость, необходимую для выхода на крупные новые рынки Европы.

Проработав несколько месяцев над переходом на новую архитектуру, Анна понимает, что микросервисы не отвечают бизнес-потребностям компании и из-за этого перехода компания может упустить некоторые возможности. Анна обсуждает этот вопрос со своим руководителем Кейшей, но та твердо намерена придерживаться плана и убеждает Анну продолжать работу.

Рассмотрим два распространенных сценария, которые я наблюдал во многих быстрорастущих компаниях (рис. 6.2): сначала — декомпозицию монолита, а затем — осмысление паутины микросервисов.

---

[1] Мелвин Конвей (Melvin E. Conway). «How Do Committees Invent?» Datamation (апрель 1968): 31, https://oreil.ly/qFpIM.

Монолитная и микросервисная архитектуры — это разные архитектурные стили, и командам приходится искать компромисс между централизованной (монолит) и распределенной (микросервисы) архитектурой. Такой поиск компромиссов сказывается на архитектуре, поскольку определяет, как команды агрегируют жизненно важную для продукта или сервиса бизнес-логику.

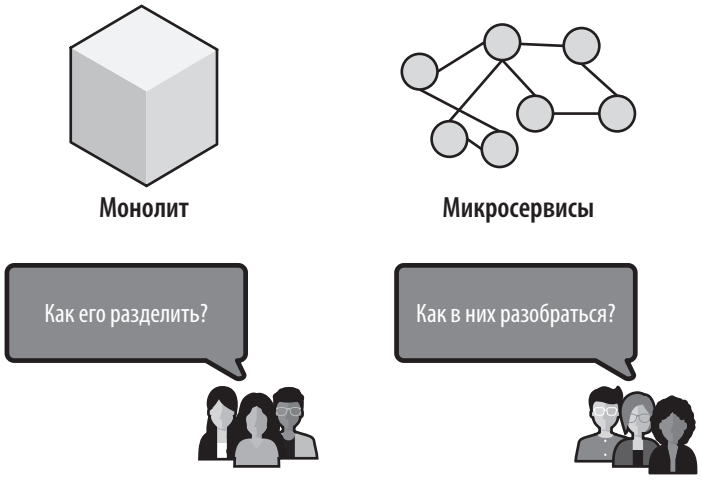


Рис. 6.2. Типичные проблемы монолитов и микросервисов

Любая из этих архитектур может быть вполне успешной при правильной реализации. Однако под влиянием различных факторов, начиная с требований бизнеса и заканчивая раскрученными технологиями, монолит часто вырождается в то, что архитекторы называют *большим комом грязи* (big ball of mud), а микросервисы — *большим распределенным комом грязи* (distributed big ball of mud), см. рис. 6.3.

Термин «большой ком грязи» ввели в 1999 году Брайан Фут (Brian Foote) и Джозеф Йодер (Joseph Yoder); они определяют его таким образом: «это как попало разработанная масса беспорядочного, сырого, наспех перемотанного изолентой и проволокой спагетти-кода»[1]. На другом полюсе находится «большой распределенный комок грязи», который обладает теми же характеристиками, но с одним дополнением: он распределен по сети, что еще больше усложняет ситуацию. Покажу пару примеров и их последствия.

[1] Фут Б., Йодер Д. «Big Ball of Mud», The Laputan Press, 26 июня 1999 г.

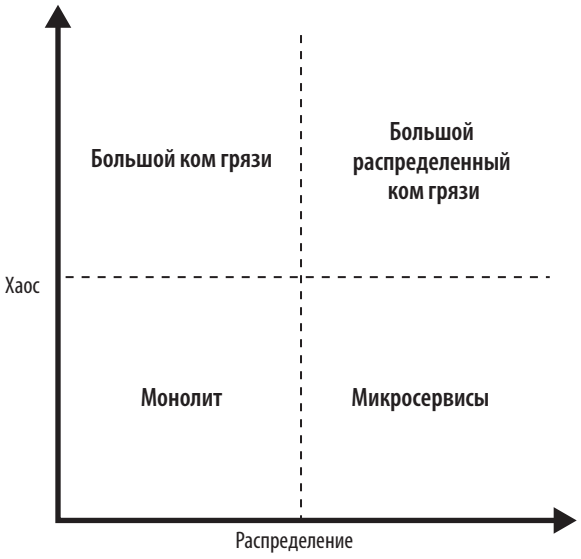


Рис. 6.3. Эта двухмерная диаграмма наглядно представляет два архитектурных стиля и их хаотичные аналоги

Когда в организации используется монолитная архитектура и возникает желание быстрее выводить на рынок новые функции, для укомплектования команд разработки нанимают больше специалистов. Эти специалисты и команды начинают пробовать разные решения поверх существующей централизованной архитектуры, добавляя в систему *непреднамеренную сложность* и формируя «большой ком грязи». Непреднамеренная сложность — это сложность, которая появляется в системе в виде новой зависимости, плохо документированного или непротестированного кода, неструктурированного дизайна и т. д. Обычно это происходит, когда команда разработки с трудом справляется с разнообразными бизнес-требованиями. Непреднамеренная сложность вызывает цепную реакцию, которая сказывается на простоте обслуживания, эксплуатации и внесении изменений в ПО; а в итоге, как отметил Фредерик Брукс (Frederick Brooks)[1], мешает организации быстро выводить на рынок новые функции.

На другом полюсе, где находится «большой распределенный ком грязи», специалисты и команды пытаются разобраться в паутине микросервисов. Такие системы обычно настолько сложно спроектированы для целевых

[1] Брукс Ф. «Мифический человеко-месяц, или Как создаются программные системы». СПб., издательство «Питер».

бизнес-задач, что *когнитивная нагрузка*, требуемая для понимания бизнес-транзакции, превышает возможности человеческого мозга. Мэтью Скелтон (Matthew Skelton) и Мануэль Паис (Manuel Pais) определяют допустимую когнитивную нагрузку человека как «ограничение на объем информации, который он может удерживать в уме в любой момент времени. Определение также справедливо и в отношении любой отдельной команды — ее допустимая когнитивная нагрузка равна сумме когнитивных способностей всех ее участников»[1].

Реализация «большого распределенного кома грязи»

Год спустя в YourFinFreedom появилось множество микросервисов, а также сохранились некоторые остатки монолита. Всеми ими управляют 10 команд отдела продуктовой разработки. Однако новая архитектура не оправдала надежд на ускорение работы и повышение качества продуктов для пользователей. Команды не реализуют свой потенциал, а с флагманским продуктом постоянно возникают проблемы.

Анна решает обсудить с Кейшей и остальными руководителями команд продуктовой разработки ежедневные трудности, с которыми сталкиваются подчиненные. Приведем некоторые из них:

- Чтобы вывести функцию в конкретную область продакшена, командам приходится заниматься оркестрацией релизов между микросервисами.
- Функциональность разных микросервисов пересекается, из-за чего командам сложнее их контролировать.
- Продуктивность команд ниже потенциально возможной. В этом контексте Анна отмечает, что у отдела постоянно меняются приоритеты, из-за чего большая часть работы оказывается заброшенной.
- Из-за постоянных проблем с продуктом у многих сотрудников опустились руки и снизилась мотивация.
- Они просто не понимают, в каком направлении развивается продукт и какой вклад в его развитие они могут внести.

Анна предлагает возможное решение: внедрить метрики и прочертить границы архитектуры, чтобы задать командам продуктовой разработки нужное направление и одновременно позволить архитектуре развиваться в соответствии с планами компании по расширению.

Руководители остались довольны тем, что Анна напрямую подняла сложные вопросы и предложила решение. Все они уже почувствовали, что текущий переход на микросервисы не оправдывает возложенных на него ожиданий, и теперь

[1] Скелтон М., Паис М. «Team Topologies» (IT Revolution Press, 2019), с. 11.

понимают, что прежде чем начать, не взвесили должным образом все за и против. Кейша решает проверить Анну в деле: поскольку у Анны выявились такие прекрасные лидерские и инженерные качества, Кейша рассматривает возможность повышения Анны до архитектора решений (Solutions Architect) — новой должности в отделе продуктовой разработки. На этой должности Анна должна будет направлять работу команд над программной архитектурой и следить за тем, чтобы весь технический ландшафт компании был внутренне согласован. Кейша предлагает Анне подумать об этом повышении.

Через несколько дней Анна принимает предложение Кейши, но с одним условием: она хочет получить полный карт-бланш на взаимодействие со всеми подразделениями компании, чтобы понять их проблемы. Кейша соглашается, и Анна чувствует, что ее видение ситуации было принято. В ее карьере открывается новая страница!

Отдел Анны страдает от проблем, которые свойственны «большому распределенному кому грязи». Трудности, описанные в ее истории, отражают непреднамеренную сложность архитектуры.

Компания YourFinFreedom намеревалась перейти от монолитной архитектуры к микросервисной, как показано линией 1 на рис. 6.4. Однако из-за отсутствия четкого, основанного на метриках направления компания пошла по пути, который обозначен линией 2 и который привел ее к «большому распределенному кому грязи». Как же Анне сориентировать компанию в нужном направлении?

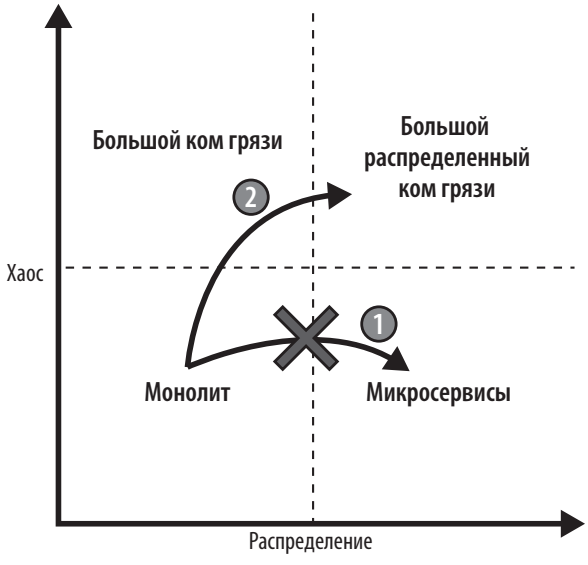


Рис. 6.4. Непреднамеренная сложность при переходе от монолита к микросервисам

В поисках направления

В компании объявляют, что Анна отныне занимает должность архитектора решений, и она приступает к работе. Чтобы лучше понять направление развития компании, она начинает общаться с сотрудниками из других отделов, которые выполняют самые разные бизнес-функции. Анна хочет понять повседневные проблемы коллег и их связь с общей картиной разработки и сопровождения ПО в ее отделе.

В ходе одного из таких разговоров она узнает, что компания YourFinFreedom недавно скорректировала свои KPI, чтобы они лучше соответствовали ее миссии: «Сделать финансовые услуги доступными всем и везде». Из корпоративных новостей Анна также знает, что высшее руководство определяет направление деятельности на базе EBITDA (earnings before interest, taxes, depreciation, and amortization — прибыль до вычета процентов, налогов и амортизации) и CLV (customer lifecycle value — пожизненная ценность клиента). Анна проявляет интерес к новому KPI организации — MAU (monthly active users — числу активных пользователей в месяц), который показывает, сколько пользователей взаимодействуют с услугами или продуктами компании за месяц. Этот KPI формирует цикл обратной связи для всех инициатив компании (отделов маркетинга, продаж, программной разработки), направленных на вовлечение клиентов. Анна задается вопросом: как связать этот KPI с архитектурой ПО?

Программы создают, чтобы решать сложные проблемы в определенном контексте. Как мы только что увидели, Анна выяснила, что в YourFinFreedom наряду с финансовыми показателями на организационном уровне используется еще один важный KPI (рис. 6.5).

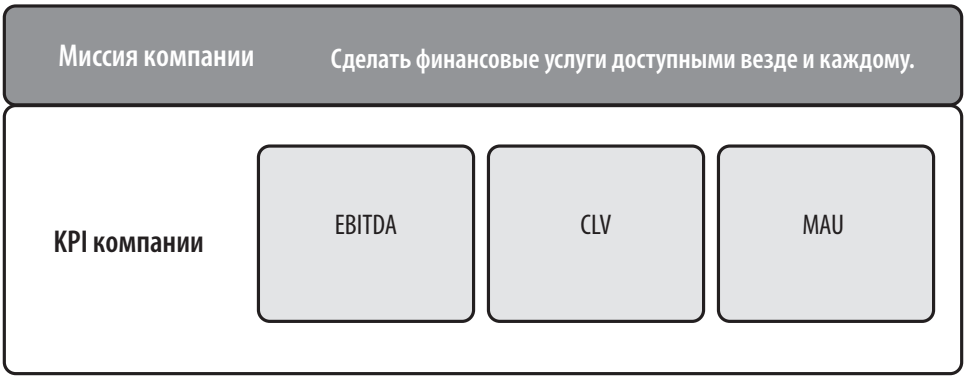


Рис. 6.5. Миссия и KPI компании YourFinFreedom

В реальности же разные варианты MAU широко используются компаниями-разработчиками соцсетей и компаниями, которые ориентируются преи-

мущественно на пользователей с мобильными устройствами (mobile-first). Компаниям-скейлапам MAU позволяет управлять возможностями роста через инструменты отделов маркетинга, продаж, поддержки клиентов либо через функции продукта или сервиса.

Программная архитектура в организации должна быть связана с миссией и KPI компании. Если KPI определены нечетко или сотрудники их не понимают, архитектура будет идти вразрез с этими показателями, и это послужит ярким примером закона Конвея, который и показывает, как структура коммуникаций влияет на программную архитектуру. Программная архитектура нужна, чтобы определять как границы ответственности между компонентами системы, так и способы взаимодействия этих компонентов. Она должна, с одной стороны, учитывать сложившиеся внутренние и внешние ограничения организации (технологии, сотрудники, нормативные требования, рынок и т. д.), а с другой — обеспечивать гибкость для бизнеса.

От максимальных усилий к целенаправленным действиям

Размышляя над запросами стейкхолдеров и над текущим состоянием архитектуры продукта, Анна замечает одну особенность. С одной стороны, все прикладывают максимальные усилия, стараясь выполнить все запросы на разработку. С другой — ей становится очевидно, что команды не учитывают одно обстоятельство, важное для качества программной архитектуры: их решения должны быть целенаправленными.

Анна обращается к различным сообществам разработчиков, чтобы посмотреть, как другие решают подобные проблемы. Там она узнает об EventStorming — «гибком формате воркшопов для совместного исследования сложных направлений бизнеса»[1], который широко используется в сообществе предметно-ориентированного проектирования (Domain-Driven Design) для визуализации процессов, а также для обсуждения границ и метрик. Она выясняет, что в формате EventStorming сначала рисуют общую картину по разным зонам ответственности, а затем привлекают профильных экспертов из разных областей, чтобы они поделились принципами своей работы. Анна решает испытать этот метод в деле и учится проводить подобные воркшопы.

Она собирает на воркшоп Big Picture EventStorming[2] группу специалистов из разных отделов компании, и эта группа составляет карту текущего бизнес-процесса.

---

[1] «EventStorming», дата обращения: 11 мая 2022 г., https://www.eventstorming.com.

[2] Подробное описание процесса EventStorming, которое выходит за рамки этой главы, см. в статье Альберто Брандолини (Alberto Brandolini), «Discovering Bounded Contexts with EventStorming», в книге «Domain-Driven Design: The

Специалисты фокусируются на потоке и последовательности бизнес-событий, по ходу работы определяя границы, на которых либо меняется тип работы, либо заканчивается процесс. В результате группа визуализирует этот поток, а также выявленные границы предметных областей.

Результат воркшопа получился похожим на тот, что приведен на рис. 6.6. Группа отражает ключевые события (обозначены маленькими квадратами), происходящие между выявленными предметными областями. Понимание цели создания ПО (и его архитектуры) имеет первостепенное значение, поэтому первым делом Анна просит группу определить бизнес-логику для каждой области.

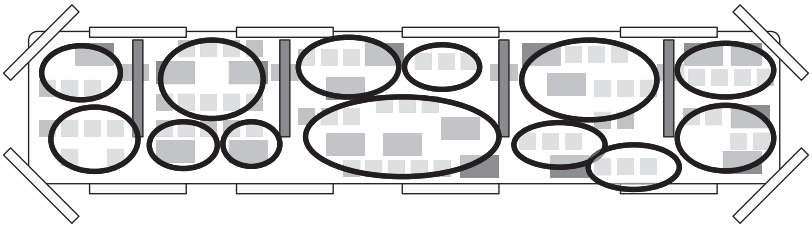


Рис. 6.6. Результат воркшопа Big Picture EventStorming с выявленными предметными областями[1]

С помощью участвующих в воркшопе инженеров-разработчиков группа наносит текущие программные компоненты на полученную схему Big Picture EventStorming (рис. 6.7). В ходе этой работы становится очевидно, насколько замысел бизнес-процесса не соответствует его реализации. Именно это несоответствие и объясняет, почему архитектура превратилась в «большой распределенный ком грязи». Группа также определяет владельцев компонентов. В ходе обсуждений Анна видит подтверждение своим догадкам о том, откуда именно идет высокая когнитивная нагрузка на команды.

Добавление текущих программных компонентов (микросервисов, монолитов и т. д.) на схему с разметкой по выявленным предметным областям позволяет группе оценить текущее положение дел и понять, почему компании YourFinFreedom так сложно развивать архитектуру. На схеме показаны компоненты поверх выявленных областей, а также их владельцы (для простоты понимания на рис. 6.7 представлены только те компоненты, владельцами которых являются участники команды А). Опираясь на эту схему, можно проанализировать когнитивную на-

First 15 Years» (Leanpub, 2020), с. 37–53. Подробнее об этой и других техниках визуальной совместной работы см. в книге под редакцией Кенни Баас-Швеглера (Kenny Baas-Schwegler) и Жоао Розы «Visual Collaboration Tools» (Leanpub, 2020), где рассматриваются как сами техники, так и истории их применения на практике.

[1] Источник: Брандолини А. «Discovering Bounded Contexts with EventStorming».

грузку команд. Анна видит, что владение компонентами, которые разбросаны по всему ландшафту, — сложная задача для любой команды: ее участникам требуются знания из разных направлений бизнеса, чтобы понимать цели создания и проектирования продуктов.

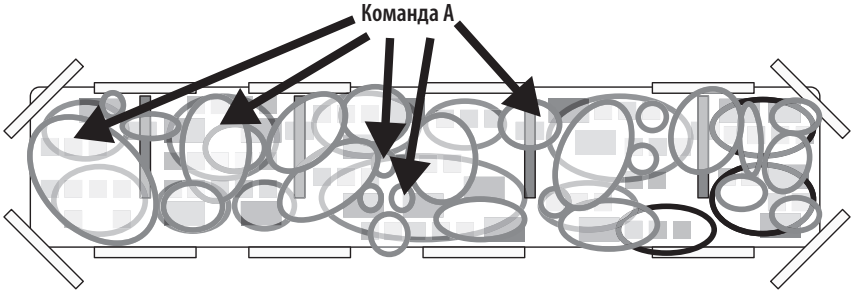


Рис. 6.7. Обозначение на схеме текущих программных компонентов и их владельцев (в упрощенном виде) в рамках возникающих ограниченных контекстов на воркшопе по Big Picture EventStorming

Воркшоп дает много новой информации и позволяет группе обсудить, как в YourFinFreedom организован процесс создания ценности для клиентов. Теперь сотрудники, которые отвечают за бизнес-операции, понимают текущую архитектуру и связанные с ней ограничения, а те, кто создает и сопровождает программный продукт, — детали бизнес-операций и то, как продукт обеспечивает их реализацию.

В заключение воркшопа Анна хочет сделать последний шаг: нанести на схему показатели KPI по разным направлениям, используя знания присутствующих специалистов (рис. 6.8). Профильные эксперты с воодушевлением берутся за дело и объясняют, почему те или иные KPI важны для их направлений. Таким образом они делятся знаниями, которые ранее существовали в организации лишь в неявном виде. Все довольны результатами воркшопа и тем, какой объем информации удалось собрать за короткое время.

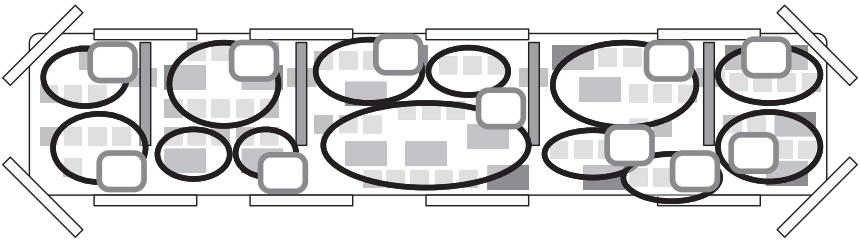


Рис. 6.8. Обозначение на схеме показателей KPI для предметных областей в ходе воркшопа Big Picture EventStorming

Метод Big Picture EventStorming позволяет организациям дополнять исходные обозначения и фиксировать информацию, необходимую для конкретного контекста или случая. Он помогает извлечь важные знания и глубоко осмыслить текущее состояние системы, а также актуальность того или иного KPI или метрики. Нанесение KPI и метрик на схему и обсуждение их назначения может инициировать важные диалоги на всех уровнях организации.

Для примера приведу свой опыт работы с клиентом, чье ПО обслуживает взаимодействие между налоговыми органами и морскими портами. В воркшопе этой компании по Big Picture EventStorming участвовали сотрудники из разных функциональных отделов — от службы поддержки клиентов до юристов по налоговому праву. Они вместе визуализировали рабочие процессы компании, в том числе взаимодействие между ними самими. Когда специалист направления поддержки клиентов объяснил, каким образом сотрудники отдела поддержки иногда, в контексте сделки по морскому судну, вносят исправления в налоговые документы, юрист по налоговому праву предупредил, что такие действия нарушают местное законодательство. Это открытие позволило компании исправить ПО и избежать серьезных финансовых и юридических последствий. Группе также удалось обсудить, как улучшить рабочий процесс поддержки клиентов и какие KPI потребуются для оценки предложенных усовершенствований. Команда, которая создавала и обслуживала ПО по этому направлению, высоко оценила полученную информацию, что позволило ей пересмотреть приоритеты в бэклоге. Они как раз занимались обновлением ПО для поддержки новых правил Brexit, регулировавших взаимодействие между ЕС и Великобританией, так что полученная информация пришлась как нельзя кстати.

Я считаю, что программную архитектуру необходимо создавать целенаправленно, для решения конкретных задач. Для этого нужно визуализировать рабочие процессы и понять, каким образом используемое ПО помогает достигать намеченных целей. Согласование миссии организации с ее KPI позволяет формировать программную архитектуру и находить баланс между компромиссными решениями.

Используйте метрику, чтобы повысить соответствие программной архитектуры поставленным целям

После семинара Анна чувствует, что находится на верном пути, но не может двигаться дальше, имея на каждый процесс взгляд лишь «с высоты птичьего

полета». Воркшоп выявил несколько проблемных мест, а также несоответствия между процессами и программной архитектурой. Анна решает изучить процессы предметной области еще немного глубже. Для этого она хочет составить карту одного операционного потока создания ценности в одной из предметной областей, подробно описав KPI, проблемные места и каждый шаг этого потока. Имея на руках такую информацию, она сможет вместе с командами продуктовой разработки составить план улучшения архитектуры и привести ее в соответствие с KPI организации.

Соответственно Анна организует новый воркшоп EventStorming — на этот раз для моделирования процессов. На этом воркшопе участники подробно разберут операционный поток создания ценности по предметной области страхования путешественников; сам же поток решили назвать «запрос стоимости полиса страхования путешественников»[1].

Повышение соответствия программной архитектуры поставленным целям с помощью упомянутых двух процессов — визуализации процессов и понимания пользы от ПО — помогает понять не только систему в целом, но и ее отдельные части. Для меня это *увлекательный процесс познания*, в ходе которого мы визуализируем текущие процессы и выявляем их ограничения. У большинства организаций нет подробных схем своих процессов (а если и есть, то, как правило, они устарели). Используя методы совместной визуализации процессов, такие как EventStorming, мы можем наглядно представлять коллективные знания, которые находятся в голове у всех участников.

Прежде чем принимать решения о текущей программной архитектуре, Анна просит группу добавить важнейшие KPI для операционного потока создания ценности. Оказалось, что у каждого свой взгляд на определения важнейших KPI! Анна помогает группе прийти к единому мнению и более четко определиться с нужными KPI. Увидев, что полученный результат поможет согласовать бизнес-процессы и процессы продуктовой разработки, участники испытали чувство удовлетворения от того, что проделали большую и важную работу.

Чтобы повысить уровень детализации, Анна просит группу отметить на схеме моделирования процессов EventStorming текущие проблемные места. Ей важно понять, какие трудности в повседневной работе испытывают коллеги и каким образом существующие программные решения помогают или мешают их пользователям. Анна начинает ясно видеть сложности и несоответствия между программной архитектурой и исследуемым операционным потоком создания ценности. И последнее, не менее важное — все наконец увидели, как именно

[1] По ссылке https://oreil.ly/Vc4y5 можно посмотреть, как выглядит переход от высокоуровневого EventStorming к детализированному EventStorming для моделирования процессов в рамках одного операционного потока создания ценности.

YourFinFreedom создает ценность для своих клиентов и за счет чего получает доход. В частности, в процессе «запрос стоимости полиса страхования путешественников» YourFinFreedom получает комиссию за каждый активированный страховой полис и, соответственно, стремится максимизировать их количество. С точки зрения клиента, процесс получения полиса должен быть максимально гладким, лучшим по цене и по страховому покрытию в своей области.

Нанесение KPI на схему в рамках воркшопа EventStorming по моделированию процессов позволяет детализировать, что именно является наиболее важным в операционном потоке создания ценности и каким образом организация измеряет эту ценность[1]. Ценность может создаваться как для клиента, так и для самой организации. Когда вы придете к такому уровню детализации, рекомендую задаться вопросом: «На чем компания зарабатывает деньги?» При разработке ПО критически важно понимать бизнес-модель, при этом архитектура ПО должна облегчать создание ценности.

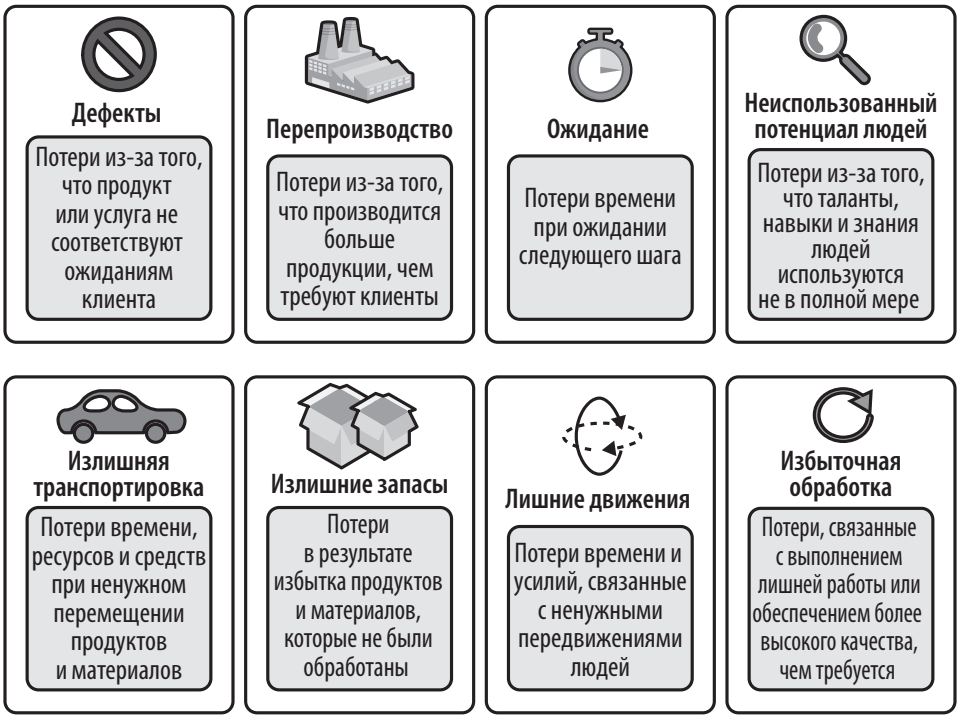
Крайне важно обеспечить понимание KPI и конкретизировать их определение. На это может уйти много времени, поэтому не исключено, что встреч будет несколько. Ваша история может отличаться от истории Анны; в начале своего профессионального пути я на горьком опыте убедился, что не стоит принимать информацию на веру и бояться задавать неудобные вопросы. У меня есть принцип: *«Тише едешь — дальше будешь»*. В данном контексте это означает, что время, вложенное в глубокое понимание KPI, а значит, и будущей среды для работы продукта, окупится при разработке его архитектуры.

Тот же подход применим и к проблемным местам. С какими трудностями сталкиваются пользователи системы? Понять, где возникают *потери*, можно также проанализировав проблемные места. В концепции бережливого производства (Lean), в частности в том виде, в котором она используется в производственной системе Toyota, потери определяются так: «идеальные условия для производства создаются, когда машины, оборудование и люди работают сообща, добавляя ценность и не допуская никаких потерь»[2]. Потери связаны с эффективностью процесса и сказываются на ценности, которую получает клиент. Наврас Схмот (Nawras Skhmot) пишет, что производственная система Toyota выделяет восемь видов потерь (рис. 6.9).

---

[1] По ссылке https://oreil.ly/IIVUI можно увидеть результаты воркшопа EventStorming по моделированию процессов с указанием KPI и проблемных мест.

[2] «Toyota Production System», Toyota, дата обращения: 30.03.2022, https://oreil.ly/pLnAp.



Источник: The Lean Way

Рис. 6.9. Восемь видов потерь

Детальное рассмотрение принципов бережливого производства выходит за рамки этой главы. Однако я настоятельно рекомендую использовать их в качестве одного из инструментов, который при поиске компромиссов повысит соответствие программной архитектуры поставленным целям. Это поможет провести архитектурные границы так, чтобы способствовать реализации миссии компании и ее целей.

Далее Анна организует работу по декомпозиции предметной области. Группа обсуждает конечный результат и цели предметной области «Страхование путешественников», а затем выполняет декомпозицию целей, чтобы понять, как преобразовать ключевые показатели эффективности (KPI) в метрики (рис. 6.10). Все четко понимают, как взаимосвязаны направления деятельности каждого, соответственно, и команда отдела продуктовой разработки понимает, что именно важно для этой предметной области. Таким образом Анна получает последний недостающий элемент собираемого пазла. Теперь она может руководить улучшением программной архитектуры.



Рис. 6.10. Декомпозиция предметной области «Страхование путешественников»: от конечного результата к метрикам

Декомпозиция сведений о предметной области — от конечных результатов до целей и метрик — позволяет детальнее описать ее назначение и задачи. Со временем цели и метрики меняются вместе с архитектурой, но с разной скоростью. При изменении архитектуры метрики, как правило, меняются чаще, а цели — медленнее. Важную роль в этом играют и бизнес-факторы. Метрики теснее связаны с программной архитектурой, чем цели, однако именно цели задают общее направление.

Метрики также можно реализовать в виде *фитнес-функций* — этот термин был введен в книге «Building Evolutionary Architectures»[1]. По определению авторов, архитектурная фитнес-функция «предоставляет объективную оценку целостности одной или нескольких характеристик архитектуры».

Собрав всю информацию, Анна и команда отдела продуктовой разработки, который отвечает за предметную область «Страхование путешественников»,

[1] Форд Н., Парсонс Р., Куа П. «Эволюционная архитектура. Автоматизированное управление программным обеспечением», 2-е межд. изд. СПб., издательство «Питер». Подробнее о фитнес-функциях см. в главах 2 и 8.

приступают к составлению плана. Они анализируют центральный микросервис с информацией о клиентах и приходят к выводу, что он на самом деле совсем не «микро». Этот сервис хранит все сведения о клиентах, включая используемые финансовые услуги. Архитектуры этого сервиса и сервиса полисов страхования путешественников тесно взаимосвязаны. И хотя за центральный сервис отвечает другая отдельная команда отдела продуктовой разработки, именно от этого сервиса зависит оркестрация релизов.

Группа планирует провести серию экспериментов. Ее участники хотят получить ответы на три вопроса:

- Может ли автономный сервис управлять полисами страхования путешественников?
- Сколько усилий в рамках предметной области «Страхование путешественников» потребуется, чтобы перенести зависимости из центрального сервиса в сервис полисов страхования путешественников?
- Сколько усилий потребуется для переноса данных между этими сервисами?

Чтобы убедиться в правильности полученных ответов, команда подробно описывает первый эксперимент в виде последовательности шагов. Они планируют создать автономный сервис, который будет хранить клиентские полисы страхования путешественников. Этот сервис будет работать как прокси-сервис для существующего центрального сервиса. Новые запросы стоимости полисов страхования путешественников будут сохраняться и в сервисе полисов, и в центральном сервисе. Цель — доказать, что сервис полисов страхования путешественников способен надежно хранить информацию.

Если все эксперименты пройдут успешно, команда намерена сделать сервис полисов страхования путешественников единственным источником истины для предметной области «Страхование путешественников» и устранить зависимость от центрального сервиса с информацией о клиентах. На основе информации из воркшопов команда решает добавить еще метрики. В качестве метрик, которые будут определять изменения в архитектуре и подтверждать правильность выбранного направления, они выбрали частоту развертываний и долю неуспешных изменений. Была выдвинута гипотеза, что создание сервиса полисов страхования путешественников и перенос в него соответствующей логики из центрального сервиса приведет к тому, что для центрального сервиса: 1) увеличится частота развертываний; 2) снизится доля неуспешных изменений. Кроме того, согласно гипотезе, потребуется пересмотр границ программной архитектуры, что должно повысить автономность ответственной команды отдела продуктовой разработки.

Чтобы связать всю информацию воедино, Анна воспользовалась деревом KPI и представила наглядно связи между KPI организации, KPI предметной области и вспомогательными метриками. Впервые за время работы в YourFinFreedom у нее сложилось полное представление о компании — от миссии до инициатив по продуктовой разработке.

Дерево KPI, показанное на рис. 6.11, состоит из трех уровней. Первый уровень — это KPI организации, второй — KPI предметной области, а третий — метрики. Как видно, KPI организации носят общий характер и измеряют состояние компании. Чаще всего они связаны с финансовыми результатами, например с EBITDA и CLV, но иногда включают и метрики, которые ориентированы на конечных пользователей, например MAU. На этом уровне KPI — это *запаздывающие показатели*: с их помощью можно проанализировать прошлое и понять, привели ли предпринятые действия к желаемому результату.

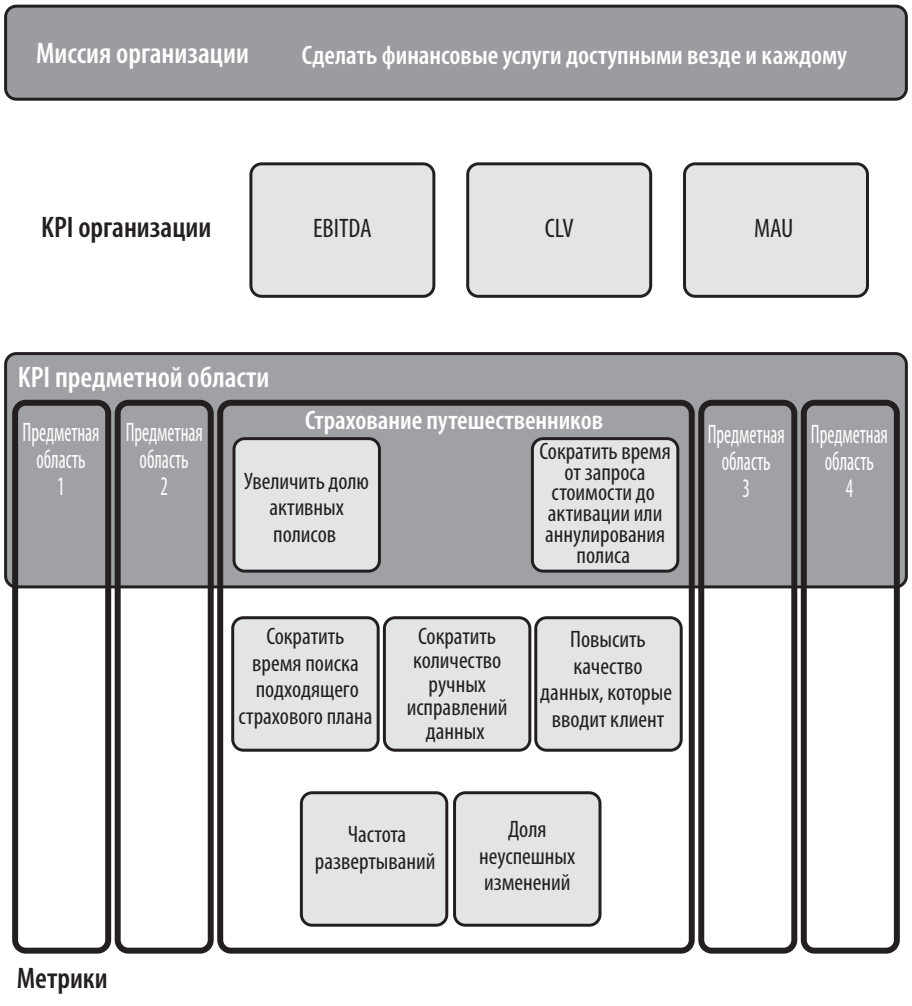


Рис. 6.11. Дерево KPI компании YourFinFreedom после первой серии воркшопов

Второй уровень ориентирован на предметную область «Страхование путешественников». Здесь KPI отражают увеличение доли активированных полисов и сокращение времени от запроса ценового предложения до активации или аннулирования полиса. Эти KPI также являются запаздывающими показателями.

Третий уровень содержит метрики. Метрики могут быть как запаздывающими, так и *опережающими показателями* — помогающими предсказать, что планируемое действие с большой долей вероятности приведет к ожидаемому результату. В данном примере запаздывающие показатели — это сокращение времени на поиск подходящего страхового плана, уменьшение количества ручных исправлений данных и повышение качества вводимых пользователем данных. Опережающие — это частота развертывания и доля неуспешных изменений.

Дерево KPI — это моментальный снимок. Поскольку и в бизнесе, и в жизни перемены — единственная постоянная, дерево KPI должно меняться по мере обсуждения полезности KPI и метрик. Важно, особенно в команде продуктовой разработки, постоянно обсуждать KPI и по необходимости их корректировать.

При масштабировании компании меняются и бизнес-факторы, и используемые технологии. Все эти изменения неизбежно сказываются на программной архитектуре. На каждом этапе развития следует основывать решения на данных (например, по KPI и метрикам), чтобы эти решения учитывали текущие ограничения (технологические, кадровые, нормативные, рыночные и т. д.).

Как гласит закон Гудхарта: «Когда мера становится целью, она перестает быть хорошей мерой»[1]. Помните: *KPI и метрики должны служить ориентирами и инструментами, а не целями*; они не должны предписывать поведение! По опыту, если кто-то начинает говорить: «Мы не достигаем KPI X или метрики Z», — это верный признак того, что KPI или метрика используется как цель. Если нужна такая цель — хорошо, но тогда не называйте ее *KPI* или *метрикой*.

В этом отношении программным архитекторам есть чему поучиться у сообщества SRE-инженеров (инженеров по обеспечению надежности), которое работает с соглашениями об уровне обслуживания (SLA, service-level agreement). SLA — это официально прописанный в договоре ожидаемый уровень обслуживания, который обязуется обеспечить определенный провайдер или вендор. Несоблюдение этого уровня влечет за собой по-

[1] См. ttps://ru.wikipedia.org/wiki/Закон_Гудхарта.

следствия для провайдера, как правило, финансовые. Если команда обеспечивает поддержку ПО согласно SLA, то для нее вполне логично иметь собственные цели. В таких ситуациях полезно наглядно представить KPI, метрики *и* цели и проанализировать, как они между собой взаимосвязаны.

Книга «Accelerate»[1], написанная Николь Форсгрен, Джезом Хамблом и Джином Кимом, оказала огромное влияние на всю сферу архитектуры и разработки ПО. В частности, широкое распространение получили предложенные в ней четыре метрики эффективности доставки и эксплуатации ПО (рис. 6.12): *время выполнения доставки, частота развертывания, доля неуспешных изменений* и *среднее время восстановления*. Эти метрики — опережающие индикаторы скорости и стабильности. Исследование авторов показывает, что эти четыре технические метрики используются в высокоэффективных командах. Однако каждая команда работает в уникальном контексте, поэтому существуют и другие релевантные метрики, которые команды продуктовой разработки могут и должны использовать.

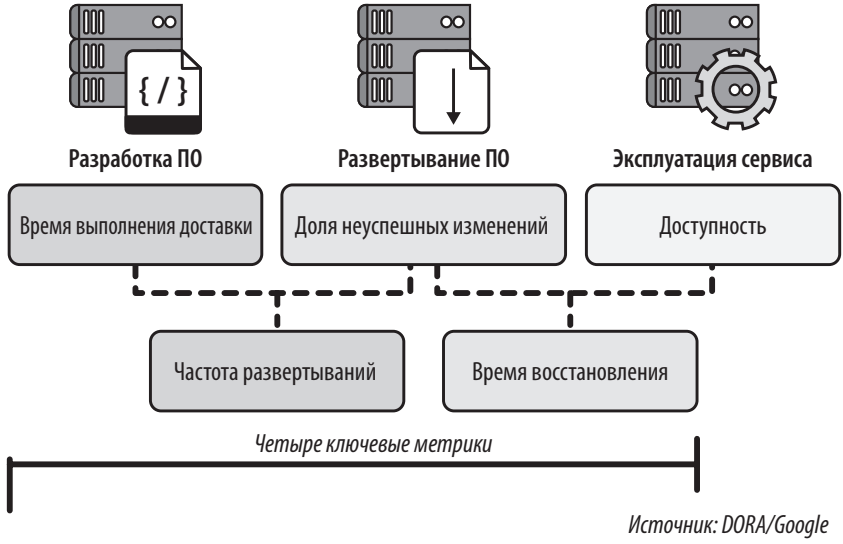


Рис. 6.12. Метрики DORA, © DORA/Google

Я также рекомендую использовать метрику *среднего времени обнаружения инцидента* (mean time to discover), если внедрены надлежащие практики

[1] Форсгрен Н., Хамбл Д., Ким Дж. «Ускоряйся! Наука DevOps. Как создавать и масштабировать высокопроизводительные цифровые организации».

и инструменты для мониторинга системы. Среднее время обнаружения инцидента — это среднее значение времени, которое проходит с момента возникновения ИТ-инцидента до его обнаружения.

Поскольку команды продуктовой разработки работают в социотехнической системе, я применяю и другие проверенные метрики. Первая — *пропускная способность* (throughput), которая служит базовым показателем способности команды продуктовой разработки поставлять партии результатов своей работы. Она пришла из Lean-сообщества и была перенята DevOps-сообществом как инструмент для непрерывного совершенствования. Вторая — *индекс лояльности сотрудников* (employee Net Promoter Score®), который измеряет удовлетворенность и помогает повысить коэффициент удержания сотрудников.

Хотя я часто использую упомянутые выше метрики, ни одна из них не является универсальной. Я также не призываю бездумно применять их все! В этой книге описаны различные подходы к программной архитектуре и метрикам, а также опыт их использования, чтобы вы могли решить, что из предложенного применить в своей организации. И последнее, но не менее важное: сравнивать производительность команд на основе этих метрик контрпродуктивно, поскольку у каждой команды свои области знаний, уровень квалификации, технологии и другие факторы, которые определяют контекст.

Пару лет назад я участвовал в DevOps-трансформации одного предприятия, которое хотело повысить прозрачность жизненного цикла программной разработки. Я курировал отдел, разрабатывавший мобильные приложения для розничного банкинга. Компания использовала четыре метрики DORA для оценки эффективности доставки и эксплуатации ПО, — и до поры до времени все шло хорошо. Однако некоторые сотрудники настаивали на том, чтобы ввести пороговые значения и классифицировать команды по всей организации на основе их «производительности». Я намеренно пишу слово «производительность» в кавычках, потому что команды работали в совершенно разных условиях. Как нетрудно догадаться, в этой организации использовались самые разные технологии: от мейнфреймов до бессерверных облачных решений и всего, что между ними. Я объяснил, что метрики зависят от контекста и нельзя ограничиваться одним лишь отслеживанием показателей DORA.

В программной разработке принято считать, что частоту развертываний нужно повышать. Это позволяет снизить затраты, избавиться от неуверенности и страха, а также повысить уровень стандартизации, предсказуемости и автоматизации. Однако в случае с мобильными приложениями

частые развертывания приводят к постоянным уведомлениям об обновлениях, что раздражает клиентов и снижает индекс потребительской лояльности (Net Promoter Score®) компании. Отдел, про который идет речь, вложил силы и средства в то, чтобы иметь возможность развертывать изменения в любое время. Они решили использовать подход с *кольцевыми развертываниями* (deployment ring), в котором было три кольца, или три среды (альфа, бета и продакшен), и каждое изменение автоматически развертывалось в альфа-среде. Такой подход соответствовал их контексту. Именно в этом и заключается сила целостного подхода к увязке KPI с метриками.

Подобная увязка может потребовать много времени и труда. В истории Анны создание дерева KPI началось с нее самой, поскольку она работает в скейлап-компании, где коммуникация между высшим руководством, архитекторами и инженерами происходит быстро. Использование такого артефакта, как дерево KPI, — это долгосрочный совместный подход, который требует согласованности между подразделениями организации и поддержки со стороны высшего руководства (ведь это дерево связано с KPI всей организации). Четкое распределение ответственности и процесс, который соответствует контексту и культуре организации, повышают вероятность того, что дерево KPI принесет устойчивый положительный эффект.

Управление ожиданиями и коммуникация

Анна знала, что в ходе эксперимента у отдела продуктовой разработки снизится производительность по выпуску нового функционала для страхования путешественников. Она встретилась с Кейшей, чтобы поделиться результатами воркшопов, деревом KPI и планом команды отдела продуктовой разработки. Там она подробно описала последствия и задала интересующий ее вопрос: как использовать точки влияния в архитектуре для снижения текущей сложности. Кейша одобрила план и пообещала поддержку, когда Анна будет представлять свои ключевые идеи стейкхолдерам. Она объяснила Анне, что очень важно сформировать правильные ожидания и поэтому при обсуждении Анна должна четко обозначить предсказуемые последствия, желаемый результат и то, как он поможет YourFinFreedom достичь цели — стать крупным игроком на рынке Евросоюза.

Участники встречи высоко оценили честность и четкость сообщения Анны, а также согласились на эксперимент в команде продуктовой разработки, результатами которого та поделится с другими областями, что поможет решить общие проблемы. Анна пообещала держать всех в курсе событий.

Далее команда отдела продуктовой разработки начала эксперименты: запустила в эксплуатацию сервис полисов страхования путешественников и заменила им

центральный сервис информации о клиентах. Эксперименты прошли успешно, что подтвердилось метриками. Границы старой программной архитектуры оказались некорректными, и их пересмотр в контексте предметной области «Страхование путешественников» означал, что его сервисам больше не требовалась оркестрация релизов. Руководство и стейкхолдеры остались довольны результатами и регулярной коммуникацией, а также поддержали Анну, когда она предложила применить тот же подход к другим предметным областям организации. Она начала использовать те же методы, чтобы помочь командам продуктовой разработки провести аналогичные эксперименты в других областях и создать программную архитектуру с четкими границами.

Тем временем, по мере роста YourFinFreedom и улучшения KPI для страхования путешественников, его руководители пришли к мнению, что текущие метрики, хотя и стабильны, и отвечают нынешним потребностям, возможно, уже неактуальны. Страховые аналитики, ссылаясь на метрику «Сократить время на поиск подходящего страхового плана», спрогнозировали, что потребуется нанять больше сотрудников, занимающихся подготовкой страховых планов, и это количество будет расти линейно с увеличением доли рынка YourFinFreedom. Однако это отразится на EBITDA. Руководители направления встретились с Анной, чтобы обсудить возможность сократить объем ручной работы, необходимой для подготовки плана страхования путешественников. Анна предложила обновить для этого архитектуру, и они обсудили, какие метрики лучше всего подойдут для поддержки изменений. Решили заменить метрику «Сократить время на поиск подходящего страхового плана» на «Увеличить долю принятия первых предложенных страховым аналитиком планов страхования» (рис. 6.13).

Анна вместе с командой отдела продуктовой разработки спроектировала архитектурные изменения. Они внедрили сервис, который получает и обрабатывает информацию от клиентов, направляет запросы страховым партнерам и генерирует несколько вариантов страховых планов. Таким образом, получилась система поддержки принятия решений для страховых аналитиков. Она сэкономила им время, которое они сейчас тратят на изучение систем страховых партнеров.

Чтобы дерево KPI приносило пользу, его необходимо поддерживать в актуальном состоянии. Когда меняются бизнес-факторы, архитектура ПО должна меняться, чтобы им соответствовать. Этот паттерн в том или ином масштабе повторяется снова и снова. Соответственно, когда происходят изменения, крайне важно перепроверять полезность KPI и метрик. В истории Анны команда отдела продуктовой разработки успешно прояснила и зафиксировала границы своей предметной области, что позволило ей развиваться. Там Анна выступила связующим звеном между разными функциями внутри предметной области «Страхование путешественников». У вас же может быть совсем иная операционная модель, и это нормально — поскольку, опять же, *все зависит от контекста*.

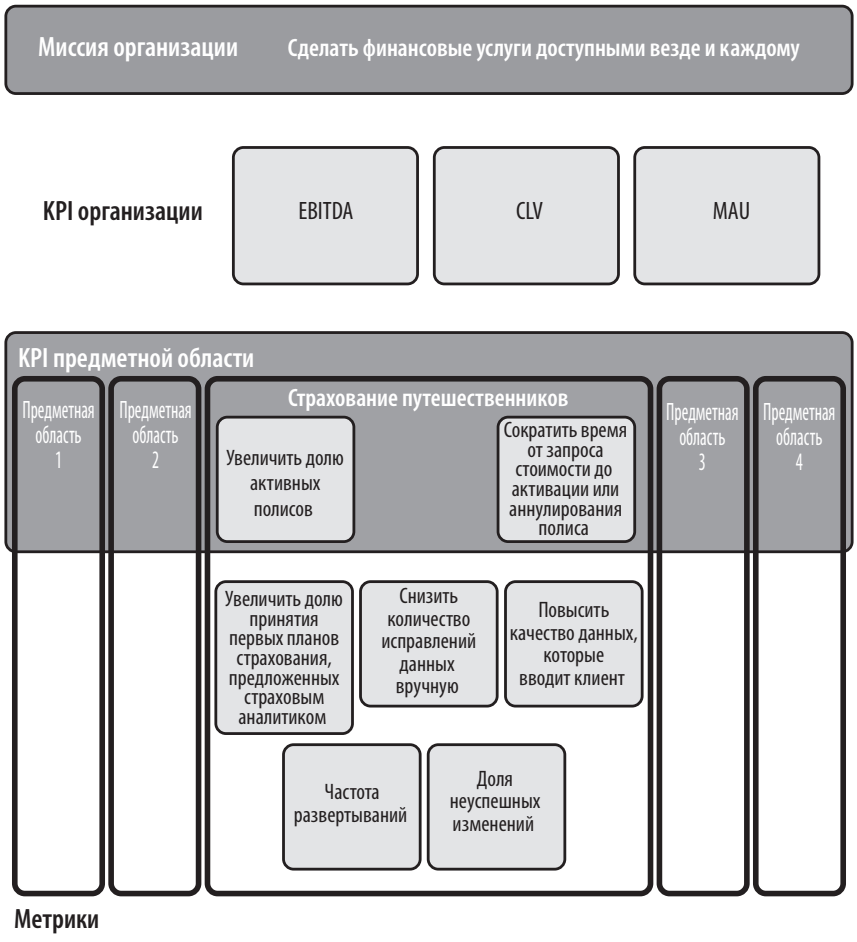


Рис. 6.13. Дерево KPI компании YourFinFreedom после изменения метрик в предметной области страхования путешественников

Отсюда следует, что метрики должны направлять естественное развитие программной архитектуры. В этом случае люди, которые создают и сопровождают ПО, могут вносить изменения более осознанно, даже если эти изменения носят сугубо экспериментальный характер. Вносить изменения полезно, и четкие границы делают этот процесс возможным, поскольку учитывают, что одним частям архитектуры важна стабильность, а другим — эксперименты.

Расскажу еще одну реальную историю об эволюции метрик. Я был техническим тимлидом в компании, которая занималась электронной коммерцией

и переносила свои системы в облако. Моя команда работала в предметной области пополнения запасов и отвечала за формирование товарных предложений для закупщиков. Основная цель заключалась в том, чтобы максимально повысить доступность товаров и одновременно сократить время их нахождения на складе. Система формирования товарных предложений — это система поддержки принятия решений для отдела закупок. Она помогает определить необходимый объем закупок на основе прогноза продаж и содержит информацию о поставщиках и ценах.

При переходе в облако для всех систем требовалось обеспечить надлежащую прозрачность процессов. Чтобы формировать товарные предложения, мы собирали данные с API сервиса, который определял оптимального поставщика (включая объем поставки). В сервисе была реализована вся логика выбора поставщика (и там были весьма специфические правила, в частности те, что имели юридические последствия). Помимо данных, мы собирали метрики, чтобы измерять долю успешных вызовов. И мы обратили внимание, что количество сбоев было довольно высоким. Для анализа ситуации мы создали метрику, которая сопоставляла неуспешные вызовы с конкретным товаром. Наш бизнес-аналитик выяснил, что жизненный цикл этих товаров подходил к концу: они больше не поставлялись, но из системы еще выведены не были. Когда мы скорректировали метрики и они стали учитывать эту информацию, количество ошибок API упало практически до нуля, а время выполнения всего процесса значительно уменьшилось. Процесс пополнения запасов стал учитывать меньше товаров, и количество ошибок API сократилось. Анализ взаимосвязей метрик, целей и ограничений, соответствующих предметной области, помог моей команде создать метрики, которые улучшили систему. Хотя эта работа потребовала времени и профильных знаний, в долгосрочной перспективе она положительно сказалась на системе.

Изучение и развитие архитектуры

В ходе работы с другими предметными областями Анна выявила три закономерности. 1. Программная архитектура не соответствовала границам предметных областей бизнеса, как и в случае с уже известной областью страхования путешественников. 2. Границы предметных областей были определены неверно, что приводило к непреднамеренной сложности в архитектуре. 3. На некоторые команды продуктовой разработки ложилась высокая когнитивная нагрузка, поскольку им принадлежали компоненты ПО, которые либо пересекали границы предметных областей, либо являлись частью разных операционных потоков создания ценности.

Пример второй закономерности — центральный сервис информации о клиентах. Хранение данных по всем предметным областям, связанных с финансами, требует

управления разным поведением для всех моделей этих областей, что и порождает сложность. Третью закономерность Анна наблюдала в команде, которая отвечает за онбординг клиентов и их биллинг. Поскольку эти процессы сильно различаются и не связаны друг с другом, команда не могла сосредоточиться на одном пространстве задач.

На встрече с Кейшей Анна представила свои идеи по решению этих проблем. Для начала она предложила поступить так же, как со страхованием путешественников: разделить центральный сервис информации о клиентах на соответствующие финансовые направления и создать новые сервисы для данных, связанных с управлением идентификацией и учетными записями. Она также предложила пересмотреть распределение ответственности за программные компоненты, чтобы ограничить когнитивную нагрузку команд продуктовой разработки на время стабилизации границ предметных областей. При этом она подробно объяснила возможные последствия для продуктовых разработчиков, чтобы Кейша увидела все риски. Кейша осталась под впечатлением от того, какую работу проделала Анна: благодаря ее идеям YourFinFreedom стала работать более четко и целенаправленно.

Какова мораль этой истории? *Используйте метрики, которые соответствуют контексту.* Также помните, что *иногда динамика изменения метрики важнее ее значений*.

Например, динамика среднего времени обнаружения инцидента — хороший косвенный показатель того, учатся ли сотрудники на своих ошибках. Допустим, что со временем эта метрика увеличивается. Тому возможны две причины. Во-первых, программная архитектура усложняется под влиянием бизнес-факторов; во-вторых, по мере повышения сложности увеличивается и когнитивная нагрузка, что отрицательно сказывается на вовлеченности сотрудников. Сопоставление метрики среднего времени обнаружения с долей неуспешных изменений может помочь выявить слабые места в архитектуре. Более того, сопоставление доли неуспешных изменений с индексом лояльности сотрудников (Net Promoter Score®) поможет руководителям организовать своевременную поддержку сотрудников. На этой простой истории мы увидели, как решения в области программной архитектуры могут влиять на людей, команды и социальную структуру компании.

Поддержка программной архитектуры — это непрерывный процесс: по мере создания новых артефактов и развертывания их на продакшен меняется ландшафт и развивается организация. То, что было актуально в прошлом, не обязательно остается таковым сегодня. В нашей вымышленной истории компания Анны начала переопределять границы предметных областей и, как следствие, границы программной архитектуры.

Когда происходят изменения подобного масштаба, крайне важно управлять всеобщими ожиданиями. Как мы увидели, выводы были сделаны и владель-

цы программных компонентов сменились. Смена владельцев повлияла на команды, а это, в свою очередь, вызвало последствия как технического, так и социального характера и потребовало некоторых компромиссов. Важно открыто говорить об этих процессах со всеми участниками, иначе люди потеряют ориентиры или интерес и, возможно, даже покинут компанию.

Программная архитектура, метрики и KPI — все должно работать на развитие организации. Руководителям необходимо понимать, что архитектура — это не техническая реализация стратегии в вакууме. Скорее, наоборот, именно программная архитектура и позволяет компании реализовать свою стратегию.

Я считаю, что нам нужно новое поколение программных архитекторов, которые умеют не только группировать паттерны между техническими компонентами, но и проводить воркшопы, понимать динамику группы и участвовать в формировании общей бизнес-стратегии. Если мы будем понимать, как технические решения влияют на социальную структуру, нам будет проще создавать надежные архитектуры, в развитии которых люди с удовольствием будут участвовать. В выигрыше будут все.

Как сейчас дела у Анны?

Спустя два с половиной года после начала своего долгого пути Анна сидит в корпоративном кафе, пьет фруктовый чай и читает почту. Ей на глаза попадается приглашение: ее просят поделиться своей историей в докладе на конференции по программной архитектуре. Она размышляет о том, к чему пришла: компания перевыполняет планы и растет быстрее, чем ожидалось. Руководство компании обсуждает, как применить новые навыки в области социотехнической архитектуры для освоения других направлений бизнеса. Анна решает назвать свой доклад «Социотехническая архитектура: больше чем программная». Как думаете, о чем она расскажет?

Заключение

Программная архитектура и метрики — увлекательные темы. В этой главе на примере истории Анны я попытался поделиться своим опытом. Мы, технические специалисты, привыкли концентрироваться на тонкостях технологий и переусложнять проекты и решения. Из-за них в программной архитектуре проявляется непреднамеренная сложность. Она, в свою очередь, влияет на социальную структуру организации, поскольку люди ожидают большего и как минимум разочаровываются. В последние годы я заметил, что топ-менеджмент стал лучше разбираться в социотехнической

архитектуре и чаще старается принимать более взвешенные технологические решения.

Крайне важно, чтобы программная архитектура соответствовала специфике бизнеса и масштабу организации. Процесс создания архитектуры небыстрый, и она развивается вместе с организацией. Соответственно очень важно взаимодействовать с разными подразделениями, чтобы понимать, как идут рабочие процессы и как создается ценность. Это понимание позволит разработать метрики, которые будут не только отражать состояние программной архитектуры, но и поддерживать бизнес. Чтобы согласовать действия разных подразделений, Анна решила визуализировать совместную работу. Согласование действий позволило получить ценные идеи, которые команды продуктовой разработки превратили в стратегический план для YourFinFreedom. Программная архитектура компании развивалась, а с ней и цикл обратной связи, который позволял оценивать состоятельность метрик. Такая совместная работа — не разовая акция. Необходимо постоянно обеспечивать общее видение процесса построения архитектуры у всех участников.

При этом важно также учесть контекст и найти подходящие метрики. Копировать чужие решения бесполезно, потому что у каждой организации свой контекст. Поиск метрик, подходящих именно вашей организации, — это целенаправленное вложение сил, которое окупается по мере ее развития.

Важно наблюдать метрики в динамике, поскольку так можно увидеть сигналы, выходящие за рамки чисто технических вопросов. Такие сигналы тесно связаны с социотехнической архитектурой, ведь в конечном счете именно люди проектируют, создают и обслуживают программы. И я действительно считаю, что сейчас на передний план выходит новое поколение архитекторов, обладающих навыками для решения этих социотехнических задач.

ГЛАВА 7

Роль метрик в программной архитектуре

Эойн Вудс

Определений *программной архитектуры* существует множество, но на практике большинство архитектурно значимых решений направлено на достижение таких свойств системы, как производительность, отказоустойчивость и безопасность, уровень которых устраивает стейкхолдеров. Архитекторы часто называют такие свойства атрибутами качества либо нефункциональными качествами (или нефункциональными свойствами).

Важные и сложные архитектурные решения зачастую принимать трудно, поскольку они требуют серьезных компромиссов. Например, приоритет отказоустойчивости может привести к снижению производительности. Часто стейкхолдеры не могут четко определить, какие атрибуты качества им необходимы.

Раньше для решения этих проблем архитекторы уделяли много времени предварительному проектированию и анализу, чтобы детально проработать требования, рассмотреть компромиссы, принять ключевые архитектурные решения и проверить их. Однако сегодня необходимо действовать гораздо быстрее и адаптироваться к изменениям эффективнее, чем позволяет такой подход.

Ряд подходов, таких как непрерывная доставка ПО[1], RCDA[2] и непрерывная архитектура[3], направлены на то, чтобы работа над архитектурой велась

---

[1] См. Хамбл Д., Фарли Д. «Непрерывное развертывание ПО: автоматизация процессов сборки, тестирования и внедрения новых версий программ».

[2] См. статью Элцйо Р. Порта (Eltjo R. Poort) и Ханса ван Влита (Hans van Vliet) «RCDA: Architecting as a Risk- and Cost Management Discipline», Journal of Systems and Software 85 (2012): 1995-2013.

[3] См. Эрдер Мурат (Murat Erder); Пурер Пьер (Pierre Purer); Вудс Эойн. Continuous Architecture in Practice — изд-во Addison-Wesley, 2021.

не столько на начальном этапе, сколько на всем протяжении жизненного цикла доставки. Это позволяет командам принимать важные решения на более поздних этапах, когда появляется больше информации, и поддерживать изменения, возникающие в процессе создания системы.

Сложность непрерывной архитектуры, как и сложность связанных с этим подходов, заключается в том, чтобы понять, достаточно ли было проделано работы и тратятся ли усилия на самые важные задачи, чтобы результат был максимальным. Задач всегда больше, чем времени на их выполнение, поэтому нужно уметь грамотно расставлять приоритеты, а также понимать, когда следует остановиться и перейти к следующей проблеме.

Решение — в метриках. Измеряя качества системы в конкретный момент времени, а не полагаясь на интуицию или жесткий архитектурный «метод», можно понять текущее положение дел. Кроме того, проводя измерения в динамике, можно отслеживать тенденции и понимать направление развития по тому, как меняются эти качества. Такой подход к метрикам помогает правильно направлять усилия и извлекать из них максимум пользы.

Далее в этой главе мы обсудим, как встроить метрики в работу над программной архитектурой, и рассмотрим общие подходы к измерению и оценке атрибутов качества, а также некоторые частные методики для ключевых из них. Затем мы разберем, с чего начать, и обсудим некоторые распространенные ошибки.

Внедрение метрик в программную архитектуру

В исторической перспективе работа над программной архитектурой выглядела примерно так: определить, что нужно создать, спроектировать это, реализовать, а затем начать измерять. Этот подход напоминает «водопадную» модель доставки, но даже итеративно работающие команды (по Agile или другим методикам) часто откладывают измерение на поздние этапы жизненного цикла, уже после того, как ПО попало в среду эксплуатации.

Сегодня мы понимаем, что необходим *непрерывный* процесс «определение — проектирование — сборка — развертывание» и что использовать метрики для оценки нужно постоянно и начинать это делать надо как можно раньше. Связь между метриками и программной архитектурой показана на простой диаграмме (рис. 7.1).

Получение метрик из пайплайна доставки ПО и из любых сред, где развернуто ПО, должно выполняться непрерывно и часто. Эти метрики служат основой для работы, позволяют расставлять приоритеты и приводят

к принятию архитектурных решений. На основе этих решений в систему вносятся изменения, а затем с помощью дополнительных метрик определяется, были ли эти решения эффективными. И далее по кругу.

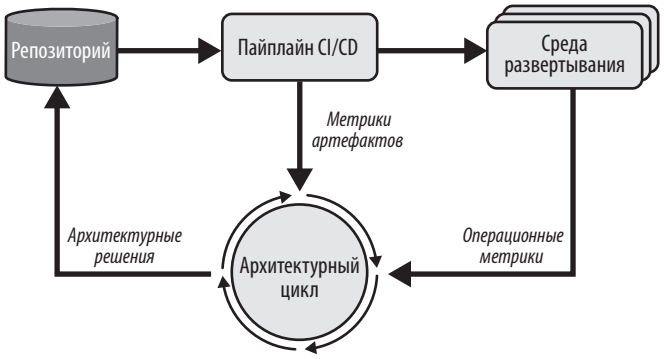


Рис. 7.1. Метрики в архитектурном цикле

Как видно на рис. 7.1, существуют *метрики артефактов* — то есть метрики, которые можно получить из артефактов, создаваемых в процессе доставки ПО, например из документов и кода. Также на рисунке упоминаются *операционные метрики* — то есть метрики системы, работающей в среде эксплуатации, например время отклика и использование дискового пространства. Полезно разобраться, когда можно получать каждый тип метрик и какие выводы они позволяют сделать. На рис. 7.2 приведены примеры метрик, классифицированных по двум осям: «метрики артефактов/операционные метрики» и «внешние/внутренние метрики».

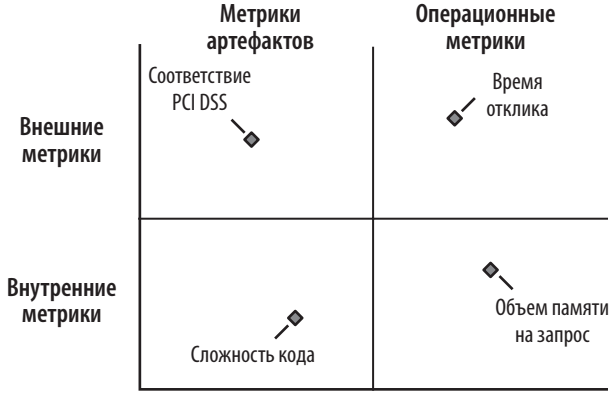


Рис. 7.2. Типы метрик

Внешние метрики заметны людям, не входящим в команды доставки ПО и эксплуатации системы, или напрямую влияют на них. *Внутренние метрики* заметны командам доставки ПО и эксплуатации или напрямую влияют на них. Оценка этих метрик на разных этапах цикла доставки ПО позволяет получать разноплановую аналитику. Рассмотрим каждый квадрант по очереди.

Внешние метрики артефактов

Проверка соответствия стандарту или набору рекомендаций по проектной документации — это, вероятно, самый слабый тип метрик, поскольку он основан на субъективных оценках. Однако у этих проверок есть большое преимущество: их можно проводить на самых ранних этапах доставки, когда дизайн только формируется. Внешние метрики артефактов помогают убедить стейкхолдеров в том, что система будет обладать определенными характеристиками (например, соответствовать GDPR) или отвечать конкретным стандартам и лучшим практикам.

Внутренние метрики артефактов

Такие метрики, как уровень сложности кода, уровень связанности модулей и количество элементов в схеме базы данных, очень конкретны и точны, а получать их, как правило, можно довольно быстро и недорого. Они служат полезными индикаторами таких атрибутов качества, как удобство сопровождения и расширение функциональности, и представляют интерес в первую очередь для команд разработки и эксплуатации. Недостаток этих метрик в том, что для их получения необходимо, чтобы соответствующие артефакты (как правило, код) были готовы.

Внешние операционные метрики

Эти метрики — эксплуатационные характеристики системы, которые влияют на работу сотрудников, не входящих в команды разработки и эксплуатации. В эту категорию попадают многие классические метрики атрибутов качества, такие как время отклика, пропускная способность, время восстановления после сбоя и количество сбоев в месяц. Как и внутренние операционные метрики, их нельзя получить, пока система не будет введена в эксплуатацию. Однако их преимущество в том, что они отражают опыт взаимодействия с системой самых важных стейкхолдеров — пользователей.

Внутренние операционные метрики

Эти метрики отражают характеристики эксплуатации системы, которые видны командам разработки и эксплуатации, например использование

памяти или скорость роста индекса базы данных по сравнению со скоростью роста данных. Эти метрики также требуют полностью работоспособной системы, но при этом дают ценное представление о том, как она работает, с внутренней точки зрения и с точки зрения пользователей. Эти метрики часто полезны для прогнозирования будущих проблем в системе.

На разных этапах жизненного цикла доставки ПО актуальны разные типы метрик, но со временем приходится использовать их все.

Способы получения метрик

Важно разработать надежные и экономически эффективные способы получения метрик, и ниже мы рассмотрим некоторые из них.

Получение метрик приложений и инфраструктуры в работающей системе

Как отмечает Синди Шридхаран (Cindy Sridharan) в книге «Distributed Systems Observability» (O'Reilly, 2018), сегодня доступны три основных типа механизмов получения метрик: логи, трассировки и, собственно, само измерение. *Логи* представляют собой последовательности записей о событиях с метками времени, которые показывают, что происходило в том или ином техническом компоненте системы с течением времени. *Трассировки* развивают эту идею, предлагая наборы взаимосвязанных событий, которые фиксируют сквозной межкомпонентный сценарий в ПО, например обработку запроса. *Данные измерений* — это прямые числовые показатели характеристик системы за определенный период, например загрузка ЦП виртуальной машины или размер хранилища образов. Собирать логи, трассировки и данные измерений можно как из инфраструктуры системы, так и из самого приложения.

Обычно проще получать логи, трассировки и данные измерений из инфраструктуры, поскольку большинство инфраструктурных сред, таких как публичные облачные платформы, имеют сложные и полнофункциональные системы сбора информации, которые без особого труда предоставляют все три способа получения данных. Сбор логов, трассировок и данных измерений приложений часто требует больше усилий, поскольку их приходится реализовывать самостоятельно, либо напрямую, либо используя существующий механизм получения метрик, например инструмент для управления производительностью приложений (APM, application performance management). Однако метрики приложений зависят от контекста, что может дать глубокое представление о характери-

стиках, которые действительно важны для стейкхолдеров (например, не о чисто технических показателях, а о бизнес-метриках типа «полученный доход»).

Анализ ПО

«Код не врет» — гласит старая поговорка. Написанный код — богатый источник метрик. Статический анализ кода — это хорошо развитая область с широким спектром мощных инструментов, однако он ограничен выявлением распространенных ошибок программирования и структурных характеристик кода (таких, как сложность или связанность). Да, он не помогает оценить многие архитектурные качества, кроме разве что удобства сопровождения и расширения. И тем не менее для оценки таких качеств, как безопасность, метрики анализа кода (например, количество найденных уязвимостей) могут быть косвенно полезны.

Анализ дизайна

Как уже говорилось, очевидная проблема анализа кода в том, что его нельзя использовать, пока код не написан, поэтому он является источником ретроспективных, а не предиктивных метрик. Описание некоторых параметров дизайна до его реализации позволяет с помощью анализа получить предиктивные метрики, например соответствие стандартам, или оценить ожидаемые архитектурные качества, такие как масштабируемость. Я ни в коем случае не рекомендую создавать, как раньше, подробную документацию, которая устареет сразу после написания, однако минимальное, но точное представление дизайна системы до начала его реализации может обеспечить глубокое понимание ожидаемых характеристик и сориентировать работу в нужном направлении.

Оценки и модели

Использование моделей и оценок для получения предиктивных метрик помогает направлять работу над архитектурой на ранних этапах цикла доставки ПО еще до того, как будет написан основной объем кода.

Можно использовать прошлые наработки, метрики других подобных систем и опубликованные результаты тестов и бенчмарков для создания математических моделей, как правило, в виде электронной таблицы. Задачей таких моделей будет отражение основных взаимосвязей между операционными параметрами (например, размер базы данных, объем запросов, типы запросов, количество серверов и объем памяти) и результирующими значениями атрибутов качества.

У такого рода предиктивных метрик есть несколько особенностей. Прежде всего, этот подход лучше всего работает для архитектурных качеств, которые легко представить численно, например для масштабируемости и производительности. Для таких качеств, как безопасность, его применять сложнее. Кроме того, для любой системы, кроме простейшей, может быть трудно (и дорого) создать достаточно несложную для понимания модель, которая даст надежные результаты. Проверить прогностическую ценность таких моделей также сложно, пока система не будет создана — тогда можно будет просто получить ее реальные метрики. Соответственно, важно убедиться, что создаваемая модель действительно полезна.

Фитнес-функции

Нил Форд, Ребекка Парсонс и Патрик Куа в своей книге «Building Evolutionary Architectures»[1] (O'Reilly, 2017) предложили использовать «фитнес-функцию» для получения метрик атрибутов качества. *Фитнес-функция* — это не новый механизм для получения метрик, а скорее способ их использования для контроля атрибутов качества системы и гарантии того, что они остаются в допустимых пределах. Подробнее о фитнес-функциях см. в главах 2 и 8 нашей книги.

Фитнес-функция определяет допустимое значение или набор значений для одного или нескольких атрибутов качества, а также способ проверки того, что система действительно им соответствует. В идеале фитнес-функцию следует реализовывать в виде автоматизированного процесса. Однако многие нужные функции автоматизировать невозможно, поэтому фитнес-функции, которые вычисляются вручную, например в электронной таблице, тоже могут быть полезны. Простой пример: если все запросы определенного типа должны обрабатываться в течение 100 мс, можно создать в среде эксплуатации автоматизированную фитнес-функцию, которая будет отслеживать время ответа на запросы и оповещать, если оно станет слишком большим.

Фитнес-функции не упрощают измерение атрибутов качества, поэтому все равно придется использовать методы, описанные в этом разделе. Но они *могут* помочь использовать полученные метрики, чтобы вести работу над архитектурой в нужном направлении.

---

[1] Форд Н., Парсонс Р., Куа П. «Эволюционная архитектура. Автоматизированное управление программным обеспечением», 2-е межд. изд. СПб., издательство «Питер».

Измерение свойств системы

Общие методы, описанные в предыдущем разделе, применимы к целому ряду свойств (качеств) системы, но каждое из важных свойств измеряется по-своему. В этом разделе подробнее рассмотрим специфику измерения некоторых ключевых свойств системы.

Производительность

Под *производительностью* (performance) понимается, насколько быстро система может обработать определенный объем рабочей нагрузки (например, ответить на веб-запрос, обработать вызов API, обработать сообщение о событии или завершить пакетное задание). Производительность обычно оценивается с помощью внешних и операционных метрик и является одним из атрибутов качества, которые легче всего измерить, поскольку ее можно напрямую оценить численно как в тестовой среде, так и в среде эксплуатации, а также частично с помощью модели.

Две стандартные метрики производительности — это *задержка* (latency): время, которое требуется для выполнения определенной работы, и *пропускная способность* (throughput): сколько единиц определенного типа рабочей нагрузки система может обработать за заданное время. Например, при обработке входящих сообщений о событиях задержка — это время, которое требуется для обработки одного сообщения, а пропускная способность — количество сообщений, которое один процессор может обработать за минуту. Эти две метрики должны быть обратно пропорциональны.

Важно не путать производительность с *масштабируемостью* (scalability) (см. следующий подраздел) — тем, насколько хорошо система реагирует на увеличение рабочей нагрузки. Эти два качества тесно взаимосвязаны, и снижение производительности часто является симптомом проблем с масштабируемостью.

Два основных фактора, которые определяют производительность, — это объем работы, который необходимо выполнить (например, обработать входящее сообщение о событии), и эффективность ее выполнения (например, используется ли индекс при доступе к базе данных).

Особенности измерения производительности

Ключевая метрика производительности — это время отклика, которое можно измерить непосредственно в тестовой среде или в среде эксплуатации. При этом используется ПО для тестирования (например, Gatling, https://gatling.io, для тестирования API и веб-интерфейсов), чтобы создать

синтетическую тестовую нагрузку, или в систему встраивается логирование для последующего ретроспективного анализа логов. Единичное измерение редко бывает полезно, поскольку производительность может меняться от запроса к запросу. Следовательно, лучше измерять распределение значений времени отклика для набора запросов, отражая его через среднее значение, медиану и стандартное отклонение.

Получив характеристику производительности для определенного типа рабочей нагрузки, можно сравнить ее с соответствующим требованием и на основе величины расхождения решить, требует ли ситуация корректировки архитектуры.

Производительность также требует компромиссов по стоимости: часто можно «купить» более качественные ресурсы выполнения (больше памяти, более быстрый ЦП, более быстрое хранилище), чтобы повысить производительность, но это обойдется дороже.

Наконец, как уже упоминалось, вместо измерения производительности напрямую можно рассмотреть ее оценку с помощью количественной модели (обычно в электронной таблице). Такие модели сложно построить так, чтобы они давали надежные прогнозы о будущей производительности, но подобные оценки могут позволить составить хоть какое-то представление о параметре системы еще до того, как его можно будет оценить напрямую.

Типичные проблемы при измерении производительности

При измерении производительности следует учитывать некоторые типичные проблемы:

Тестирование и реальность

Часто для измерения производительности бывает трудно создать тестовую среду, которая ведет себя так же, как среда эксплуатации. Трудности включают создание реалистичных хранимых данных, прогнозирование вероятных паттернов рабочей нагрузки, обеспечение абсолютно идентичных ресурсов и конфигурации среды и т. д. Если нет уверенности в репрезентативности тестовой среды, нельзя быть уверенными и в полезности измерения.

Модели и реальность

Еще сложнее оценить предиктивную способность, а следовательно, и полезность модели производительности по сравнению с реальной системой. Это значит, что для калибровки и проверки моделей следует всегда проводить тестирование как можно раньше. И не стоит слишком сильно на них полагаться.

Генерация рабочей нагрузки

Даже если известно, какой тип рабочей нагрузки нужно сгенерировать, сама по себе генерация высокоточной, воспроизводимой и полезной рабочей нагрузки — сложная задача. Это особенно актуально для сложных направлений бизнеса, где рабочая нагрузка требует определенных паттернов запросов или пакетной обработки для взаимодействия с определенными паттернами данных.

Спорадические явления

Распределенные системы ведут себя сложно и порой непредсказуемо, что приводит к спорадическим явлениям в производительности, которые трудно воспроизвести. Хотя велик соблазн их проигнорировать, следует воспринимать такие случаи как возможность для исследования и получения новых знаний. Вполне возможно, вы обнаружите, что необходимо скорректировать архитектуру.

Отсутствие логирования

Многие приложения создаются без учета логирования для измерения производительности. Предусмотрите ведение логов на ранних этапах разработки, чтобы в дальнейшем избежать большого объема ретроспективной работы.

Избыточное логирование

Противоположная проблема — слишком подробное логирование. Из-за большого количества логов (значительная часть которых бывает избыточна) объем данных может стать настолько большим, что его будет трудно обработать для надежного измерения. Избежать этого помогают четкая стратегия и настраиваемый механизм логирования.

Масштабируемость

Масштабируемость тесно связана с производительностью, но относится к тому, как система реагирует на увеличение рабочей нагрузки. Эти два качества тесно взаимосвязаны, поскольку первым индикатором проблем с масштабируемостью обычно является снижение производительности, и оба они зависят от эффективности реализации системы.

Ключевой показатель масштабируемости — степень изменения мощности системы по обработке рабочей нагрузки в зависимости от доступных ресурсов (таких, как ЦП, память, хранилище и пропускная способность сети). В идеале система должна масштабироваться линейно, чтобы добавление

50 % ресурсов позволяло ей обрабатывать на 50 % больше рабочей нагрузки. На практике, однако, этого редко удается достичь.

Масштабируемость — это многогранное свойство, которое можно рассматривать с различных точек зрения, включая пропускную способность обработки запросов, пропускную способность пакетной обработки, емкость хранилища данных и организационную (люди и процессы) способность. Все это может влиять на способность системы обрабатывать возрастающие рабочие нагрузки.

Особенности измерения масштабируемости

Масштабируемость также хорошо поддается численному измерению, практическому тестированию и численному моделированию. Охарактеризовать ее с помощью метрик несложно, хотя и здесь есть свои тонкости.

Ключевая метрика масштабируемости — это объем рабочей нагрузки, который система способна обработать при приемлемой производительности и заданном уровне ресурсов. Например, для определенной конфигурации можно измерить количество запросов, которые система обрабатывает за 5 секунд, при среднем времени отклика 0,5 секунды, причем у 95 % запросов время отклика не превышает 2 секунд. Затем можно пошагово увеличивать ресурсы системы (например, на 20 % за раз), снова измерять пропускную способность и использовать соотношение полученных значений, чтобы охарактеризовать масштабируемость системы относительно возросших ресурсов для данной нагрузки.

Как и в случае с производительностью, важным фактором является стоимость. Можно оценить стоимость ресурсов (например, оборудования или облачных сервисов), необходимых для достижения определенной мощности обработки, и то, как она меняется с ростом нагрузки.

Оценка объема хранилища и его стоимости — это еще одна возможность измерить эффективность архитектуры. Метрики здесь довольно просты: объем хранилища, необходимый для выполнения конкретных требований (например, индексирования для повышения производительности или хранения данных в соответствии с требованиями регуляторов), и его стоимость.

Оценить масштабируемость можно также на основе параметров эксплуатации системы. Например, важной метрикой может быть количество сотрудников, которое требуется для поддержки системы при определенных уровнях нагрузки (например, специалисты по бизнес-операциям для обработки исключений или инженеры по эксплуатации для выполнения

рутинных задач). Определив набор метрик для оценки этих факторов масштабируемости, вы сможете охарактеризовать мощность системы в соответствии с требованиями при конкретных уровнях ресурсов. Сравнение мощности при разных уровнях ресурсов помогает понять характеристики масштабируемости под тем углом, под которым их видят стейкхолдеры. А следовательно, оценить, является ли масштабируемость приемлемой или требует внимания к архитектуре. Со временем это также поможет понять, улучшается или ухудшается масштабируемость.

Распространенные проблемы при измерении масштабируемости

Многие из распространенных проблем, касающиеся производительности, относятся и к масштабируемости. Но кроме них есть и другие.

Непредсказуемые узкие места

При исследовании масштабируемости в сложных распределенных системах очень часто обнаруживаются совершенно неожиданные узкие места. Пока они не будут найдены и устранены, они могут значительно снижать достоверность и полезность метрик масштабируемости. Изучайте через тестирование как можно больше характеристик масштабируемости системы, чтобы найти наиболее полезные метрики.

Непредсказуемое и нелинейное поведение

Бывает и другая, смежная проблема — неожиданное нелинейное поведение при масштабировании, например, когда система не справляется с дополнительной нагрузкой так, как ожидалось, даже при увеличении ресурсов. Такое поведение снижает ценность метрик, пока мы не знаем, где оно проявляется. Почти всегда стоит уделить этому вопросу больше времени на этапе проектирования, чтобы таких проблем избежать, и дополнительно протестировать систему, чтобы пристальнее ее изучить.

Комбинирование ресурсов

Для некоторых типов нагрузок увеличение мощности сводится к добавлению одного вида вычислительных ресурсов (например, увеличению мощности ЦП для повышения пропускной способности при интенсивных вычислениях). Однако для большинства нагрузок требуется комбинация ресурсов (например, для масштабирования базы данных обычно нужна не только память и ЦП, но и, вероятно, производительность IOPS). Это требование необходимо учитывать при получении метрик масштабируемости. Например, может понадобиться метрика, которая связывает ресурсы ЦП и памяти, необходимые для обработки веб-запросов.

Доступность

Доступность (availability) — это мера того, какую часть определенного периода времени сервисы системы были доступны для использования. На нее может влиять как плановая недоступность (например, на время техобслуживания системы), так и внеплановая (сбои).

Особенности измерения доступности

Классическая метрика для характеристики доступности — среднее время между сбоями (MTBF, mean time between failures), которая показывает, как часто в системе происходят сбои (надежность). Другая характеристика доступности — время, которое требуется на восстановление после сбоя, обычно называемое средним временем восстановления (MTTR, mean time to recover). Джон Олспо (John Allspaw) в 2010 году сделал знаменитое замечание о том, что время восстановления после инцидентов зачастую важнее, чем частота их возникновения[1], что привлекло к этой метрике большое внимание.

Преимущество MTTR в том, что эту метрику можно учитывать при проектировании, оценивать и тестировать в процессе создания системы, тогда как MTBF можно оценить только после значительного количества сбоев. Часто бывает трудно найти приемлемые параметры для оценки MTBF. На практике нужно моделировать и тестировать вероятное или фактическое значение MTTR в процессе разработки системы, использовать эти значения и приблизительную оценку MTBF для расчета доступности, а затем уточнять ее по мере получения реальных данных.

Еще одна особенность доступности — учет возможной потери данных. Он предполагает поиск компромисса с учетом других целевых метрик, включая MTTR. Для поиска такого компромисса используются метрики целевой точки восстановления (RPO, recovery point objective) и целевого времени восстановления (RTO, recovery time objective). RPO определяет объем данных, который допустимо потерять (обычно измеряется во времени или в количестве транзакций, например «обновления за 10 минут» или «100 транзакций»). RTO определяет промежуток времени, в течение которого данные должны стать доступными после сбоя. Конечно, RTO влияет на MTTR, но это разные понятия. Например, система может восстановить работу сервисов, имея в распоряжении лишь часть данных, из-за чего RTO будет больше, чем MTTR. RPO и RTO обычно находятся в обратной

[1] Олспо Д. «MTTR Is More Important Than MTBF (for Most Types of F)», Kitchen Soap, 7 ноября 2010 г., https://oreil.ly/NFLYz.

зависимости. Если допустимо RPO, равное бесконечности (то есть можно потерять все данные), то обычно можно достичь RTO, близкого к нулю. И наоборот: если ценен каждый байт, RTO может быть значительно больше.

Обе метрики, RPO и RTO, можно моделировать и тестировать во время разработки, а затем измерять их реальные значения после любых инцидентов в среде эксплуатации.

Типичные проблемы при измерении доступности

Рассмотрим несколько частых проблем измерения доступности.

Измерение MTBF

Систему можно спроектировать так, чтобы она соответствовала целевым показателям MTTR, RPO и RTO, оценить эти показатели через количественные модели и получить по ним метрики в ходе тестов. Однако с MTBF дело обстоит сложнее. Эту метрику можно оценить только с помощью моделей надежности (которые не очень полезны для оценки сбоев ПО) или в процессе эксплуатации, когда происходят реальные сбои, — то есть в тех самых ситуациях, которых все стараются избежать. Единственный способ решить эту проблему — собирать информацию о сбоях в других доступных системах (вероятно, в той же организации) и использовать ее в качестве косвенной оценки (прокси) MTBF, пока не будут накоплены собственные данные, полученные в реальных условиях.

Различные режимы сбоев

Как уже упоминалось, такие метрики, как MTTR и RPO, концептуально довольно просты, когда речь идет об одном, атомарном компоненте ПО. Проблема в том, что реальные системы состоят из множества компонентов, формы отказов у которых могут быть совершенно непредсказуемы. Из этого следует, что для любой реальной системы метрики MTTR, MTBF, RPO и RTO необходимо рассматривать все сразу, а затем каким-то образом связывать их в общую картину. Картина эта зависит от конкретной среды, поэтому советую тщательно продумать, как объединить метрики отдельных частей в набор для всей системы.

Зацикленность на «девятках»

Традиционно в индустрии ПО основное внимание уделялось одной метрике доступности систем — *проценту доступности*. Обычно ее измеряют в «девятках»: например, 99,99 %, или «четыре девятки», означает недоступность в течение одной минуты в неделю. Однако такой взгляд на доступность слишком упрощен. На доступность, например, также

влияют такие факторы, как режим и время сбоя, которые эта метрика не учитывает. Кроме того, часто требуемые высокие проценты доступности (например, «пять девяток» — 99,999 %, или одна секунда недоступности в сутки) настолько близки к 100 %, что практически лишены смысла. Избегайте требований, сформулированных в «девятках». Вместо этого используйте сценарии сбоев, чтобы понять, что нужно бизнесу, и определите, как оценивать и получать критически важные метрики доступности для каждой части системы.

Безопасность

Безопасность — это сложное и многогранное свойство. Оно проявляется во многих частях системы, включая инфраструктуру, прикладное ПО и процессы, в которых участвуют люди. Безопасность одновременно и критически важна, и очень трудна для оценки. Одна незначительная ошибка может долгое время оставаться незамеченной, пока некто не обнаружит вызванную ею уязвимость и не решит ею воспользоваться. И внезапно возникнет серьезная проблема безопасности с тяжелыми последствиями.

Тем не менее можно измерить некоторые параметры безопасности, чтобы лучше понять текущую ситуацию и то, на что следует обратить внимание при проектировании архитектуры.

Особенности измерения безопасности

Очевидная метрика безопасности — количество инцидентов, которые происходят в системе. Но не стоит ждать инцидента, чтобы оценить безопасность! Можно выполнять косвенные замеры регулярно на протяжении всей разработки и эксплуатации системы. В частности, следующие:

Статический анализ кода

Позволяет выявить количество потенциальных уязвимостей в коде.

Динамический анализ

Автоматизированное тестирование на проникновение (также называемое динамическим анализом) и ручное тестирование на проникновение позволяют обнаружить и подсчитать уязвимости в системе, развернутой в тестовой среде или среде эксплуатации.

Сканирование инфраструктуры

Тесты сканирования инфраструктуры выявляют потенциальные уязвимости в платформе (опять же, в тестовой среде или среде эксплуатации).

Уязвимости можно взвесить по степени риска, учитывая вероятность их эксплуатации и потенциальные последствия, а затем использовать их агрегированные значения в качестве косвенной метрики безопасности. Конечно, это означает, что измеряются результаты тестовых процессов, а не фактическая безопасность системы. Но такие косвенные метрики — лучший из известных способов оценить уровень безопасности и понять, повышается он или снижается.

Типичные проблемы при измерении безопасности

При измерении безопасности обращайте внимание на следующее.

Согласованность сред

Если вы тестируете безопасность и в тестовых средах, и в средах эксплуатации, убедитесь, что у них одинаковые (или, по крайней мере, максимально близкие друг к другу) конфигурации, чтобы избежать противоречивых результатов.

Учетные данные

Как правило, не рекомендуется использовать одни и те же учетные данные (для пользователей, сертификатов и т. д.) в тестовой среде и среде эксплуатации. Возможно, вам не удастся добиться одинакового уровня согласованности между средами, и в таком случае расхождения следует учитывать при анализе результатов тестирования безопасности.

Ложные срабатывания

Многие инструменты безопасности, особенно для статического сканирования, могут давать большое количество ложных срабатываний. Лучше не включать их в метрики безопасности, поскольку они могут исказить результаты и представить ситуацию в худшем свете, чем она есть на самом деле.

Корректировка с учетом риска

Не все потенциальные проблемы безопасности одинаково серьезны, но бывает трудно определить уровень представляемого ими риска (который, напомним, включает в себя и вероятность, и последствия). Простого ответа здесь нет; консультируйтесь с экспертами, и с опытом ваши оценки станут точнее.

Понимание разумных пределов

Трудно понять, когда тестов достаточно, поскольку невозможно доказать отсутствие чего-либо (например, уязвимостей). Эта проблема особенно

сложна, учитывая, насколько потенциально серьезными могут быть последствия сбоя безопасности и насколько трудно выявлять потенциальные проблемы. Решение заключается в привлечении экспертов для оценки, основываемой на достоверных данных и систематическом, последовательном подходе.

С чего начать

Надеюсь, я убедил вас в важности проведения измерений в цикле доставки ПО и использования полученных результатов для оценки и корректировки работы над программной архитектурой. Следующий шаг — применить эти знания в собственных проектах. Так с чего же начать?

К сожалению, у меня нет единственно верного ответа, поскольку все зависит от конкретной среды, приоритетов и характеристик системы. Соответственно, совет для специалиста, который работает над зрелой корпоративной Java-системой, эксплуатируемой много лет, будет отличаться от рекомендаций для облачного финтех-стартапа, который стремится как можно быстрее вывести на рынок свои первые сервисы.

Однако я могу дать несколько общих полезных советов, которые помогут вам начать применять метрики в проекте.

Начинайте с малого

Всегда есть соблазн начать собирать много данных и измерять все подряд. Такая работа может выглядеть как большой шаг вперед, но на деле она требует много времени и сил, при этом дает мало полезных метрик. Лучше наметить несколько конкретных метрик, на основе которых можно будет принимать решения, и начать с них, чтобы на раннем этапе получить реальный прогресс.

Измеряйте значимые показатели

Значение имеют только те метрики, которые помогают глубже понять систему и направить работу. Разумеется, *проще* всего начать с того, что легко измерить, но правильнее будет определить реально важные для работы метрики и найти способ их получить. Такой подход позволит уже на раннем этапе извлечь ценность из метрик.

Действуйте с учетом полученных результатов

Когда вы с помощью метрик получите представление о системе, вы можете с уверенностью его использовать, чтобы принимать решения — например, чтобы задать приоритеты атрибутам качества, над которыми

следует работать дальше. Это также позволит извлечь ценность из метрик.

Начинайте как можно раньше

Как я уже отмечал, долгое время было принято откладывать измерение на более поздние этапы, когда системы уже некоторое время работают на продакшен. Надеюсь, я убедил вас, что при использовании различных типов метрик можно и запустить систему гораздо раньше, и извлекать из нее ценность в течение всего жизненного цикла ее доставки. Найдите время, чтобы внедрить метрики в самом начале процесса.

Сделайте измерение видимым

Чтобы измерение ценилось и стояло в приоритете, руководство должно понимать, какие именно метрики собираются и с какой целью. Делитесь этой информацией! Регулярно сообщайте о ходе работ над метриками и публикуйте результаты в проектной среде. Такое информирование поможет всем оценить вашу работу и извлечь из нее пользу, а также может вдохновить других принять в ней участие и получать важные для себя метрики.

Сделайте измерение непрерывным

Я рекомендую вам интегрировать механизмы измерения в регулярный цикл доставки ПО, чтобы при создании каждой части системы разработчики думали о том, какие метрики будут полезны и как встроить необходимые для них механизмы.

Следование этим рекомендациям позволит постепенно реализовать измерения при работе над архитектурой и как можно раньше начать извлекать из метрик пользу.

Вымышленный пример

Посмотрим на вымышленном примере, основанном, однако, на реальных событиях, как на практике происходит внедрение в проект механизмов измерения.

Представим новую муниципальную «платформу для взаимодействия с гражданами» Civis в городе с населением 120 000 человек. Организация, реализующая эту платформу, хочет создать единый цифровой интерфейс, через который граждане смогут получать доступ к информации, направлять запросы в местные органы власти и подавать заявки на услуги.

Как у разработчиков платформы, у нас нет уверенности, какие сервисы будут популярны. Мы планируем начать с малого, предложив несколько ключевых сервисов, но нужно быть готовыми быстро добавлять новые (и, возможно, удалять имеющиеся) в зависимости от того, что граждане сочтут полезным. Также неизвестно, сколько пользователей будет с ней взаимодействовать. Иными словами, в этой модели много неизвестных. Разрабатывать такую платформу лучше последовательно, по мере изучения ее работы.

На платформе мы используем проверенные технологии: несколько Java-сервисов в контейнерах на публичном облаке и управляемый сервис облачных реляционных СУБД (в будущем, возможно, и другие облачные сервисы). Сервисы разместим за шлюзом API; также создадим веб-интерфейс и мобильные приложения для Android и iOS.

Начиная проектировать платформу, исходим из того, что для нас будет важен размер базы данных, поскольку от него зависят стоимость, производительность, гибкость и прочие параметры. Мы создаем простую модель в электронной таблице, чтобы оценить количество таблиц в СУБД, ожидаемое количество строк в каждой значимой таблице и необходимый объем хранилища при различных сценариях использования. (Эту таблицу мы сохраним и будем к ней возвращаться.) Эти метрики внутренних артефактов и оценки — наш первый набор. Их цель — помочь понять, нужно ли на этом этапе оптимизировать структуру базы данных.

Приступая к созданию платформы, мы понимаем, что нужно следить за качеством кода и простотой его сопровождения, ведь в будущем платформа может претерпеть значительные изменения. Из этих соображений мы внедряем инструмент статического анализа и с его помощью получаем метрику сложности кода в пайплайне доставки ПО. Интегрируем этот инструмент в фитнес-функцию (на данном этапе это простой Python-скрипт, который прерывает сборку, если сложность становится слишком высокой), которую мы вызываем, когда выполняется одно из действий в пайплайне. Эта метрика внутреннего артефакта показывает, когда пора заняться рефакторингом или переосмыслением архитектуры.

Поскольку это публичная система, которая содержит конфиденциальную информацию, мы приступаем к оценке метрик безопасности. Мы еще не готовы к тестированию на проникновение, но можем провести статический анализ безопасности кода, поэтому добавляем его в пайплайн. Выводим агрегированную метрику «количество уязвимостей», которую сохраняем при каждом запуске пайплайна, чтобы отслеживать тенденцию их появления по мере изменения кода. Эта метрика внутреннего артефакта служит

косвенным показателем состояния безопасности и предупреждает нас о тех участках кода, которым нужно уделить больше внимания. Пока не добавляем для этой метрики автоматизированную фитнес-функцию, поскольку считаем, что какое-то время она будет нестабильна.

Доступность и производительность — ключевые метрики среды эксплуатации, поэтому для основных сценариев использования их нужно оценивать как можно раньше.

Мы создаем набор метрик времени отклика и объединяем их в метрику производительности с автоматизированным набором тестов, которые можно запускать в среде непрерывной интеграции. Будем отслеживать динамику этой метрики. Мы также делаем из нее автоматизированную фитнес-функцию, которая будет предупреждать нас о выходе за допустимые пределы. Эта внешняя операционная метрика предупреждает, когда нужно пересмотреть производительность на уровне системы, чтобы убедиться, что архитектура по-прежнему соответствует требованиям к производительности. Вскоре мы будем использовать эти тесты и в среде эксплуатации.

Пишем тесты для определения времени восстановления, чтобы оценить, сколько времени займет восстановление системы после различных сбоев, и таким образом спрогнозировать уровень ее доступности. Этот набор внутренних и внешних операционных метрик будет предупреждать нас о том, что надо пересмотреть и при необходимости изменить процесс восстановления после сбоев. Чтобы упростить эту задачу, создаем модель в электронной таблице, которая будет служить своего рода ручной фитнес-функцией.

После запуска кода на продакшен сразу же начинаем его измерение. Наконец-то наш код работает в реальных условиях! Мы проводим обычные измерения использования через мобильные приложения и веб-запросы, а также начинаем отслеживать размер базы данных и количество записей в каждой основной таблице, строя графики и сравнивая их с оценочной таблицей, созданной в самом начале. Используем эту внутреннюю операционную метрику, чтобы оценить потенциал роста базы данных и понять, где нужно сосредоточить усилия по оптимизации ее размера.

Со временем на продакшен случаются сбои, поэтому мы вручную ведем статистику времени между сбоями и времени восстановления. Рассчитываем фактическую доступность системы и сообщаем о ней почти в реальном времени стейкхолдерам (включая жителей города). Эта внешняя операционная метрика предупреждает о необходимости позаботиться о повышении доступности, пока ее уровень не стал критически низким.

По мере развития платформы мы продолжаем добавлять (а иногда и удалять) метрики в рамках цикла доставки. Таким образом, мы видим, отвечает ли система ключевым требованиям к качеству, эффективна ли архитектура и на чем сосредоточить внимание, чтобы получить лучший результат.

Итак, мы использовали метрики для повышения эффективности работы над программной архитектурой с самого начала. Мы начали с простого проекта, а затем постепенно добавляли новые метрики, которые помогали нам правильно распределять время и внимание.

Хотя это полностью вымышленный пример, он отражает мой личный опыт применения метрик в работе над программной архитектурой. Возможно, он подскажет вам новые идеи, как применить метрики в ваших проектах.

Подводные камни

В этом разделе я кратко перечислю распространенные ошибки и проблемы, которые возникают при работе с метриками. Знание этих подводных камней поможет вам избежать их в работе.

Упор на механизмы, а не на метрики

Проектирование и внедрение механизмов измерения — сложная и увлекательная задача, и решая ее, есть опасность увязнуть в деталях реализации вместо того, чтобы сосредоточиться на самих метриках. В результате получается отличная инфраструктура для измерения, но самих полезных метрик оказывается мало. Начинайте с малого, с простейших механизмов, и постепенно наращивайте их возможности, когда работа с метриками начнет приносить ценность.

Выбор того, что проще измерить

Некоторые показатели измерить сложно, поэтому возникает понятное желание начать с того, что измерить легко, — например, с размера кода и времени отклика. Конечно, иногда стоит на раннем этапе получить несколько простых метрик, чтобы быстро увидеть первые результаты. Но для достижения значимого эффекта нужно измерять то, что наиболее важно для системы, даже если это сложно.

Упор на технические, а не на бизнес-метрики

Почти всегда проще измерить технический показатель (например, время восстановления базы данных), чем бизнес-показатель (например, суммарный доход в час). Однако если вы сосредоточитесь только на технических метриках, стейкхолдеры со стороны бизнеса вряд ли

поймут или оценят пользу от такой работы. Чтобы правильно расставить приоритеты, учитывайте мнения всех заинтересованных лиц. Предусмотрите не только чисто технические метрики, но и те, что относятся к бизнесу.

Бездействие

Очень часто мы получаем качественные метрики, определяем приоритеты в работе, а затем ничего не предпринимаем — всегда найдутся более неотложные задачи. Однако чтобы работа с метриками приносила пользу, нужно применять ее результаты на практике. Поставьте в приоритет те действия, которые требуется предпринять исходя из полученных результатов. Например, можно выделять в каждом спринте определенный процент времени на *оптимизацию* или на выполнение действий по результатам, полученным при анализе метрик.

Точность в ущерб пользе

Будучи инженерами и исследователями, мы стремимся повышать точность метрик. Такое стремление похвально, но обычно наступает момент, когда дальнейшее повышение точности уже не влияет на качество принимаемых решений. Важно не упустить этот момент, чтобы не потратить время и силы впустую.

Слишком много метрик

С этим связан и другой вопрос: как много всего нужно измерять? Современные платформы позволяют с приемлемыми затратами получать огромное количество метрик. Принцип «начинать с малого» как раз и помогает со временем не превратить работу с метриками в монстра, который генерирует огромные объемы данных и десятки бесполезных показателей. Периодически пересматривайте набор метрик и задавайтесь вопросом, все ли они по-прежнему полезны и от каких можно отказаться.

Если избегать этих распространенных ошибок, то можно сделать работу с метриками по-настоящему полезной и стабильной.

Заключение

В этой главе мы рассмотрели, почему измерения — важнейшая часть работы над программной архитектурой. Вы узнали, что:

- Получение метрик по атрибутам качества системы — один из немногих способов понять, эффективна ли работа над архитектурой.

- Метрики позволяют понять, на что в архитектуре нужно обратить больше внимания и как расставить приоритеты в работе, помогая таким образом делать сложный, но рациональный выбор между разными видами архитектурных задач.
- Работа с метриками также заставляет искать способы сделать атрибуты качества измеримыми и конкретизировать требования. (А иногда четко показывает, что это невозможно.) Такая конкретика помогает формулировать требования к атрибутам качества и улучшает взаимодействие со стейкхолдерами.

Когда вы поймете, какими атрибутами качества должна обладать система, вы сможете определить механизмы для получения метрик этих качеств и спроектировать соответствующие реализации.

Как вы уже знаете, существуют разные типы метрик. Одни относятся к самой системе, другие — к артефактам, которые используются при ее создании. Некоторые метрики измеряют внешние проявления атрибутов качества, а другие — внутренние параметры системы. Определив, какие метрики вы хотите получать, классифицируйте их, чтобы понимать ценность и ограничения каждой.

Когда механизмы для получения метрик будут готовы, можно начинать передавать метрики в цикл непрерывной архитектуры и сравнивать их с требованиями (по возможности с помощью фитнес-функций). Этот цикл обратной связи выявляет те области системы, которые не соответствуют требованиям и нуждаются во внимании со стороны архитектора. Такой подход к работе с метриками позволит вам четко объяснять ценность различных архитектурных задач и придаст уверенности в том, что вы работаете над самым важным и полезным.

ГЛАВА 8

От метрик к инженерным практикам

Нил Форд

Мой студенческий путь был извилист: я не раз менял специальности. Несколько лет я усердно учился на инженера-механика. Проучившись два года и получив специализацию по физике, я поступил в крупный технический университет неподалеку. Через год я решил перейти на факультет computer science (наконец-то я стал ходить на занятия, которые приносили удовольствие, а не требовали терпения) и навсегда покинул мир инженерной физики.

Однако я достаточно долго изучал этот предмет, чтобы оценить разницу между фундаментальной математикой в физике и тем, как она становится реальной прикладной дисциплиной — механикой. Математика дает инструменты измерения, но инженеры не могли использовать эти инструменты для созидания, пока не поняли, как именно математика отражает реальный мир.

У архитекторов и разработчиков такое же отношение к метрикам, как у инженеров к физике: метрика служит мерой, но оценка этой меры в полезном контексте превращает метрики в инженерные практики. Архитекторы и разработчики десятилетиями использовали метрики для проверки отдельных частей архитектуры, но зачастую делали это бессистемно. Нам нужен последовательный подход к использованию метрик, который лежал бы в основе инженерии. Хотя программная разработка еще далека от уровня развития инженерной физики, мы учимся превращать метрики в инженерные практики.

Путь к фитнес-функциям

В книге «Building Evolutionary Architectures» («Эволюционная архитектура»), которую я написал в соавторстве с Патриком Куа и Ребеккой Парсонс, мы ввели понятие архитектурной фитнес-функции. У Ребекки был

опыт проектирования генетических алгоритмов — это алгоритмы, которые выдают результат, затем мутируют, выдают новый результат и так далее до выполнения условия остановки. Например, один из методов мутации известен как «рулетка»: если алгоритм использует одну или несколько констант, этот метод выбирает новое значение случайным образом, как в рулетке.

При проектировании такого алгоритма разработчик может скорректировать метод мутации, например, если более низкие или отрицательные значения дают более желаемые результаты. Для этих целей разработчики используют механизм под названием *фитнес-функция* — это целевая функция, которая помогает определить соответствие дизайна требованиям.

В книге «Эволюционная архитектура» мы объединили концепцию управления программной архитектурой и фитнес-функции, чтобы дать архитектурным фитнес-функциям следующее определение: «Архитектурная фитнес-функция — это механизм, который обеспечивает объективную оценку целостности выбранной характеристики (характеристик) архитектуры»[1].

В этом определении есть несколько важных терминов. Рассмотрим их в обратном порядке появления:

Характеристика(-и) архитектуры

Архитекторы могут разделить структурную часть проекта на *характеристики домена*, или *предметной области,* и *характеристики архитектуры*. *Домен* (domain) — это движущая сила программной разработки, область решаемых продуктом проблем. *Характеристики архитектуры* (также известные как «нефункциональные требования», «атрибуты качества системы» и т. д.) — это свойства, не связанные с доменом: производительность, масштабируемость, эластичность, доступность и многие другие. Фитнес-функции в первую очередь относятся к характеристикам архитектуры, поскольку для тестирования домена у нас уже есть зрелые инструменты: модульные, функциональные, приемочные тесты и т. д. Однако до сих пор проверка характеристик архитектуры была ситуативной и распределялась между проверками на этапе сборки, мониторингом продакшена, логированием для расследования инцидентов и множеством других инструментов. Фитнес-функции объединяют эти проверки и позволяют систематизировать работу с тем, что всегда было связано между собой (проверка характеристик архитектуры), но не обрабатывалось единообразно.

[1] Форд Н., Парсонс Р., Куа П. «Эволюционная архитектура. Автоматизированное управление программным обеспечением». Глава 2 «Фитнес-функции», с. 35.

Объективные критерии оценки

Существует множество определений характеристик архитектуры, и в отрасли так и не удалось их стандартизировать из-за быстрых изменений в экосистеме программной разработки. Например, метрики производительности для веб-приложений не подходят для мобильных приложений — с изменением экосистемы меняются и типы измеряемых показателей. Однако какой бы ни была характеристика, для ее проверки архитекторы должны иметь возможность объективно ее измерить.

Некоторые характеристики архитектуры слишком обширны, например *надежность*, которая может включать в себя доступность, целостность данных и многое другое. Такие характеристики называют *составными*, так как они состоят из других, объективно измеримых значений. Таким образом, если архитектор не может определить, как что-то измерить, возможно, это составная характеристика, которую нужно декомпозировать.

Любой механизм

Разработчики привыкли к тому, что для каждой платформы есть единый набор инструментов тестирования. Например, для платформы Java существует ряд привязанных к ней тестовых фреймворков. Однако архитектура выходит за рамки одной платформы и учитывает множество вариантов поведения системы. Поэтому для реализации фитнес-функций в проекте архитекторы и разработчики должны использовать разнообразные инструменты: библиотеки для тестирования, средства мониторинга производительности, средства хаос-инжиниринга и т. д.

Архитекторам следует шире смотреть на то, что является проверкой, не ограничиваясь инструментами тестирования домена (рис. 8.1).

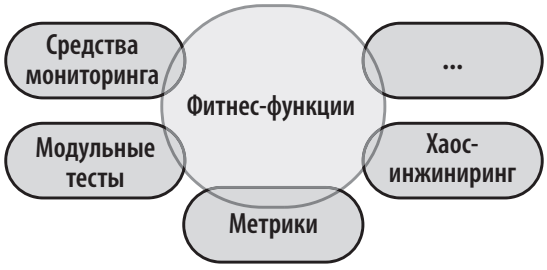


Рис. 8.1. Фитнес-функции задействуют разнообразные инструменты и механизмы

Как видно на рис. 8.1, фитнес-функции пересекаются с модульным тестированием. И те и другие используют метрики на уровне кода в модульных тестах и специализированных библиотеках, инструменты оценки метрик, такие как SonarQube (https://www.sonarqube.org), средства мониторинга для операционных архитектурных характеристик, комплексные фреймворки для стресс-тестирования, например Simian Army от Netflix (https://oreil.ly/08qk4), и многие другие.

Фитнес-функции — это единый термин для множества способов, которыми архитекторы проверяют компоненты архитектуры, но только автоматизация превращает эти способы в инженерию.

От метрик к инженерным практикам

Как метрики становятся фитнес-функциями? Это происходит благодаря регулярному их применению, предпочтительно автоматическому, при каждом изменении кода, по аналогии с модульным и другими видами тестирования в конкретной предметной области.

Многие команды встраивают в процесс сборки такие инструменты, как уже упомянутый SonarQube, чтобы создавать дашборды и получать другие метрики качества кода. На самом деле в нашей книге вы найдете множество отличных потенциальных средств проверки архитектуры. Однако если команда не идет дальше и не запускает измерения регулярно, с заранее установленными объективными пороговыми значениями, то полученные данные становятся лишь констатацией факта, а не упреждающим инструментом.

Вот пример: проверка циклических зависимостей между компонентами. Это распространенная метрика на уровне кода, применимая практически ко всем платформам. Рассмотрим три компонента на рис. 8.2.

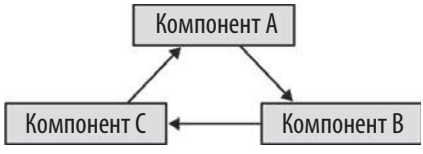


Рис. 8.2. Три компонента с циклической зависимостью

Циклическая зависимость, показанная на рис. 8.2, считается антипаттерном, поскольку затрудняет повторное использование одного из компонентов — приходится тянуть за ним и все остальные связанные с ним компоненты.

Из этих соображений архитекторы обычно стремятся свести количество циклических зависимостей к минимуму. Однако сама среда разработки с ее привычными инструментами активно мешает архитектору решить эту задачу. Что происходит, когда в современной IDE разработчик ссылается на класс, пространство имен или пакет которого еще не импортирован? IDE предлагает диалоговое окно для автоматического импорта нужного пакета.

Разработчики настолько привыкли к этой удобной функции, что принимают предложение рефлекторно, не вникая в суть. В большинстве случаев автоимпорт — это очень удобно и не вызывает проблем. Но иногда он все же создает циклическую зависимость между компонентами. Как архитекторам этого избежать?

Рассмотрим набор пакетов, проиллюстрированный на рис. 8.3.

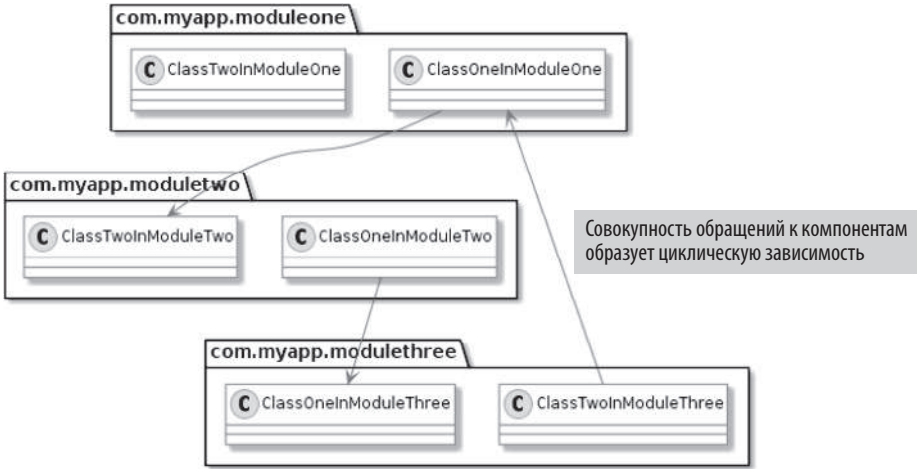


Рис. 8.3. Циклические зависимости в виде пакетов Java

ArchUnitArchUnit (https://www.archunit.org) — это инструмент для тестирования, созданный по мотивам JUnit (и использующий некоторые его возможности), но предназначенный для проверки различных характеристик архитектуры, включая поиск циклических зависимостей в заданной области, как показано в примере 8.1.

Пример 8.1. ArchUnit позволяет обнаруживать циклические зависимости между компонентами

```
public class CycleTest {
    @Test
         public void test_for_cycles() {
```

```
        slices().
          matching(«com.myapp.(*)..»).
          should().beFreeOfCycles()
}
```

Тест из примера 8.1 — это типичная метрика, которая используется во многих инструментах. Как же она превращается в инженерную практику? Благодаря постоянному автоматизированному применению, как показано на рис. 8.4.

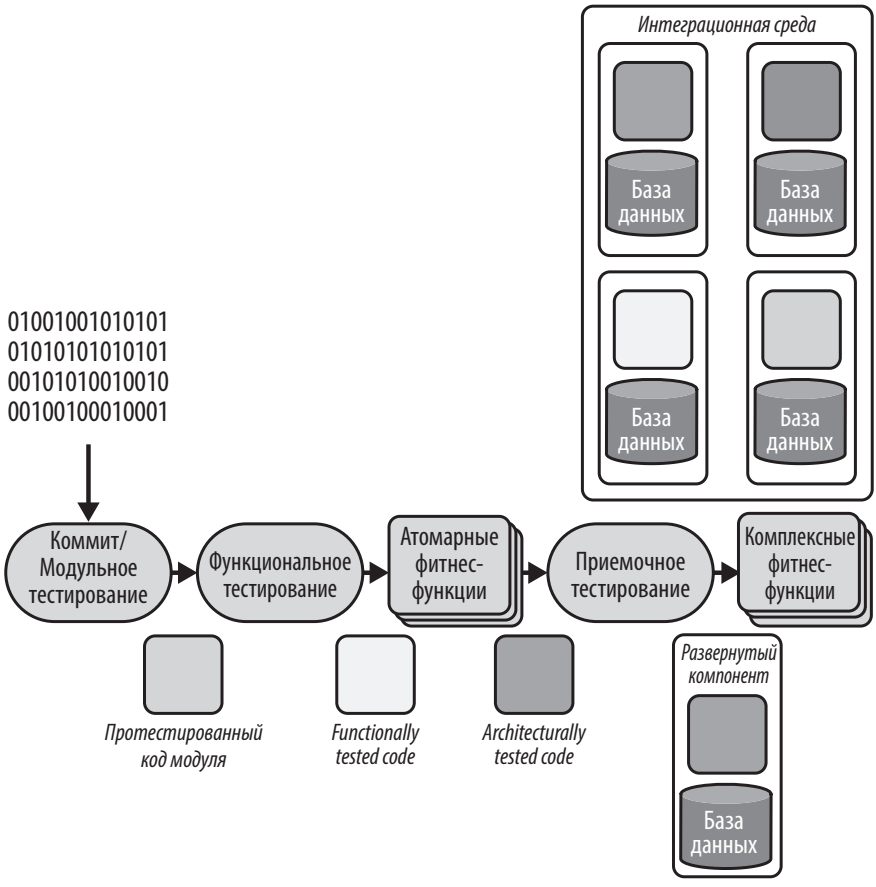


Рис. 8.4. Добавление фитнес-функций в пайплайны непрерывной интеграции и развертывания

Как видно на рис. 8.4, фитнес-функции применяются наряду с уже существующими в проектах механизмами проверки, такими как модульное, функциональное и приемочное тестирование. Постоянный запуск фитнес-

функций гарантирует, что архитекторы максимально быстро выявляют нарушения установленных правил.

Автоматизация превращает метрики в рабочий инструмент

В начале 1990-х Кент Бек (Kent Beck) возглавлял группу разработчиков, которая открыла одну из движущих сил прогресса в программной разработке в последующие три десятилетия. Группа состояла из дальновидных специалистов, работавших над проектом C3. Члены команды хорошо разбирались в современных на тот момент процессах разработки, но не были от них в восторге: казалось, ни один из популярных в то время процессов не приносил устойчивого успеха. Так Кент Бек заложил основы *экстремального программирования* (XP, eXtreme programming): команда брала на вооружение хорошо знакомые им практики и доводила их до экстремального уровня. Например, их коллективный опыт подсказывал, что проекты с более высоким покрытием кода тестами, как правило, отличаются и более высоким качеством самого кода.

Они пришли к стратегии *разработки через тестирование* (TDD, test-driven development), которая гарантирует, что весь код будет протестирован, поскольку тесты пишутся раньше кода.

Одно из их ключевых наблюдений касалось интеграции. В то время в большинстве проектов программной разработки была предусмотрена отдельная фаза интеграции. Предполагалось, что разработчики будут неделями или даже месяцами писать код изолированно, а затем объединять свои изменения на этапе интеграции проекта. Более того, многие популярные в то время системы контроля версий (например, ClearCase) принудительно изолировали работу каждого разработчика. Эта практика основывалась на производственном опыте, который часто переносился на программную разработку. Однако XP-разработчики на основе прошлых проектов заметили, что чем чаще происходит интеграция, тем меньше возникает проблем. Это привело их к концепции *непрерывной интеграции*: каждый разработчик фиксирует изменения в основной ветке разработки не реже одного раза в день.

Непрерывная интеграция и многие другие практики XP показали широкие возможности автоматизации и инкрементальных изменений. Команды, использующие непрерывную интеграцию, не только тратят меньше времени на регулярное слияние кода, но и в целом экономят время. Когда команды применяют непрерывную интеграцию, конфликты слияния выявляются

и разрешаются по мере их возникновения, то есть как минимум ежедневно. Если же в проектах используется заключительная фаза интеграции, то это позволяет множеству конфликтов слияния накопиться и превратиться в «ком грязи», который приходится распутывать в конце проекта.

Автоматизация важна не только для интеграции; это оптимизирующая сила для всей инженерии. До появления непрерывной интеграции разработчикам приходилось снова и снова вручную выполнять интеграцию и слияние кода; непрерывная интеграция (и задаваемый ею ритм) избавила их от большей части этой боли.

Мы заново открыли для себя преимущества автоматизации в начале 2000-х, во время революции DevOps. Инженеры по эксплуатации носились по датацентрам, вручную устанавливая операционные системы, накатывая патчи и выполняя другие рутинные задачи, из-за чего важные проблемы часто упускались из виду. С появлением инструментов автоматического развертывания инфраструктуры, таких как Puppet и Chef, команды получили возможность автоматизировать управление инфраструктурой и обеспечивать ее согласованность.

В книге «Эволюционная архитектура» мы описали то же самое: архитекторы пытались контролировать соблюдение правил с помощью код-ревью, архитектурных комитетов и других ручных бюрократических процессов. Привязывая фитнес-функции к процессу непрерывной интеграции, архитекторы превращают метрики и другие проверки в регулярную процедуру контроля целостности системы.

Фитнес-функция для проверки циклических зависимостей из примера 8.1 наглядно доказывает преимущество автоматизации контроля. Как еще архитектор может предотвратить появление циклических зависимостей? Стоит учитывать, что код-ревью и другие проверки, выполняемые вручную, требуют человеческого вмешательства и затормаживают контроль. И наоборот, внедрение автоматизированного тестирования через фитнес-функции не дает проблемному коду попасть в репозиторий и не требует от архитекторов бдительности выше человеческих возможностей.

Кейс: связанность

Разработчикам следует искать фреймворки фитнес-функций под конкретные платформы, но не стоит отчаиваться, если подходящего инструмента не нашлось. Рассмотрим пример теста структурных метрик: сначала воспользуемся имеющимся инструментом, а затем, при необходимости, создадим собственный.

Внутренняя структура компонентов — это часть архитектуры, для которой полезен контроль с помощью метрик. Рассмотрим распространенный архитектурный стиль — многоуровневую архитектуру, которая показана на рис. 8.5.

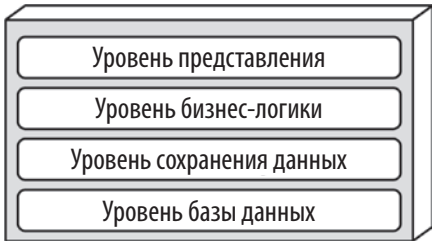


Рис. 8.5. Топология традиционной многоуровневой архитектуры

Архитектор проектирует многоуровневую архитектуру, как на рис. 8.5, чтобы обеспечить разделение ответственности. Но как архитектору быть уверенным, что команды разработчиков реализуют ее правильно? Команды могут не осознавать важность изоляции, или же архитектор работает в организации, где предпочитают сначала сделать и попросить прощения, чем не сделав просить разрешения.

В любом случае архитекторы могут быть уверены в правильной реализации разработанной архитектуры, если определят для этой топологии структурные тесты с помощью ArchUnit. Рассмотрим структуру пакетов, показанную на рис. 8.6.

Архитектор может определить правила управления, которые сохранят архитектурную структуру с рис. 8.6, в виде модульного теста, включив в него следующий код из фреймворка ArchUnit.

Используя код из примера 8.2 и мэтчеры Hamcrest из фреймворка ArchUnit, архитектор может создать компактный модульный тест, который практически на обычном английском языке определяет отношения между слоями и предотвращает нежелательную связанность.

Пример 8.2. Модульный тест ArchUnit для проверки многоуровневой архитектуры

```
layeredArchitecture()
  .layer(«Controller»).definedBy(«..controller..»)
  .layer(«Service»).definedBy(«..service..»)
  .layer(«Persistence»).definedBy(«..persistence..»)

  .whereLayer(«Controller»).mayNotBeAccessedByAnyLayer()
  .whereLayer(«Service»).mayNotBeAccessedByAnyLayer(«Controller»)
  .whereLayer(«Persistence»).mayNotBeAccessedByAnyLayer(«Service»)
```

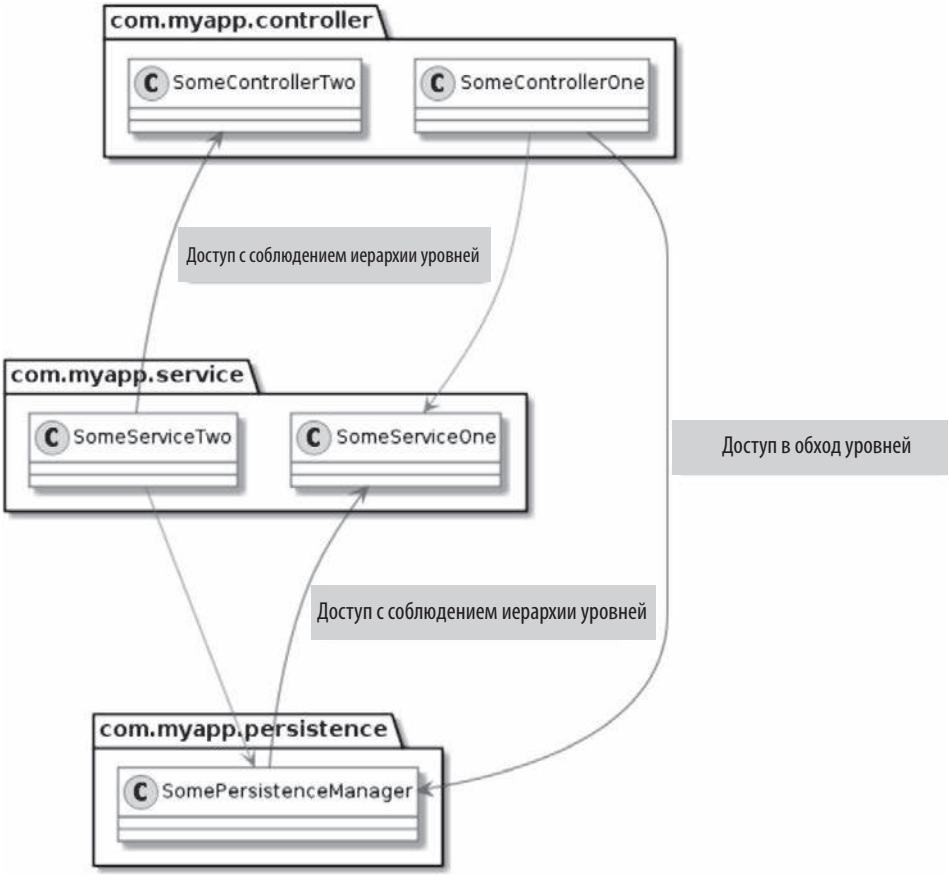


Рис. 8.6. Структура пакетов, которая иллюстрирует многоуровневую архитектуру

Инструмент наподобие ArchUnit чрезвычайно удобен, но доступен только для команд, которые работают на платформе Java, и только для проверки на этапе компиляции. Аналогичный инструмент, NetArchTest (https://oreil.ly/CUako), доступен для платформы .NET. Но как быть с другими платформами? И что еще важнее, как выполнять такие же проверки для распределенных архитектур, например микросервисных?

Часто для архитектуры приходится создавать самописные решения. Большинство архитектур не являются типовыми, а представляют собой смесь хороших и плохих, старых и новых, собственных и навязанных инструментов, а также фреймворков, пакетов и т. д. Однако с помощью стандартных инструментов архитекторы могут создавать простые фитнес-функции, которые выполняют ту же задачу.

Например, рассмотрим топологию микросервисов, показанную на рис. 8.7.

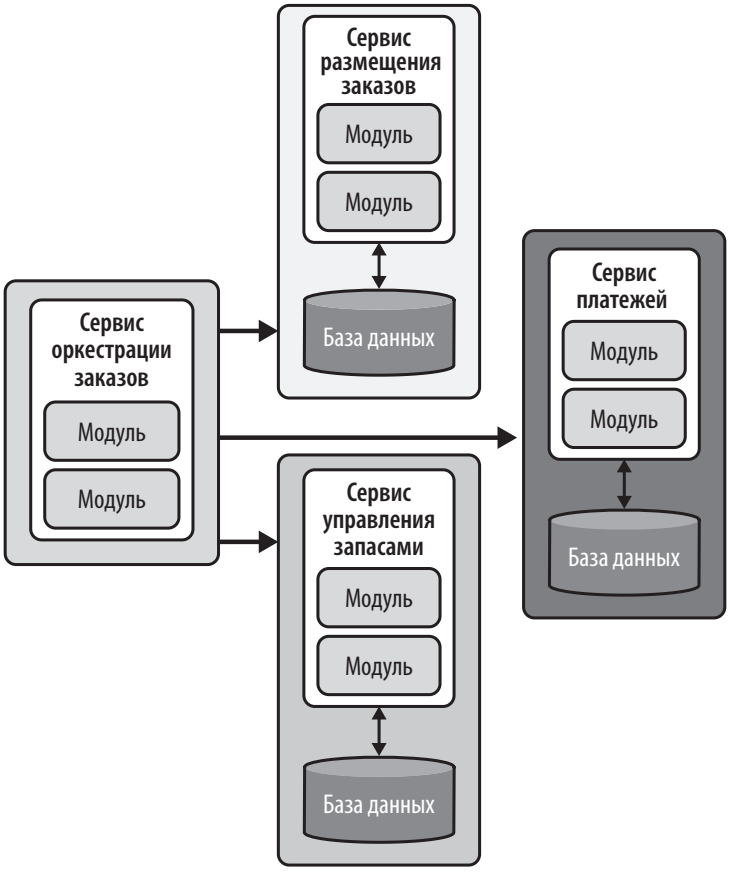


Рис. 8.7. Оркестрация рабочего процесса в микросервисной архитектуре

На рис. 8.7 сервис слева выступает в роли оркестратора, который координирует рабочие процессы трех доменных сервисов, расположенных справа. Задача архитектора — убедиться, что доменные сервисы взаимодействуют только с оркестратором, но не друг с другом.

Хотя эта задача похожа на задачу с топологией слоев (см. рис. 8.6), единого инструмента для контроля над ней не существует, поскольку создать его практически невозможно из-за множества вариаций в распределенных архитектурах. Это отличный пример ситуации, когда инструмент необходим, но архитекторам приходится его собирать самим, используя возможности доступного им стека технологий.

Архитектор может реализовать фитнес-функцию для проверки взаимодействия с оркестратором, как показано в псевдокоде в примере 8.3.

Пример 8.3. Фитнес-функция на псевдокоде для проверки допустимых взаимодействий с оркестратором

```
def ensure_domain_services_communicate_only_with_orchestrator
  list_of_services = List.new()
                        .add("Сервис оркестрации заказов")
                        .add("Сервис размещения заказов")
                        .add("Сервис платежей")
                        .add("Сервис управления запасами")
  list_of_services.each { |service|
    service.import_logsFor(24.hours)
    calls_from(service).each { |call|
      unless call.destination.equals("Сервис оркестрации заказов")
         raise FitnessFunctionFailure.new()
    }
  }
end
```

В примере 8.3 архитектор определяет список сервисов и требуемые правила взаимодействия. Однако, в отличие от ArchUnit, не существует фреймворка для проверки этих правил в конкретной архитектуре. Следовательно, архитектору приходится писать код, который соберет необходимую информацию для реализации такой проверки. В этом примере мы исходим из того, что каждый сервис ведет журнал всех исходящих вызовов за определенный промежуток времени. Тело фитнес-функции загружает журналы каждого сервиса за последние 24 часа, а затем анализирует их, чтобы определить адресата вызова. Если адресат не соответствует правилам, код генерирует исключение, чтобы сообщить об ошибке.

Фитнес-функцию из примера 8.3 можно реализовать по-разному. Например, если команда использует средства мониторинга, а не логи для расследования инцидентов, фитнес-функция будет получать информацию о вызовах сервисов в реальном времени и использовать обработчики событий для выявления некорректных вызовов и отправки оповещений.

Архитекторам не стоит отчаиваться, если не удается сразу найти готовый инструмент для реализации фитнес-функции. Многие инструменты в современной экосистеме разработки позволяют создавать небольшие специализированные фитнес-функции, которые объединяют их выходные данные.

Кейс: проверка безопасности на наличие уязвимостей нулевого дня

Как правило, архитекторы воспринимают метрики как низкоуровневые скрипты оценки, похожие на многочисленные уже имеющиеся. Однако в сочетании с фитнес-функциями их область применения может быть настолько широкой, насколько это необходимо организации.

7 сентября 2017 года Equifax, крупное американское бюро кредитных историй, объявило об утечке данных. Причиной стало появление эксплойта для популярного веб-фреймворка Struts из экосистемы Java (Apache Struts vCVE-2017-5638). В Apache Foundation опубликовали заявление об уязвимости и выпустили патч еще 7 марта 2017 года. На следующий день Министерство внутренней безопасности США (Department of Homeland Security) связалось с Equifax и другими подобными компаниями, чтобы предупредить их о проблеме. 15 марта 2017 года они провели сканирование и обнаружили большинство уязвимых систем... но *не все*. В результате на многие устаревшие системы критически важный патч не устанавливали вплоть до 29 июля 2017 года, то есть до того дня, когда эксперты по безопасности Equifax выявили злонамеренные действия, из-за которых и произошла дальнейшая утечка данных в сентябре 2017 года.

А ведь все могло бы быть иначе. У каждого проекта (даже у простаивающего) мог быть свой пайплайн развертывания, а у команды безопасности — «слот» в пайплайне каждой команды, куда можно было бы развертывать фитнес-функции (рис. 8.8).

В большинстве случаев на этапе обеспечения безопасности, показанном на рис. 8.8, могли выполняться рутинные проверки, например на предмет того, что разработчики не хранят пароли в базах данных, и решаться другие подобные стандартные задачи управления. При появлении эксплойта нулевого дня этот же механизм мог бы позволить команде безопасности встроить в каждый проект тест на наличие определенного фреймворка и его версии и запустить сборку проекта. При обнаружении опасной версии тест мог бы останавливать сборку и уведомлять команду безопасности. И если бы в Equifax архитекторы во всех проектах, даже неактивных, использовали пайплайн развертывания, то это позволило бы автоматизировать управление. Команды, в свою очередь, могли бы настроить пайплайны развертывания так, чтобы они запускались при любом изменении внутри экосистемы: в коде, в схеме базы данных, в файле конфигурации развертывания или фитнес-функциях. Такой подход позволил бы компаниям повсеместно автоматизировать важные задачи управления в гораздо более широком масштабе, чем принято думать применительно к метрикам.

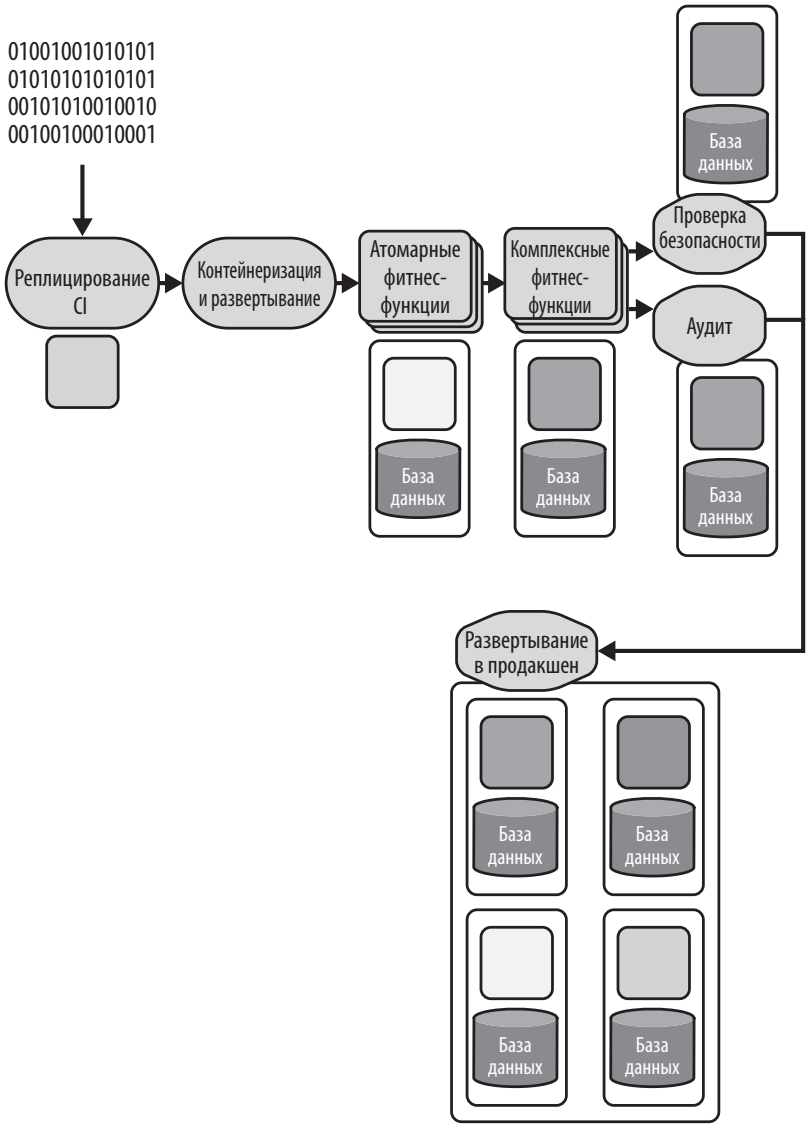


Рис. 8.8. Пайплайн развертывания, включающий этап управления безопасностью

Фитнес-функции дают архитекторам множество преимуществ, и не последнее из них — возможность снова кодить! Архитекторы часто жалуются, что им почти не удается писать код, а ведь фитнес-функции — это и есть код. Чтобы создать для архитектуры исполняемую спецификацию, которую любой желающий, запустив сборку проекта, может проверить в любое

время, архитекторы должны хорошо понимать систему и направление ее развития. И это требование совпадает с основной задачей — следить за кодом по мере роста проекта.

Кейс: фитнес-функции для проверки точности воспроизведения систем

Фитнес-функция для проверки точности воспроизведения систем, которая позволяет командам сравнивать старую и новую реализации, демонстрирует весь потенциал возможностей сочетания метрик и инженерных практик. Многие архитекторы сталкиваются с одной и той же сложной задачей: как заменить старую систему новой и при этом убедиться, что новая система будет работать с точно таким же результатом, что и старая? Иными словами, как архитектору гарантировать точность воспроизведения реализации системы? Ответ — использовать соответствующую фитнес-функцию.

Рассмотрим один пример из блога GitHub, опубликованный в посте под названием «Move Fast and Fix Things»(«Сделай быстро и исправь»)[1]. В этом примере оказались взаимосвязанными многие аспекты из определения эволюционной архитектуры. Кроме того, этот пример опроверг распространенный тезис о том, что агрессивные Agile-практики, такие как непрерывное развертывание, повышают риск до неприемлемого уровня. На самом же деле у команд, применяющих эти практики, в арсенале есть способы снизить эти риски.

Отмечу, что в GitHub придерживаются довольно агрессивных инженерных практик. Там, в частности, используется непрерывное развертывание: изменения, которые разработчики вносят в код, сразу проходят через пайплайн развертывания и, если ошибок нет, попадают на продакшен. В GitHub в среднем выполняется 60 развертываний в день, но масштабы работы компании таковы, что пограничные случаи возникают практически мгновенно.

Как описано в посте, проблема, за которую взялись в GitHub, касалась слияния файлов. Раньше для этого запускали shell-скрипт, который использовал Git из командной строки. Способ работал безупречно, но плохо масштабировался. В блоге как раз и рассказывается о том, как его меняли для повышения производительности.

---

[1] Висент Марти (Vicent Martí), «Move Fast and Fix Things», блог GitHub, 15 декабря 2015 г., https://oreil.ly/JXuEu.

Команда разработала новую функцию для выполнения слияния в оперативной памяти и протестировала ее поведение, чтобы убедиться в корректности работы. Однако в какой-то момент ее нужно было развернуть на продакшен, и вот тут начинается самое страшное: что, если что-то пойдет не так? Если код слияния просто перестанет работать — это полбеды. Но что, если в старом коде обнаружится такая ранее не известная точка привязки, которая приведет к еще более катастрофическим последствиям? И вот этот страх мешает многим командам внедрять современные решения.

Команда GitHub создала (и сделала его код открытым) инструмент Scientist (https://oreil.ly/0AF3j), который дает возможность безопасно проводить эксперименты в архитектуре, не подвергая пользователей риску столкнуться с ошибками.

Инструмент Scientist позволяет разрабатывать эксперименты, каждый из которых содержит два блока: `use` и `try`. Блок `use` содержит старый код, который подлежит замене, а блок `try` — новую реализацию. В эксперименте со слиянием логика инкапсулирована в методе `create_merge_commit`, как показано в примере 8.4.

Пример 8.4. Метод commit с блоком science

```
def create_merge_commit(author, base, head, options = {})
  commit_message = options[:commit_message] || "Выполнить слияние #{head}
c #{base}"
  now = Time.current

  science «create_merge_commit» do |e|
    e.context :base => base.to_s, :head => head.to_s, :repo => repository.nwo
    e.use { create_merge_commit_git(author, now, base, head, commit_message) }
    e.try { create_merge_commit_rugged(author, now, base, head, commit_message)
}
  end
end
```

В примере 8.4 блок `science` выступает в роли диспетчера для двух блоков, `use` и `try`. При каждом вызове блок `use` выполняется всегда, и именно его результат возвращается пользователю. Таким образом, пользователь даже не подозревает, что участвует в эксперименте. Архитектор также настраивает фреймворк, определяя, как часто будет выполняться блок `try` — в эксперименте со слиянием он запускался в 1 % запросов. Когда выполняются оба блока, `use` и `try`, фреймворк:

- в случайном порядке запускает `use` и `try`, чтобы предотвратить аномалии, связанные с временем выполнения;

- сравнивает результаты обоих вызовов на предмет соответствия;
- перехватывает, но регистрирует любые исключения, которые возникают в блоке `try`;
- выводит результаты на дашборд (рис. 8.9).

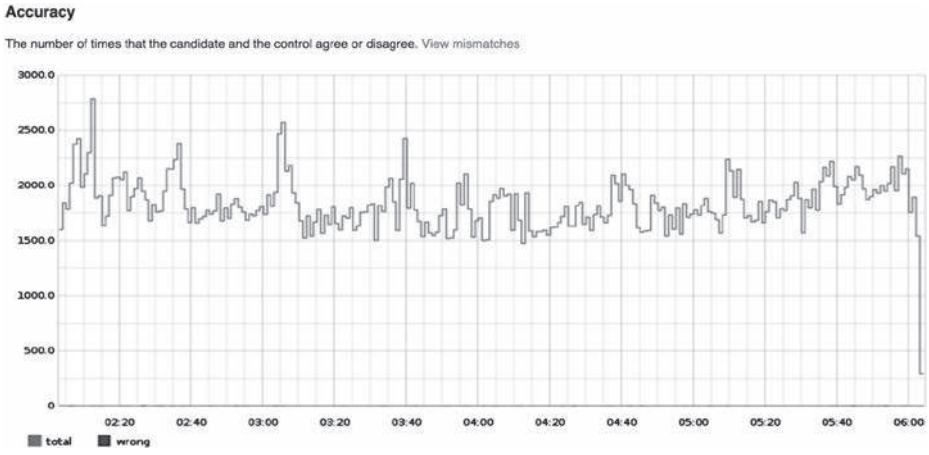


Рис. 8.9. Дашборд с результатами эксперимента по слиянию за несколько часов (надпись над графиком: «Количество случаев, когда кандидатная система и контрольная система совпадают или расходятся в результатах. Просмотреть расхождения»)

Дашборд на рис. 8.9 показывает, что в 02:20 GitHub выполнил чуть более двух тысяч слияний. Однако из-за масштабов, в условиях которых работает GitHub, это не очень наглядное представление ошибок. На рис. 8.10 показаны только ошибки за тот же период.

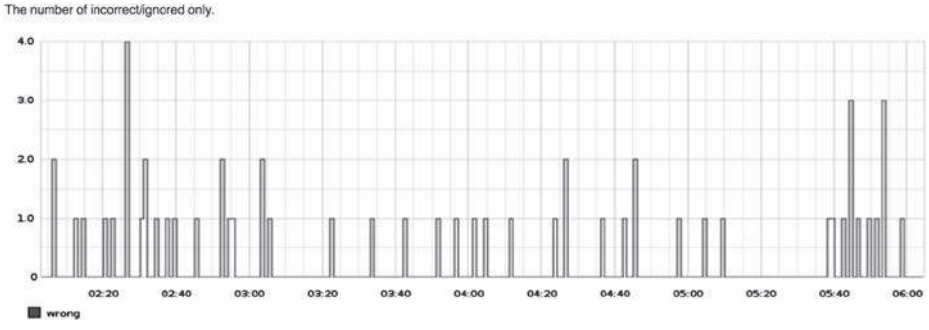


Рис. 8.10. Ошибки за тот же период, что и на графике на рис. 8.9 (надпись над графиком: «Только ложные/проигнорированные»)

Как видно на рис. 8.10, в новом коде были ошибки. Однако благодаря фреймворку Scientist пользователи этих ошибок не видят. Вместо этого разработчики исправляют проблему и повторно развертывают код — как вы помните, в этот период непрерывное развертывание продолжалось как для этого эксперимента, так и для другого кода.

Одной из целей эксперимента было повышение производительности, и на графике на рис. 8.11 уже виден успех.

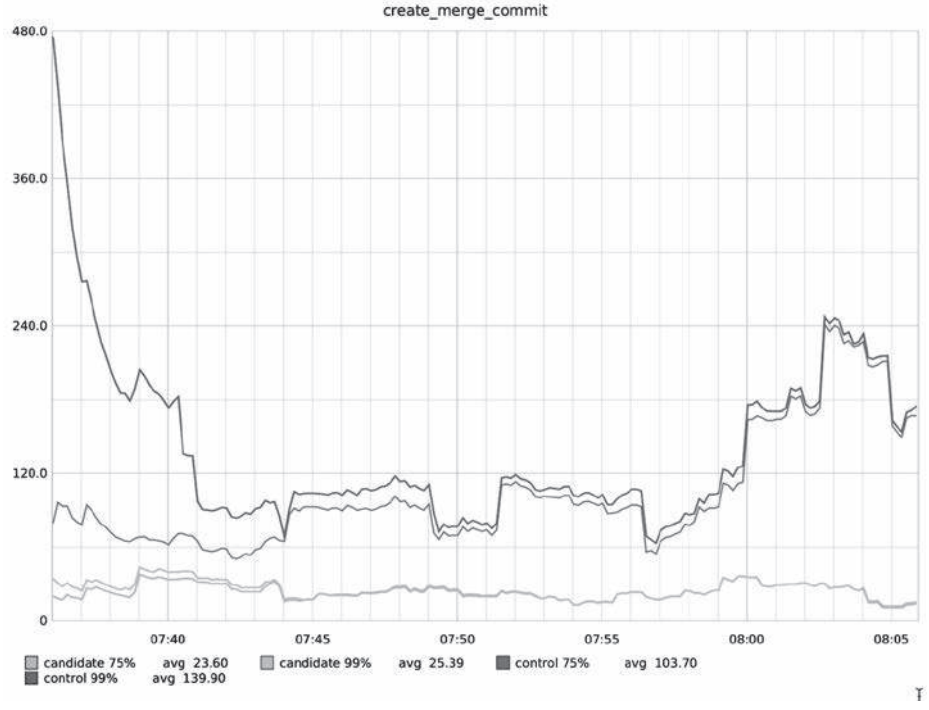


Рис. 8.11. Метрики производительности во время эксперимента, который проводился под управлением Scientist (candidate — кандидатная система, control — контрольная система, avg — среднее)

У архитекторов GitHub этот эксперимент занял четыре дня, пока они не добились того, что за 24 часа непрерывной работы не было выявлено ни одного расхождения или замедления работы функции. После этого они удалили старый код слияния и оставили новый. За эти четыре дня эксперимент запускался более 10 миллионов раз, поэтому разработчики были твердо уверены в корректной работе нового кода.

Scientist — это фитнес-функция для проверки точности воспроизведения систем, реализованная с помощью переключателей функций и метрик производительности. Этот подход показывает, как благодаря синергии между инженерной практикой и метриками возникают исключительные возможности.

Заключение

Как бы ни были широки возможности фитнес-функций, ими не следует злоупотреблять. Архитекторы не должны запираться в башне из слоновой кости, чтобы построить в итоге запутанный, замкнутый набор фитнес-функций, который будет только мешать разработчикам и командам. Фитнес-функции — способ создать исполняемый контрольный список *важных*, но не *срочных* принципов в проектах программной разработки. Многие проекты изобилуют срочными задачами, из-за чего некоторые важные принципы остаются без внимания. По этой же причине зачастую растет технический долг: «Мы знаем, что это плохо, но вернемся и исправим это позже», и это «позже» никогда не наступает. Когда архитекторы регламентируют качество и структуру кода, а также прикладывают другие усилия, чтобы работа не зацикливалась на непрерывно работающих фитнес-функциях, они таким образом создают чеклист требований к качеству, который разработчики игнорировать не смогут.

В своей превосходной книге «A Checklist Manifesto»[1] (Metropolitan Books, 2009) Атул Гаванде (Atul Gawande) рассказывает, как профессионалы — хирурги, пилоты и другие — повсеместно используют в своей работе чек-листы (иногда это требуется по закону). И дело не в том, что они не знают свою работу или забывчивы, — просто, если выполнять одну и ту же задачу многократно, легко допустить ошибку, случайно пропустив какой-то шаг; чек-листы как раз этого сделать и не дают. Фитнес-функции представляют собой чек-лист важных принципов, которые определяются архитекторами и выполнение которых проверяется в процессе сборки, чтобы разработчики случайно (или намеренно, из-за внешних факторов, например сжатых сроков) их не пропустили.

Сочетание метрик (наряду с другими методами проверки) и инженерных практик открывает новые уровни и возможности в управлении архитектурой, тем самым помогая превратить программную разработку из понятного не многим ремесла в полноценную инженерную дисциплину.

---

[1] Гаванде А. «Чек-лист. Как избежать глупых ошибок, ведущих к фатальным последствиям».

ГЛАВА 9

Использование метрик ПО для обеспечения сопровождаемости

Александр фон Цитцевиц

В этой главе представлены несколько интересных метрик ПО, которые можно использовать для управления проектом. Эти метрики измеряют такие параметры, как связанность кода, архитектурная эрозия, сложность кода и качество проектирования. При правильном применении они играют важную роль в поддержании высокого уровня сопровождаемости, снижении общих затрат на разработку и сопровождение и уменьшении проектных рисков. Регулярное отслеживание метрик позволяет выявлять негативные тенденции на ранней стадии и устранять проблемы, пока это еще легко сделать.

Зачем использовать метрики

В любой отрасли, которая создает сложные продукты, метрики нужны для обеспечения качества и удобства использования продуктов. Современное производство немыслимо без строгих критериев качества. В этом отношении программная разработка явно отстает от других отраслей, хотя именно в ней такой подход был бы особенно полезен.

Лучший способ применять метрики — настроить *цикл обратной связи на их основе* (рис. 9.1). Использование таких циклов гарантирует, что продукты соответствуют измеримым стандартам качества. Такое применение не только повышает общее качество, но и упрощает сопровождение ПО, а следовательно, и производительность каждого разработчика в проекте. Хорошая сопровождаемость ПО означает, что читать и понимать его код легче. Это, в свою очередь, приводит к тому, что разработчики тратят меньше времени на чтение кода и больше — на его улучшение и дополнение.

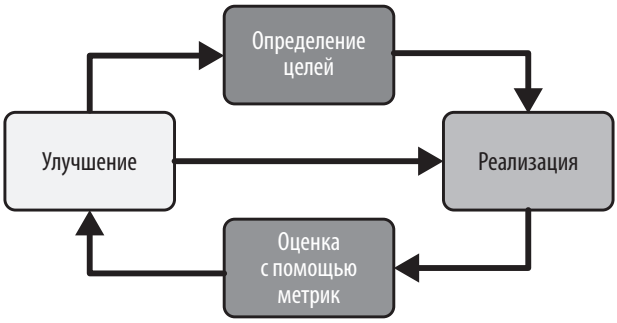


Рис. 9.1. Цикл обратной связи на основе метрик

На рис. 9.1 показан цикл обратной связи на основе метрик. В таком цикле сначала определяются измеримые цели, которые можно оценить с помощью набора метрик. Затем идет работа над реализацией продукта с постоянной проверкой достижения поставленных целей. Если цель не достигнута, реализация дорабатывается до тех пор, пока не будет ей соответствовать, и только потом работа продолжается.

При работе с унаследованными системами бывает очень трудно применять стандартный набор целевых метрик, просто потому, что этих целей не существовало при первоначальной реализации системы. Обычно в этом случае метрики показывают множество нарушений, и имеет смысл начать с более мягких целей, достижимых разумными усилиями. В противном случае разработчики будут перегружены потоком проблем, что негативно скажется на их моральном состоянии. Когда первичные цели будут достигнуты, можно задавать более строгие требования, тем самым постоянно улучшая унаследованный код. Конечно, это имеет смысл, только если унаследованная система все еще представляет ценность и находится в активной разработке. Улучшать метрики для статичного кода практически бессмысленно.

Энтропия убивает ПО

Главный враг разработки нетривиальных программных систем — *энтропия*, также известная как *структурная эрозия*. Конечное состояние структурной эрозии хорошо известно большинству разработчиков как пресловутый «большой ком грязи» — синоним безнадежно запутанного кода, потерявшего всякую архитектурную целостность. Этот термин описывает сильно связанную систему со множеством нежелательных зависимостей между частями, которые не должны быть связаны друг с другом. Типичный симптом: изменение в одной части системы нарушает работу другой, совершенно не связанной с ней части.

Другой симптом — множество циклических зависимостей, которые приводят к образованию больших групп таких зависимостей (рис. 9.2). Программные метрики отлично подходят для получения численных значений энтропии, тем самым становясь идеальным инструментом для борьбы с ней.

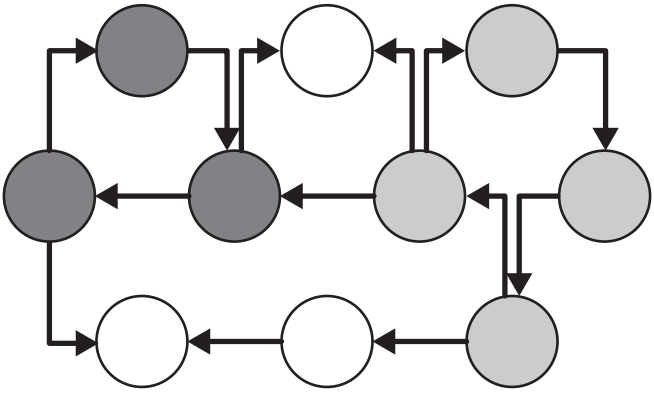


Рис. 9.2. Визуализация групп циклических зависимостей

На рис. 9.2 показана концепция группы циклических зависимостей. Узлами графа могут быть исходные файлы, пространства имен, пакеты или любые другие части программной системы. Стрелки представляют зависимости между этими элементами. В этом примере есть две группы циклических зависимостей, выделенные разными оттенками серого. Белые узлы не участвуют ни в каких циклических зависимостях.

Анализ систем с открытым исходным кодом показывает, что, достигнув определенного размера, эти группы начинают непрерывно расти. Проект Apache Cassandra — хороший пример подобного явления. В версии 2 в нем уже была группа циклических зависимостей, состоящая примерно из 450 Java-файлов. В версии 3 эта группа выросла до более чем 900 элементов, а в четвертой — до более чем 1300. Такие большие группы можно назвать «раком кода»: они растут и поглощают все бóльшие куски кода. В четвертой версии Cassandra этот «рак» даже дал метастазы, породив две новые группы из 143 и 31 элемента. Теперь около 75 % всех элементов вовлечены в большие группы циклических зависимостей.

На уровне пакетов дела обстоят еще хуже. Из 113 Java-пакетов 102 входят в одну большую группу циклических зависимостей (рис. 9.3). Поскольку пакеты или пространства имен идеально подходят для выражения архитектурной группировки и замысла, особенно важно, чтобы зависимости между ними не были зациклены.

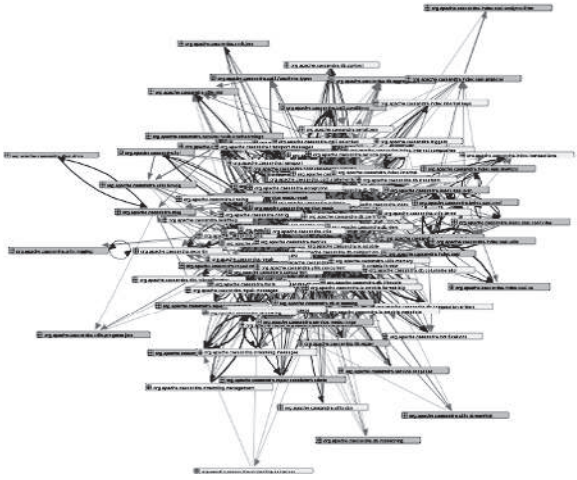


Рис. 9.3. Группа циклических зависимостей в пакете из 102 элементов в Apache Cassandra (визуализация Sonargraph)

Токсичность циклических зависимостей

Почему циклические зависимости — это плохо? Ведь с Cassandra, кажется, все в порядке. Во-первых, из-за циклических зависимостей невозможно тестировать изолированные участки кода. Во-вторых, эти зависимости усложняют понимание кода новыми разработчиками, поскольку произвольно взятый исходный файл может прямо или косвенно зависеть практически от чего угодно.

Еще одна проблема сильной связанности — невозможность изолировать и заменять отдельные функции без рискованных и затратных по времени глобальных изменений. Декомпозиция на модули становится невозможной. Можно сказать, что разработчики Cassandra свели ее архитектурную диаграмму к одному блоку под названием «Cassandra». Хотя такая диаграмма легко читается, она ничего не говорит о внутренней структуре ПО.

Но не все так плохо — все циклические зависимости можно разорвать. Например, можно применить «принцип инверсии зависимостей», впервые описанный Робертом Мартином (Robert C. Martin) в его книге «Agile Software Development, Principles, Patterns, and Practices» (Pearson, 2002). Этот принцип позволяет инвертировать зависимость в группе, введя интерфейс, что обычно разрывает цикл. Есть и другие способы разрыва циклов. Например, можно вынести циклическую зависимость в класс более высокого уровня, который будет зависеть от элементов цикла, освобождая их от прямой зависимости друг от друга. Или же можно, наоборот, вынести

цикл в класс более низкого уровня, который будет управлять взаимодействием между элементами цикла. Наконец, можно просто переместить часть функций из одного класса в другой.

Иными словами, нет веских причин позволять «раку кода» бесконтрольно разрастаться. Если избегать больших групп циклических зависимостей, код станет лучше. Его будет проще тестировать, понимать и сопровождать, и, конечно, повторно использовать.

Как могут помочь метрики

Чтобы избежать структурной эрозии, можно анализировать структуру зависимостей кода с помощью метрик. Например, количества элементов в самой большой группе циклических зависимостей исходных файлов. Задав пороговое значение, скажем, в пять элементов, вы получите предупреждение, как только в группе циклических зависимостей их станет шесть или больше. Можно даже настроить процесс так, чтобы превышение этого порога приводило к ошибке сборки. Тогда, чтобы сборка снова стала «зеленой», придется изменить код: разорвать группу циклических зависимостей или хотя бы уменьшить ее до пяти элементов или менее и устранить таким образом причину предупреждения. Поскольку количество циклических элементов еще невелико, такое исправление можно внести легко и быстро.

Конечно, лучше всего полностью избегать циклических зависимостей, но будем придерживаться разумных ограничений. Небольшие группы циклических зависимостей в пределах одного пространства имен или пакета не наносят большого вреда, если не давать им разрастаться до чудовищных размеров. Стоит также отметить, как уже было сказано, что любую циклическую зависимость можно разорвать, например, с помощью принципа инверсии зависимостей Роберта Мартина, который позволяет инвертировать направление зависимостей путем реализации интерфейсов (рис. 9.4). Есть и другие способы разрыва циклических зависимостей, но в этой главе мы не будем на них останавливаться.

Если вы просто введете ограничение размера групп циклических зависимостей, то только одно это уже исключит превращение системы в «большой ком грязи». По моему многолетнему опыту, более 80 % сложных программных систем, в которых больше 100 000 строк кода, в итоге превращаются в «большие комья грязи». В этом легко убедиться, проанализировав несколько случайно выбранных систем с открытым исходным кодом. Если система сможет избежать этой участи, она уже окажется лучше 80 % всех систем схожего размера и сложности. И на мой взгляд, это легко — главное, выставить этот приоритет в начале нового проекта.

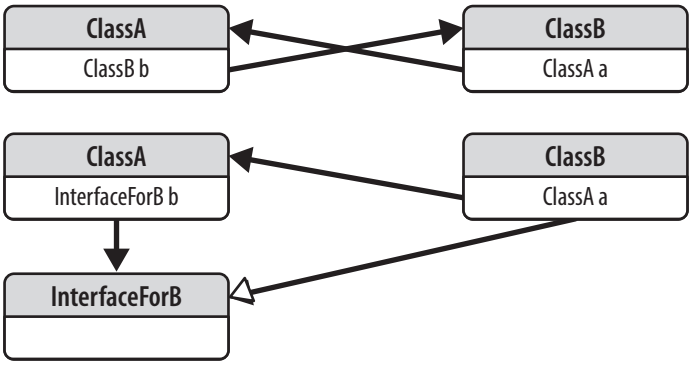


Рис. 9.4. Использование принципа инверсии для разрыва циклических зависимостей

Существуют успешные проекты, которые избежали ловушки структурной эрозии, следуя подобным правилам. Популярный и известный пример — фреймворк Spring, который хорошо структурирован, имеет хорошую архитектуру и незначительное количество небольших групп циклических зависимостей.

Итак, если вы согласны, что ограничивать циклические зависимости — это хорошо, возникает вопрос: как это сделать? Все просто: нужны инструменты. Например, Sonargraph-Explorer (https://oreil.ly/VVVCR) позволяет прерывать сборку исходя из размера групп циклических зависимостей и бесплатен абсолютно для всех[1].

Почему метрики не используются повсеместно

После всех доводов в пользу метрик трудно понять, почему циклы обратной связи на их основе так редко применяются в программной разработке. Когда я выступаю с докладами на эту тему и спрашиваю, кто в работе использует метрики, из ста слушателей руки поднимают максимум один-два человека. Полагаю, тому есть несколько важных причин:

- Многие разработчики и архитекторы мало что знают о метриках и способах их применения. Даже те, кто изучал компьютерные науки, знают, что метрики редко входят в учебную программу или в лучшем случае упоминаются вскользь. При отсутствии академического образования вероятность того, что человек целенаправленно изучал метрики, еще ниже.

[1] К сожалению, на начало 2022 года Sonargraph-Explorer поддерживал только Java, C# и Python. Для других языков, возможно, придется писать собственный инструмент или полагаться на коммерческие решения.

- Для работы с метриками нужны инструменты, которые будут их собирать. Это по-прежнему нишевая область в инструментарии программной разработки. И хотя отличные инструменты существуют (например, Sonargraph-Explorer), они малоизвестны и поддерживают только основные языки, такие как Java или C#.
- Для эффективного применения большинства метрик необходимы контекст и определенный уровень экспертных знаний. Если учитывать лишь несколько правил на основе метрик и выбрать для них не те показатели, то улучшить код не получится — более того, его можно сделать хуже. Если просто научить разработчиков подгонять код под метрику, улучшения останутся поверхностными.
- Слишком большое количество правил на основе метрик будет только раздражать разработчиков и замедлять работу. Я считаю, что правил на основе метрик не должно быть более пяти-шести. Каждое дополнительное правило будет приносить все меньше пользы.
- Многие организации и так с трудом справляются с непомерным техническим долгом, поэтому на улучшение процессов у них зачастую просто не остается ресурсов.
- Правила на основе метрик полезны, только если их нарушение инициирует определенные действия. Следовательно, для успеха нужна автоматизация, внедрение которой тоже требует времени.

В конце главы я предложу согласованный набор правил, соблюдение которых значительно повысит техническое качество любого проекта.

Инструменты для сбора метрик

Прежде чем углубляться в детали различных метрик, разберемся с инструментами. Ведь метрики нужно как-то собирать, и в большинстве случаев делать это вручную нецелесообразно. Конечно, можно попытаться написать собственные инструменты для сбора метрик, но это почти всегда плохая затея, если уже есть готовое решение. Более того, разработка собственного инструмента всегда обходится дороже покупки коммерческого решения, особенно в долгосрочной перспективе с учетом сопровождения. Что, если в языке появятся новые возможности? Инструмент должен будет их поддерживать. Кроме того, для сбора более сложных метрик, которые описаны в этой главе, потребуется построить полную модель зависимостей всего приложения. Для этого, по сути, придется реализовать этапы парсинга и разрешения зависимостей компилятора, а это сложно, долго и рискованно. Это время лучше потратить на улучшение кода и процесса разработки.

В табл. 9.1 приведен выборочный список популярных инструментов для сбора метрик.

Таблица 9.1. Выборочный список популярных инструментов для сбора метрик и их основные возможности

Инструмент	Возможности
Understand (http://scitools.com)	Коммерческий, поддерживает много языков; в основном метрики размера и сложности
NDepend (http://ndepend.com)	Коммерческий, поддерживает .Net; в основном метрики размера и сложности
Source monitor (https://oreil.ly/GmGE3)	Бесплатный, поддерживает C++, C, C#, VB.NET, Java, Delphi, Visual Basic (VB6) и HTML; только метрики размера и сложности
SonarQube (http://sonar qube.org)	Бесплатен для некоторых языков, в коммерческой версии больше возможностей; в основном ориентирован на метрики размера и сложности
Sonargraph-Explorer (https://oreil.ly/ILjMe)	Бесплатный для Java, C# и Python; полный набор метрик, включая связанность, циклические зависимости, размер и сложность; коммерческая версия, Sonargraph-Architect, также поддерживает метрики истории изменений (Git) и C/C++

Некоторые инструменты бесплатны, а во всех коммерческих предусмотрены пробные версии, поэтому я рекомендую попробовать их все и выбрать тот, который понравится больше всего. Важный критерий при выборе — возможность проверять пороговые значения метрик в автоматизированной сборке. Автоматизация — это действительно решающий фактор успеха.

Полезные метрики

Рассмотрим несколько полезных категорий метрик. Начнем с метрик для оценки связанности и структурной эрозии, поскольку эти параметры наиболее важны для сохранения удобства сопровождения продукта в долгосрочной перспективе. Затем рассмотрим метрики размера и сложности. Далее — метрики истории изменений. И наконец, обсудим несколько метрик, которые не относятся ни к одной из этих категорий.

Метрики для оценки связанности и структурной эрозии

Следующие метрики позволяют оценить связанность и структурную эрозию.

Среднее количество зависимостей компонента, стоимость распространения и другие родственные им метрики

Метрику `average component dependency, ACD` («*среднее количество зависимостей компонента*») впервые описал Джон Лакос (John Lakos) в книге «Large Scale C++ Design» (Addison-Wesley, 1996). Эта метрика показывает, от скольких элементов в графе зависимостей в среднем зависит случайно выбранный элемент (прямо или косвенно, включая самого себя). Чтобы разобраться в этой метрике, рассмотрим граф зависимостей на рис. 9.5.

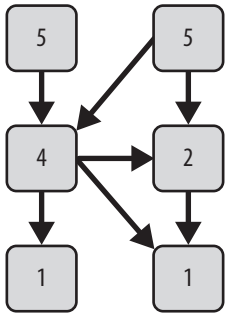


Рис. 9.5. Граф зависимостей со значениями метрики Depends Upon («зависит от»)

Лакос называет блоки, изображенные на рисунке, *компонентами*. В языке C/C++ компонент состоит из файла с исходным кодом и связанного с ним файла заголовка. В других языках, например в Java, компоненты, как правило, представляют собой отдельные файлы с исходным кодом.

Стрелки обозначают направленные зависимости, а числа — значение метрики `Depends Upon` («*зависит от...*») для соответствующего блока. Например, блоки в нижней части графа зависят только от самих себя, поэтому их значение `Depends Upon` равно 1. Компонент в середине справа зависит только от компонента под ним, поэтому его значение равно 2. Компонент в середине слева получает значение 4, поскольку он зависит от среднего элемента справа, обоих нижних элементов и самого себя. Компоненты верхнего уровня зависят от всех элементов первого и второго уровней, а также от самих себя, поэтому их значение равно 5. Объяснить эти числа можно и по-другому: нужно подсчитать количество достижимых узлов и прибавить к этому числу единицу. В теории графов множество узлов, которых можно достичь из заданного узла, называется его *замыканием* (closure).

Если сложить все значения в блоках, получится сумма, которая называется `Cumulative Component Dependency, CCD` («*кумулятивная зависимость компонентов*»). В данном случае она равна 18. Теперь, если разделить ее на количе-

ство блоков, получится ACD: в нашем случае 18, деленное на 6 (количество узлов), дает 3. Минимальное значение ACD всегда равно 1. Таким было бы значение для системы, в которой нет никаких зависимостей. Максимальное значение равно количеству узлов; в нашем примере это 6.

У метрики Depends Upon есть аналог — Used From («*используется из...*»). На рис. 9.6 показан тот же граф со значениями метрики Used From.

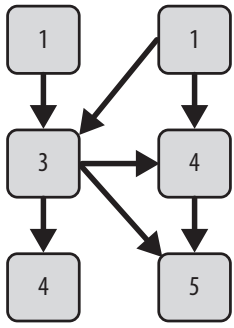


Рис. 9.6. Тот же граф со значениями метрики Used From

Метрика Used From показывает количество узлов, которые прямо или косвенно связаны с выбранным узлом. Если сложить значения Used From и Depends Upon, всегда получится одно и то же число. Так и должно быть, поскольку каждая метрика рассматривает только один из двух концов направленной зависимости.

Если для каждого узла разделить значение его метрики Depends Upon на общее количество узлов, получится новая метрика узла — Fan Out («*коэффициент исходящих зависимостей*»). Например, значение Fan Out для верхнего левого узла будет равно 1/6. Если проделать то же самое для Used From, получится Fan In («*коэффициент входящих зависимостей*»). Среднее значение Fan In равно среднему значению Fan Out, а также метрике propagation cost («*стоимость распространения изменений*»), о которой я расскажу ниже.

Любопытно, что сумма всех значений Fan Out дает то же значение, что и ACD. Если разделить это значение на количество узлов, получится среднее значение Fan Out, или propagation cost. Для графа на рис. 9.6 propagation cost составляет 0,5, или 50 %.

Это метрики базового уровня, которые используются для расчета метрик более высокого уровня. Они также могут быть полезны для выявления компонентов с большим количеством входящих или исходящих зависимостей.

Когда использовать ACD и что нужно учитывать при работе с этой метрикой? Значение ACD — один из первых показателей, на которые я обращаю внимание при оценке системы. Он дает довольно точное представление о том, насколько сильно связана система. Разумеется, его нужно рассматривать в соотношении с общим количеством компонентов в системе. Значение 100 для системы из 1000 компонентов приемлемо, тогда как то же значение для системы всего из 100 компонентов будет губительным.

Метрика Propagation Cost, PC («*стоимость распространения изменений*»)[1] показывает, насколько сильно связана система. Чем выше процент, тем выше связанность.

PC также можно рассчитать, разделив ACD на количество узлов (математически это то же самое, что и расчет среднего значения Fan In или Fan Out). По сути, такое деление нормализует ACD до значения, которое легче сравнивать. В нашем примере это значение составит 3/6, или 50 %. Иными словами, каждое изменение в этой системе в среднем может затронуть 50 % всех компонентов. Для большой системы это очень плохой показатель, а для таких небольших примеров он не имеет значения.

Важно понимать, что PC можно также определить как CCD, деленное на количество компонентов (n) в квадрате. ACD = CCD/n, следовательно, PC = CCD/n^2. Таким образом, если количество компонентов удваивается, CCD системы должно вырасти в 4 раза, чтобы значение PC осталось прежним. Эту метрику всегда нужно минимизировать, но, если ее значение снижается из-за добавления в систему новых компонентов, это не всегда хорошо. В больших системах (от 500 компонентов) мы обычно наблюдаем снижение PC при росте количества компонентов, даже если связанность очень высока. Это происходит потому, что для роста CCD пропорционально квадрату числа компонентов требуется добавить очень много дополнительных связей.

С учетом этих оговорок PC следует оценивать так:

- Для небольших систем ($n < 500$) высокие значения PC не так критичны.
- Для систем среднего размера ($500 \leq n < 5000$) значения PC выше 20 % уже служат поводом для беспокойства, а выше 50 % — указывают на серьезные проблемы с большими группами циклических зависимостей.

---

[1] Впервые описано в работе: Карлисс Болдуин (Carliss Baldwin), Джон Руснак (John Rusnak) и Алан Маккормак (Alan MacCormack), «Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code», Management Science 52, no. 7 (2006): 1015-1030.

- Для огромных систем ($n \geq 5000$) даже 10 % — это уже серьезный недостаток.

Оценивать влияние метрики PC проще, если одновременно смотреть на ACD. Если в системе из 5000 компонентов PC составляет 10 %, то ACD будет равен 500. Это тревожный показатель, поскольку каждое изменение может в среднем затронуть 500 компонентов.

Цикличность и относительная цикличность

Как правило, высокие значения ACD и PC указывают на наличие больших групп циклических зависимостей в графе зависимостей анализируемой системы. Чтобы убедиться в этом, можно использовать специализированные метрики для выявления циклических зависимостей. Ранее в этой главе уже упоминалась простая, но полезная метрика: количество элементов в самой большой группе циклических зависимостей системы. Эту величину можно рассчитать для элемента любого типа, но обычно она наиболее полезна на уровне компонентов и пространств имен/пакетов.

Размер самой большой группы циклических зависимостей — это один из тех показателей, на которые стоит обращать внимание в первую очередь при анализе системы. В хорошо спроектированной системе для компонентов это значение не должно превышать 5. Иногда более крупные группы циклических зависимостей допустимы, если на то есть веская техническая причина или если они находятся в той части кода, которая почти не меняется.

В любом случае циклические зависимости компонентов не должны выходить за пределы одного пространства имен или пакета. К группам циклических зависимостей на уровне пространств имен/пакетов рекомендуется применять политику нулевой терпимости. Иными словами, в системе не должно быть циклических зависимостей между пространствами имен или пакетами.

Еще одна полезная метрика — `Relative Cyclicity` («*относительная цикличность*»), которая обычно рассчитывается как для каждого модуля, так и для системы в целом. Она основана на метрике `Cyclicity` («*цикличность*»), которая определяется как квадрат количества элементов в группе циклических зависимостей. Группа циклических зависимостей из 5 элементов будет иметь цикличность, равную 25. Теперь нужно сложить показатели цикличности всех групп циклических зависимостей для модуля или системы в целом, извлечь из полученного значения квадратный корень и разделить его на количество элементов в системе или модуле:

$$\text{Относительная цикличность} = 100^* \frac{\sqrt{\text{Сумма цикличностей}}}{n}.$$

Эта метрика может быть очень полезна. Для демонстрации проведем небольшой мысленный эксперимент. Допустим, в системе 100 компонентов, и все они входят в одну большую группу циклических зависимостей из 100 элементов. Цикличность такой группы составит 100^2, или 10 000. По формуле получится 1, что означает стопроцентную относительную цикличность. Это наихудшее возможное значение; система имеет максимально возможную циклическую зависимость.

Теперь допустим, что вместо одной большой группы циклических зависимостей имеется 50 маленьких групп по 2 элемента в каждой. В этом случае цикличность одной группы будет $2^2 = 4$, а сумма цикличностей для всей системы — 50 * 4 = 200. По формуле получится:

$$100* \frac{\sqrt{200}}{100} = 14{,}14$$

или 14,14 %. Это значение гораздо лучше, хотя каждый компонент попрежнему находится в циклической зависимости. Когда речь заходит о группах циклических зависимостей, чем они меньше, тем лучше, поскольку небольшие группы гораздо легче разорвать.

Индекс структурного долга

Относительная цикличность — неплохой показатель того, насколько система страдает от циклических зависимостей, но у этой метрики есть недостаток: она ничего не говорит о том, насколько сложно разорвать обнаруженные циклы. Чтобы лучше понять эту проблему, проведем еще один мысленный эксперимент.

Представим простую систему из 10 исходных файлов. Первый файл зависит от второго, второй — от третьего и так далее, пока мы не дойдем до десятого файла, который, в свою очередь, зависит от первого. Так образуется простая группа циклических зависимостей из 10 элементов, и в итоге все файлы оказываются задействованными в цикле. Относительная цикличность такой системы составит 100 %. Но подобную группу циклических зависимостей легко разорвать, удалив или инвертировав всего одну зависимость. Поэтому относительную цикличность системы легко свести к нулю.

Конечно, бывают и гораздо худшие сценарии. Рассмотрим другую крайность. Снова возьмем систему из 10 исходных файлов, но теперь пусть каждый файл имеет двунаправленную циклическую зависимость с каждым прочим файлом. В итоге получится 90 зависимостей: (9 + 8 + 7 + ... + 1) × 2. В этом случае относительная цикличность тоже составит 100 %, но, чтобы разорвать все циклические зависимости, придется удалить или

инвертировать не менее 45 из них. Это потребует куда больше усилий, чем в первом примере.

Именно для таких случаев я и разработал метрику `Structural Debt Index, SDI` («*индекс структурного долга*»). Эта метрика с помощью алгоритма обхода графа находит в группе циклических зависимостей минимальный набор связей для разрыва и формирует список зависимостей, которые необходимо разорвать. Дело в том, что зависимости между исходными файлами могут состоять из множества отношений использования. Например, если класс A вызывает три разных метода класса B, зависимость будет состоять из трех использований, или *синтаксических зависимостей* (parser dependency). Количество использований выступает в роли веса связи, чтобы алгоритм отдавал предпочтение разрыву связей с меньшим весом. Затем `SDI` рассчитывается по формуле:

$$SDI = 10^{*}\text{количество разрываемых связей} + \Sigma \text{ весов разрываемых связей}.$$

В первом примере (10 компонентов в простом цикле) достаточно разорвать всего одну связь. Если предположить, что вес этой связи равен 1, `SDI` системы составит 11 (10 × 1 + 1).

Количество разрываемых связей умножается на константу, поскольку для каждой из них нужно придумать способ разрыва (например, инвертировать ее через инверсию зависимостей). Разумеется, каждое использование также может потребовать дополнительных усилий. Замысел этой метрики в том, чтобы ее значение было хотя бы приблизительно пропорционально усилиям, которые требуются для разрыва группы циклических зависимостей. Метрика рассчитывается для каждой группы циклических зависимостей, а затем значения суммируются на уровне модуля и системы. Такие инструменты, как Sonargraph-Architect, используют компоненты этой метрики, чтобы ранжировать группы циклических зависимостей по простоте их устранения.

Метрику `SDI` лучше всего использовать в сочетании с относительной цикличностью и удерживать ее значение как можно ближе к нулю. Если значение SDI постоянно растет, это симптом того, что в системе развивается «рак кода».

Уровень сопровождаемости

В этом разделе я расскажу, как создавал новую метрику для оценки того, насколько код прост в сопровождении и хорошо спроектирован, — священный Грааль в мире метрик ПО. Эту работу я выполнил совместно с заказчиком,

который предоставил для проверки моей гипотезы несколько крупных проектов. Предполагалось, что значения этой метрики будут более или менее соответствовать тому, как сами разработчики оценивают удобство сопровождения системы. Было решено отслеживать эту метрику в ночных сборках и использовать ее как лакмусовую бумажку: если значения ухудшаются, значит, пора делать рефакторинг. Также ее планировалось использовать для сравнения состояния всех программных систем в организации и для принятия решений о том, что дешевле: переписать часть систем с нуля или выполнить их рефакторинг.

Приступая к этой работе, мы уже изучили несколько метрик для оценки связанности и циклических зависимостей. Замысел этой новой экспериментальной метрики состоял в том, чтобы свести существующие показатели в единую метрику, которую можно было бы использовать как фитнес-функцию для оценки качества дизайна.

Под «качественным дизайном» я подразумеваю такой дизайн проекта, который использует как горизонтальное разделение на уровни, так и вертикальное разделение (или изоляцию) функциональных компонентов. Разделение программной системы на функциональные части я называю *вертикализацией* (verticalization). Оно показано на рис. 9.7.

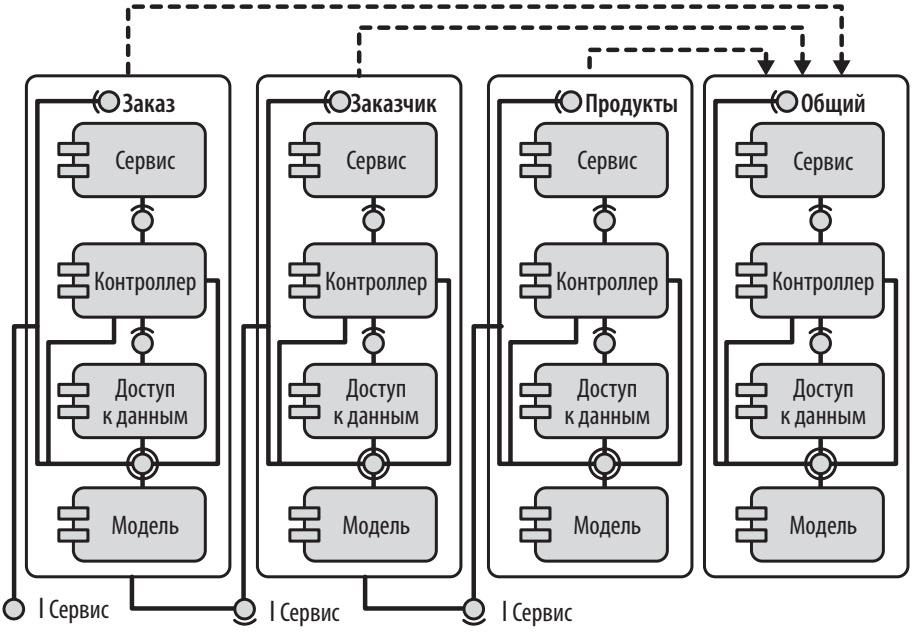


Рис. 9.7. Качественный вертикальный дизайн

Функциональные компоненты находятся в собственных изолированных модулях (silo), и их зависимости не являются циклическими; между модулями существует четкая иерархия. Такой дизайн также можно охарактеризовать как вертикальное разделение на уровни или как микросервисы в монолите. Еще раз подчеркну: управление зависимостями — ключевой аспект любого архитектурного стиля. Если вы реализуете микросервисы, важно рассматривать зависимости между ними как более тяжеловесные, поскольку все они используют тот или иной способ межпроцессного взаимодействия. Задержка при обмене данными между микросервисами всегда намного выше, чем при вызове функций внутри процесса, и требует повышенного внимания к обработке ошибок. Что произойдет, если целевой сервис недоступен или возникли проблемы с сетью?

К сожалению, во многих программных системах вертикализация отсутствует. Основная причина в том, что организовывать код в изолированные модули необязательно. Поскольку сделать это правильно сложно, границы между модулями размываются, и функциональность, которая должна находиться в одном модуле, оказывается «распыленной» по нескольким. Это провоцирует появление циклических зависимостей между модулями и тем самым — постоянное усложнение сопровождения.

Как же измерить вертикализацию? Сначала нужно построить многоуровневый граф зависимостей компонентов системы. Однако его можно правильно разбить на уровни, только если между компонентами нет циклических зависимостей. Соответственно, первым делом нужно объединить все циклические группы в единые узлы.

В системе, показанной на рис. 9.8, узлы F, G и H образуют циклическую группу, поэтому их следует объединить в один логический узел FGH.

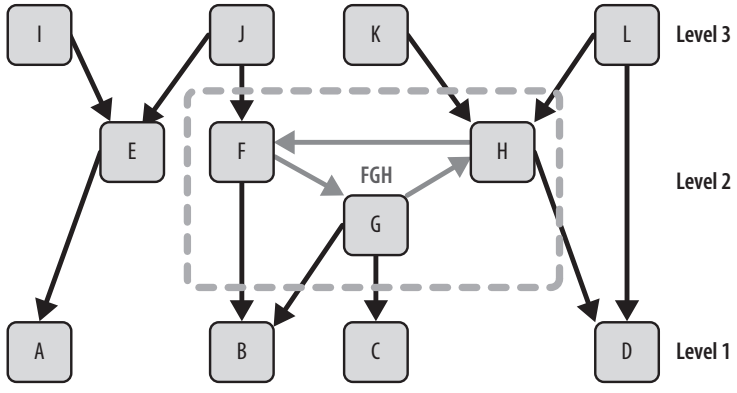


Рис. 9.8. Многоуровневый граф зависимостей с циклической группой, объединенной в узел (FGH)

В результате получится три уровня. У нижнего уровня есть только входящие зависимости, у верхнего — только исходящие. Чтобы обеспечить сопровождаемость, желательно, чтобы как можно меньше компонентов имели входящие зависимости. Компоненты без зависимостей можно изменять, не опасаясь затронуть другие части системы. Кроме того, необходимо, чтобы остальные компоненты оказывали влияние на как можно меньшее число компонентов на вышележащих уровнях.

В нашем примере узел A прямо или косвенно влияет только на узлы E, I и J. Узел B, напротив, влияет на все компоненты второго и третьего уровней, кроме E и I.

Циклическая группа FGH, очевидно, ухудшает этот показатель. Таким образом, узел A больше способствует сопровождаемости, чем B, так как с меньшей вероятностью нарушит работу вышележащих уровней. Для каждого логического узла можно рассчитать его вклад (c_i) в новую метрику, которая позволяет оценить удобство обслуживания:

$$c_i = \frac{\text{размер}(i) * \left(1 - \frac{\inf(i)}{\text{число компонентов на верхних уровнях } (i)}\right)}{n}.$$

Здесь n — это общее число компонентов; *размер* (i) — число компонентов в логическом узле (это значение превышает 1 только для тех логических узлов, которые были образованы из циклических групп); *inf*(i) — число компонентов, на которые влияет узел c_i.

Для примера рассчитаем это значение для узла A:

$$c_A = \frac{1 \times \left(1 - \frac{3}{8}\right)}{12}.$$

Результат: ~0,052. Если суммировать значения c_i для всех узлов, получим первую версию новой метрики, которую назовем `Maintainability Level`, `ML` («*уровень сопровождаемости*»):

$$ML_1 = 100 \times \Sigma_{i=1}^{k} c_i.$$

В этой формуле k — количество логических узлов, которое будет меньше n, если в системе присутствуют циклические зависимости между компонентами. В нашем примере k = 10, а n = 12. Чтобы получить значение в процентах, умножаем результат на 100. Чем выше итоговое значение `ML`, тем проще обслуживать систему.

В примере, показанном на рис. 9.8, сумма c_i составит 53/96, а итоговое значение ML — 55 %. Поскольку мы рассматриваем очень небольшую систему, которая состоит всего из 12 модулей для компиляции, полученное значение не критично. Далее я усовершенствую эту метрику, чтобы учесть тот факт, что для небольших систем она, как правило, дает относительно плохие результаты. Та же проблема наблюдалась и у метрики `Propagation Cost` (стоимость распространения изменений): из-за малого числа элементов средняя связанность в таких системах всегда выше. Важность снижения связанности возрастает по мере увеличения размера системы.

Поскольку в любой системе есть зависимости, достичь значения в 100 % невозможно, если только ни один из компонентов не имеет входящих зависимостей. Однако все узлы на самом верхнем уровне будут вносить в метрику свой максимальный вклад c_i. Вклад узлов, расположенных на нижних уровнях, будет тем меньше, чем на большее число узлов на верхних уровнях они влияют. Группы циклических зависимостей увеличивают для всех своих членов число узлов, на которые они влияют на вышестоящих уровнях, и поэтому, как правило, отрицательно сказываются на значении метрики.

Известно, что циклические зависимости отрицательно влияют на удобство обслуживания, особенно если циклическая группа содержит большое количество узлов. В первой версии метрики ML мы поняли, что не сможем учесть это негативное влияние, если узел, образованный циклической группой, окажется на самом верхнем уровне. Из этих соображений мы добавили штраф для циклических групп, в которых более пяти узлов:

$$\text{штраф}(i) = \begin{cases} \dfrac{5}{\text{размер}(i)}, & \text{если размер } (i) > 5 \\ 1, & \text{в противном случае.} \end{cases}$$

В данном случае значение штрафа, равное 1, означает, что штраф отсутствует. Значения ниже 1 уменьшают вклад соответствующего логического узла. Например, если циклическая группа состоит из 100 узлов, ее вклад составит лишь 5 % от первоначального значения. Вторая версия метрики ML (ML_2) учитывает этот штраф:

$$ML_2 = \left| 100 \times \sum_{i=1}^{k} c_i^* \times \text{штраф}(i) \right|.$$

Эта метрика довольно эффективна. Когда мы применяем ее к хорошо спроектированным системам, то получаем значения выше 90. Для систем без четко выраженной архитектуры, таких как Apache Cassandra, мы получаем значение в районе 20.

Когда я проверял эту метрику на проектах заказчиков, я сделал еще два наблюдения, которые потребовали ее корректировок. Во-первых, она не очень хорошо работала для небольших модулей, содержащих менее 100 компонентов. Такие модули часто показывали относительно низкие значения `ML`, поскольку небольшое число компонентов естественным образом увеличивает относительную связанность и при этом не снижает удобство обслуживания.

Во-вторых, у меня был проект на Java, разработчики которого считали, что его трудно сопровождать. Тем не менее метрика показала для него значение в районе 90. При более внимательном рассмотрении оказалось, что проект действительно имел хорошую структуру компонентов, почти без циклических зависимостей, но вот структура пакетов была в полном беспорядке. Почти все пакеты в самом важном модуле входили в одну циклическую группу. Обычно такое происходит, когда отсутствует четкая стратегия распределения классов по пакетам. Отсутствие такого распределения затрудняет разработчикам поиск нужных классов.

Первую проблему можно решить, добавив плавающее минимальное значение для `ML`, если в анализируемом модуле или системе менее 100 компонентов:

$$ML_3 = \begin{cases} (100-n)+\dfrac{n}{100}\cdot ML_2, & \text{если } n<100 \\ ML_2, & \text{в противном случае.} \end{cases}$$

Здесь n снова обозначает количество компонентов. Такой подход можно обосновать тем, что небольшие системы изначально проще обслуживать. Таким образом, при использовании плавающего минимального значения система, состоящая из 40 компонентов, никогда не получит значение ниже 60. Если подставить в эту формулу значение ML_2 из примера на рис. 9.8 (55 %), то получится 94,6 %. Это значение выглядит гораздо более адекватным, чем 55 %, если учесть, что подобные небольшие системы не создают серьезных сложностей в обслуживании.

Вторую проблему решить сложнее. В этом случае было решено рассчитывать альтернативное значение на основе `Relative Cyclicity` (относительной цикличности), которая вычисляется для зависимостей между пакетами или пространствами имен (`RCp`):

$$ML_{alt} = 100^* \left(1 - \frac{\sqrt{\text{сумма значений цикличности для пакетов}}}{n_p}\right).$$

Значение ML рассчитывается для каждого модуля системы. Затем вычисляется средневзвешенное значение (по количеству компонентов в модуле) для всех наиболее крупных модулей системы. Чтобы определить, какие модули следует учитывать при взвешивании, их сортируют по убыванию размера и добавляют к средневзвешенному значению до тех пор, пока не будет охвачено 75 % всех компонентов или модуль не будет содержать как минимум 100 компонентов. Обоснование такого подхода заключается в том, что основная активность обычно происходит вокруг более крупных и сложных модулей. Небольшие модули не вызывают сложностей в сопровождении и оказывают незначительное влияние на общую сопровождаемость системы.

Чтобы обеспечить удобство сопровождения, необходимо тщательным образом проектировать и структуру компонентов, и структуру пакетов или пространств имен. Если в одной из этих структур или в обеих есть недостатки или признаки структурного износа, удобство сопровождения также снизится.

На данный момент Sonargraph, включая его бесплатную версию Sonargraph-Explorer, — единственный инструмент, который вычисляет эту экспериментальную метрику. Если вам интересно, какие результаты покажет код, получите бесплатную лицензию на Explorer (http://www.hello2morrow.com) и проанализируйте свою систему с его помощью.

На работу по ML нас вдохновила статья о другой многообещающей метрике — Decoupling Level, DL (*«уровень разделения»*)[1]. В основе DL лежит исследовательская работа Ран Мо (Ran Mo), Юаньфан Цай (Yuangfang Cai), Рика Казмана (Rick Kazman), Лу Сяо (Lu Xiao) и Цюн Фэн (Qiong Feng) из Дрексельского университета и Гавайского университета. К сожалению, часть алгоритма вычисления DL защищена патентом, поэтому на момент написания этой книги мы не можем представлять эту метрику в своих инструментах. Но было бы интересно сравнить эти две метрики на разнообразных проектах.

Метрики для измерения размера и сложности

Следующая категория полезных метрик измеряет размер и сложность кода. Ограничение сложности важно, когда требуется обеспечить простоту кода. Разработчики бóльшую часть времени тратят на чтение кода, а запутанный

[1] Ран Мо и соавторы. «Decoupling Level: A New Metric for Architectural Maintenance Complexity» в сборнике трудов Международной конференции по программной инженерии (ICSE '16), 14–22 мая 2016 г., Остин, Техас.

код читать намного сложнее. Соответственно, чтобы избежать избыточной сложности кода, целесообразно использовать пороговые значения для метрик сложности.

Метрики размера

Начнем с нескольких простых метрик размера. Вероятно, самая известная из них — это *количество строк кода* на файл: `LoC` (`Lines of Code`). Метрика `LoC` подсчитывает каждую строку, которая содержит исполняемый код, пропуская пустые строки и строки с комментариями. В свою очередь, метрика `Total Lines` подсчитывает абсолютно все строки, включая пустые и с комментариями. Можно также подсчитывать отдельно строки с комментариями, но это немного сложнее. Часто исходные файлы содержат в начале заголовочный комментарий, в котором указана только информация об авторских правах. Такие комментарии не относятся к коду, и их следует исключать из подсчета строк с комментариями.

В некотором смысле `LoC` можно использовать и как метрику сложности. Если исходный файл содержит 5000 строк кода, он, скорее всего, сложен. Настоятельно рекомендуется ограничивать размер файлов с исходным кодом примерно 800 строками. Если файл становится больше, его следует разбить на меньшие файлы. Пороговые значения для метрик размера и сложности в основном гибкие, хотя всегда будут обоснованные исключения (главное — сохранять над ними контроль).

Для определения размера функций и методов хорошо подходит метрика *количество инструкций* (`Number of Statements`). Из названия уже понятно, как она работает: просто подсчитывает количество инструкций в методе. Всегда полезно делать функции и методы достаточно короткими, чтобы сохранить код читаемым и простым в сопровождении. Рекомендуется установить порог в 100 инструкций на функцию или метод.

Цикломатическая сложность

Метрика *цикломатическая сложность* (`Cyclomatic Complexity`) была первоначально разработана Томасом Маккейбом (Thomas McCabe) в 1976 году. Она вычисляет количество возможных путей выполнения кода в методе или функции, что также является минимальным количеством тестовых сценариев, которые необходимы для обеспечения стопроцентного покрытия кода тестами. Изначально определение этой метрики основывалось на графе потока управления, а также на количестве вершин и ребер в этом

графе. Однако ее вычисление можно упростить: начать с минимального значения, равного 1, а затем прибавлять по единице за каждый оператор цикла или условный оператор. Для операторов `switch` следует прибавлять количество вариантов `case`. Высокие значения цикломатической сложности, как правило, указывают на то, что функции или методы являются очень сложными и их трудно читать.

Эта метрика хорошо изучена, и известно, что при значениях выше 24 количество ошибок начинает быстро расти. Чтобы обезопасить себя, рекомендуется установить пороговое значение на уровне 15.

Существует несколько разновидностей этой метрики. *Модифицированная цикломатическая сложность* (`Modified Cyclomatic Complexity`) прибавляет к значению всего 1 за каждый оператор `switch`, поскольку такие операторы обычно значительно увеличивают метрику, не внося при этом дополнительной сложности в код. *Расширенная цикломатическая сложность* (`Extended Cyclomatic Complexity`) также прибавляет по единице за каждое логическое выражение `&&` и `||`, поскольку сокращенное вычисление этих выражений компилятором по сути представляет собой дополнительный условный оператор.

Также имеет смысл агрегировать значения до уровня классов, пакетов/пространств имен и модулей. Такая метрика называется *средней цикломатической сложностью* (`Average Cyclomatic Complexity`) и должна основываться на взвешенном среднем значении метрики цикломатической сложности.

В качестве веса для расчета среднего значения обычно используется метрика *количество инструкций* (`Number of Statements`). Использование взвешенного среднего позволяет гарантировать, что множество небольших методов, таких как сеттеры и геттеры, не будут размывать показатель общей сложности, которая складывается из крупных методов.

Глубина вложенности

Еще один хороший способ получить метрику сложности — проанализировать максимальные уровни вложенности отступов в коде функций и методов. Чем глубже вложенность, тем сложнее метод. Эта метрика на удивление хорошо помогает выявлять сложный код. Ее также легко агрегировать до уровня класса или файла с исходным кодом, используя взвешенное среднее по всем функциям и методам в классе или файле. Как и в случае со средней цикломатической сложностью, среднее зна-

чение следует взвешивать по количеству инструкций. Я рекомендую устанавливать пороговое значение для максимального уровня вложенности равным 4.

Метрики истории изменений

Как отмечает Адам Торнхилл (Adam Tornhill) в своей прекрасной книге «Your Code as a Crime Scene» (Pragmatic Bookshelf, 2015), система контроля версий — это настоящий кладезь ценных данных, из которых можно узнать, какие файлы изменяются чаще всего, сколько людей владеют информацией об определенных участках кода, какой объем кода был изменен за заданный промежуток времени и многое другое. Эта информация ценна тем, что помогает выявлять в коде проблемные точки, которым в первую очередь необходим рефакторинг.

Частота изменений

Интересно знать, как часто конкретный исходный файл изменяется в течение определенного периода времени, поскольку частые изменения могут свидетельствовать о нестабильных элементах в дизайне ПО. Ответ на этот вопрос дает метрика *количество изменений (дн)* (`Number of Changes (d)`), где `d` — это период времени в днях. Например, метрика `Number of Changes (30)` показывает, как часто файл изменялся за последние 30 дней.

Отток кода

Метрика *изменчивость кода (дн)* (`Code Churn (d)`) показывает, сколько строк было добавлено в файл или удалено из него за определенный период времени, где `d` — это также рассматриваемый период в днях. На основе этой метрики можно также рассчитать *коэффициент изменчивости кода (дн)* (`Code Churn Rate (d)`), разделив значение `Code Churn(d)` на общее количество строк в файле. Допустим, `Code Churn(90)`=2. Это можно интерпретировать так: за последние 90 дней файл был переписан дважды. Эта метрика более информативна, чем простое количество изменений, так как она учитывает фактическое число измененных строк. Ее также можно использовать для выявления нестабильных элементов в дизайне ПО.

Количество авторов

Метрика *количество авторов (дн)* (`Number of Authors(d)`) сообщает, сколько разных пользователей зафиксировали изменения в данном файле за определенный промежуток времени (в днях). Это весьма показательно, поскольку помогает выявить «монополию на знания». Например, все

файлы, у которых значение метрики `Number of Authors(365)` равно 1, — это файлы, в которые в течение последнего года вносил изменения только один человек. Велика вероятность, что этот сотрудник — единственный, кто досконально знает этот файл. Это может нести риск для компании, если такой специалист решит уволиться.

Использование метрик системы контроля версий для поиска кандидатов на рефакторинг

Как я уже отмечал в начале этой главы, большинство проектов в той или иной степени подвержены структурной эрозии. Один из ее симптомов заключается в том, что изменения часто приводят к сбоям в, казалось бы, никак не связанных с ними частях системы. Чаще всего подобные проблемы возникают в сложных и часто изменяемых файлах. Есть неплохой шанс исправить ситуацию, если выявить такие проблемные точки и продумать, как снизить их сложность через рефакторинг кода. Часто оказывается, что эти «горячие точки» также являются «классами — бутылочными горлышками», то есть классами с большим количеством входящих и исходящих зависимостей.

Новейшие методы визуализации программной архитектуры значительно упрощают эту задачу. Так, визуализация Apache Cassandra в виде «программного города» (software city), показанная на рис. 9.9, использует трехмерное представление для одновременного отображения нескольких метрик.



Рис. 9.9. Программный город, визуализированный с помощью Sonargraph-Architect

Каждое здание в этом городе представляет собой один исходный файл. Исходные файлы сгруппированы по модулям, пакетам или пространствам имен. Площадь основания каждого здания пропорциональна количеству строк кода в файле (`LoC`). Высота здания определяется средней сложностью файла, а оттенок — частотой его изменения за последние 90 дней. Для высоких и темных зданий стоит провести рефакторинг. Взгляните на более темное по сравнению с соседними здание в левой части рис. 9.9. Оно не

очень высокое, но при этом является третьим по величине файлом (по значению LoC), что довольно легко заметить на этой визуализации. Темный цвет указывает на частые изменения. Оказывается, что этот файл содержит класс StorageManager, очевидно, важный для базы данных NoSQL.

Преимущество такой визуализации в том, что она позволяет сочетать произвольные пары метрик: например, высота зданий может соответствовать количеству входящих зависимостей, а цвет — определяться сложностью. Это значительно облегчает сложную аналитику.

Другие полезные метрики

Есть еще две метрики, которые я считаю полезными, но они не относятся к метрикам связанности и сложности.

Ранг компонента

Метрика *ранга компонента* (Component Rank) основана на метрике Page Rank от Google. Алгоритм Page Rank предназначен для поиска популярных страниц в интернете, в то время как Component Rank использует тот же алгоритм для поиска «популярных» классов в системе. Алгоритм Page Rank сначала выбирает случайную страницу. Затем с настраиваемой вероятностью (по умолчанию 80 %) он переходит по случайной исходящей ссылке на другую страницу. На конечной странице алгоритм останавливается и увеличивает для нее счетчик. Цель алгоритма — для каждой страницы вычислить вероятность того, что она окажется последней. Это достигается многократным запуском алгоритма до тех пор, пока значения вероятности для каждой страницы не стабилизируются.

Тот же самый алгоритм можно применить к классам или исходным файлам. Вместо ссылок используются исходящие зависимости. Вы можете спросить, что полезного в этой информации? Предположим, вы новичок в проекте и вам нужно добавить новую функциональность в сложный модуль. Вы никогда раньше не видели его код. С чего начать его изучение? Оптимально — с классов, у которых самый высокий ранг компонента, поскольку именно на них ссылаются многие другие классы.

На рис. 9.10 показано, как вычислить вероятность того, что каждая вершина графа станет конечной в сеансе обхода, который начинается со случайной вершины, а затем с вероятностью 80 % продолжается переходом по случайной ссылке и с вероятностью 20 % — завершается.

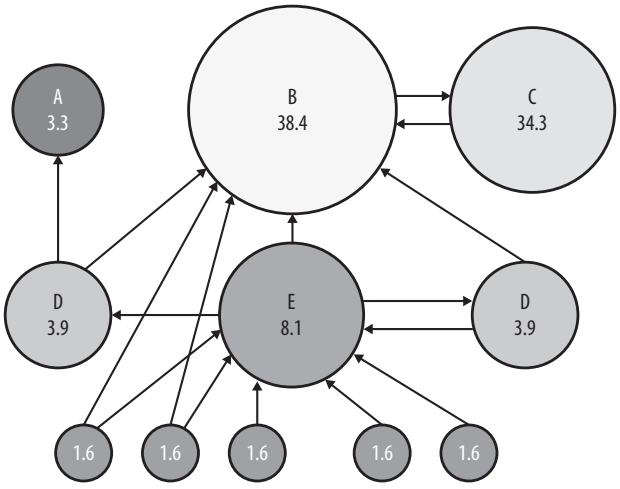


Рис. 9.10. Визуализация алгоритма Google Page Rank

LCOM4

`LCOM` — это аббревиатура от `Lack of Cohesion of Methods` (нехватка связности методов), а число 4 обозначает версию этой метрики. Цель метрики — выяснить, нарушает ли класс принцип единственной ответственности (https://oreil.ly/Uq3A3). Для этого она строит граф зависимостей между всеми методами класса (кроме конструкторов, переопределенных или статических методов) и всеми полями класса (кроме статических полей). Значение метрики равно количеству подграфов (называемых *связанными компонентами*), которые не связаны между собой. В идеале это значение должно быть равно 1 для всех классов. Если значение больше 1, класс легко разделить на несколько меньших классов.

На рис. 9.11 в классе два связанных компонента: один включает x, f и g, а другой — h и y. Это значит, что данный класс легко разделить на два разных класса: один будет состоять из f, g и x, а другой — из h и y.

Следует иметь в виду, что эта метрика часто дает сбой, когда класс вызывает методы или обращается к полям из суперкласса. Она плохо работает в иерархиях классов. С другой стороны, если класс не является частью сложной иерархии, эта метрика довольно хорошо находит классы, которые выполняют слишком много задач одновременно и, следовательно, нарушают принцип единственной ответственности.

```
class A
{
  private int x;
  private int y:

  public int f()
  {
    return x;
  }

  public int g()
  {
    return 2+f();
  }

  public int h()
  {
  return y;
  }
}
```

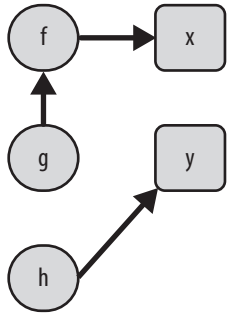


Рис. 9.11. Пример класса со значением LCOM4, равным 2

Архитектурные фитнес-функции

Понятие архитектурных фитнес-функций было впервые введено в книге «Building Evolutionary Architectures» Нила Форда, Ребекки Парсонс и Патрика Куа. Они определяют архитектурную фитнес-функцию как «объективную функцию, используемую для оценки того, насколько разрабатываемое решение достигает поставленных целей»[1]. (См. главы 2 и 8 этой книги.)

Архитектурные характеристики, также известные как атрибуты качества или разного рода «-ости», — это цели, которых стремятся достичь при проектировании архитектуры. К ним относятся, например, стабильн*ость*, масштабируем*ость*, сопровождаем*ость* и гибк*ость*. Фитнес-функции показывают, насколько архитектура соответствует одной или нескольким таким характеристикам. Например, для оценки масштабируемости можно использовать данные из продакшена, такие как количество одновременных пользователей и среднее время отклика, а для оценки сопровождемости кода — метрики относительной цикличности (`Relative Cyclicity`) и уровня сопровождаемости (`Maintainability Level`).

[1] Форд Н., Парсонс Р., Куа П. «Эволюционная архитектура. Автоматизированное управление программным обеспечением». СПб., издательство «Питер». Глава 2 «Фитнес-функции».

Одна из ключевых задач программного архитектора — умение находить компромиссы. Невозможно реализовать все желаемые характеристики в одной программной системе, не доведя ее сложность до неуправляемого уровня. Поэтому необходимо расставить приоритеты среди желаемых характеристик так, чтобы они лучше всего отвечали целям бизнеса и не переусложняли систему. Ситуация усугубляется тем, что некоторые из этих характеристик являются взаимоисключающими. Например, максимальная производительность и максимальная безопасность — это противоположные цели, поскольку для обеспечения безопасности требуется шифрование, которое, в свою очередь, требует значительных вычислительных мощностей процессора. Следует найти оптимальный баланс между ними.

В конечном счете рекомендуется выделять в качестве приоритетных не более трех характеристик плюс сопровождаемость. Существует очень мало сценариев, в которых последняя не имеет значения. Для оценки каждой из этих характеристик следует использовать подходящую фитнес-функцию.

В качестве фитнес-функций для оценки сопровождаемости, которая включает в себя и легкость понимания, можно применять некоторые из метрик, рассмотренных в этой главе. Например:

- уровень сопровождаемости (`Maintainability Level`) с пороговым значением 75 % и выше;
- относительная цикличность (`Relative Cyclicity`) на уровне пакетов/пространств имен и на уровне компонентов с пороговым значением не более 4 % для компонентов и 0 % для пакетов/пространств имен;
- индекс структурного долга (`Structural Debt Index`) для компонентов с пороговым значением в пределах нескольких сотен.

Можно использовать метрики сложности, чтобы определить, какой процент файлов с исходным кодом считать сложным. Например, каждый файл, у которого средний уровень вложенности превышает 3, средняя сложность — более 10, а размер — более 800 строк кода. Затем можно суммировать количество строк кода (`Lines of Code`) в сложных файлах и сопоставить его с общим количеством строк кода в системе. Можно установить правило, согласно которому доля сложного кода не должна превышать 10 %.

Как видите, метрики могут быть мощным инструментом, если применять их правильно. Используя знания о метриках, можно объединить несколько из них в одну полезную фитнес-функцию. Проверка с помощью фитнес-функций в пайплайне непрерывной интеграции (CI) и остановка сборки в случае, если заданные ими требования не выполняются, — это

эффективный способ гарантировать, что система никогда не превратится в пресловутый «большой ком грязи».

Как отслеживать динамику метрик

Чтобы реализовать цикл обратной связи на основе метрик, необходимо отслеживать их динамику. Лучший способ — ежедневно собирать метрики в ходе автоматической сборки и передавать их в инструмент, который будет вести их учет.

Получив данные о динамике изменений, вы можете представить их на графике. Например, на рис. 9.12 показано, как за последние 90 дней выросла метрика `Lines of Code`. Рекомендуется отслеживать все фитнес-функции в сочетании с метриками связанности и размера. Графики ключевых метрик позволяют выявлять нежелательную динамику уже на ранней стадии и реагировать, пока ситуация не усугубилась. Конечно, того же можно добиться, установив для фитнес-функций жесткие пороговые значения, но часто бывает полезно видеть, как метрики меняются с течением времени. (При нарушении жесткого порога сборка прерывается. При нарушении мягкого порога, в свою очередь, выдается лишь предупреждение.)

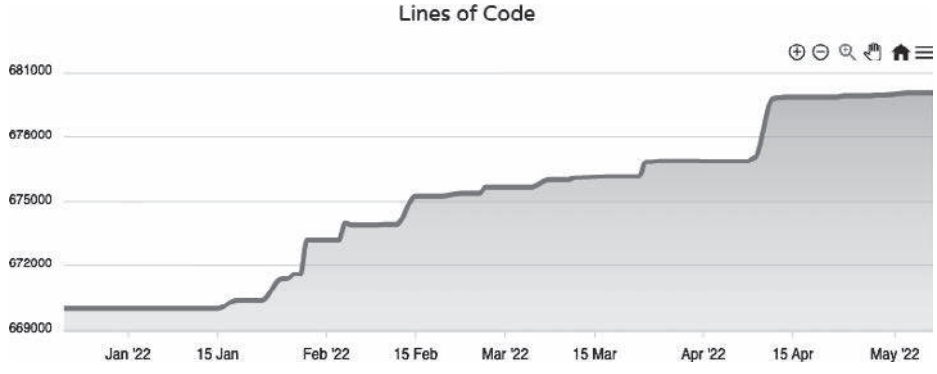


Рис. 9.12. График динамики метрик, построенный в Sonargraph-Enterprise

Существует несколько инструментов, в том числе бесплатные. SonarQube (http://sonarqube.org) бесплатен для некоторых языков, но дает ограниченный набор метрик. Дополнительные метрики доступны в плагине Sonargraph для SonarQube. Можно использовать Jenkins (http://jenkins.io) вместе с бесплатным Sonargraph-Explorer; в этом варианте хороший набор метрик, но ограниченные возможности для построения графиков. Sonargraph-Enterprise поставляется с коммерческой командной лицензией Sonargraph.

Этот инструмент предлагает большой выбор метрик, а также возможность строить гибкие и настраиваемые графики. Наконец, при наличии хорошего источника данных не так уж сложно создать и собственное решение.

Несколько золотых правил для улучшения ПО

Далее я поделюсь несколькими своими золотыми правилами, которые помогают остановить структурное разрушение и обеспечить модульный и легко обслуживаемый дизайн продуктов. Если начинать новый проект, придерживаясь этих правил, то создаваемый продукт будет лучше, чем 90 % всех остальных проектов сопоставимого размера и сложности. Если вы начинаете работать с уже существующим кодом, первоочередная цель — сделать так, чтобы ситуация не становилась хуже. Затем можно установить ежемесячные или ежеквартальные цели по сокращению количества нарушений на несколько процентов. Со временем это даст накопительный эффект и значительно упростит обслуживание и понимание кода.

Вот эти правила:

- Создайте формальную и обязательную к исполнению архитектурную модель, которая определяет различные части программного продукта и допустимые зависимости между ними. Это правило, по сути, является логическим следствием необходимости контролировать и минимизировать связанность, о чем я подробно говорил выше. Такая модель дает разработчику больше контекста и позволяет контролировать соблюдение дизайна архитектуры системы.
- Избегайте циклических зависимостей на уровне пространств имен и пакетов.
- Ограничивайте циклические зависимости на уровне исходных файлов и классов. Любая группа циклической зависимости, в которой больше пяти элементов, имеет все шансы превратиться в «рак кода» и разрастаться до тех пор, пока распутать этот клубок не станет очень сложно. По возможности следует избегать даже небольших групп циклических зависимостей.
- Избегайте дублирования кода (кодинга методом копирования и вставки). Я об этом не упоминал, поскольку глава посвящена метрикам, но писать код копированием и вставкой — это верный путь к тому, чтобы в нем появился запах (https://oreil.ly/xUtfn). Иногда дублирование кода можно использовать для разрыва циклических зависимостей, которые трудно устранить иным способом, но лишь в редких, исключительных случаях.

- Установите ограничение на размер исходных файлов в 800 `LoC` (мягкий порог).
- Ограничьте максимальный уровень вложенности четырьмя отступами, а модифицированную цикломатическую сложность `Modified Cyclomatic Complexity` — значением 15 (мягкие пороги).

Все упомянутые правила можно внедрить, используя знания, полученные из этой главы. Лучший способ это сделать — использовать инструменты, автоматически проверяющие правила в CI-пайплайне. Также целесообразно внедрить такие правила в масштабах всей организации, чтобы их знали все разработчики. Это практически гарантирует повышение сопровождаемости, качества кода и, как следствие, значительный рост производительности разработчиков. Я работал с несколькими клиентами, которые внедряли подобный подход на базе правил, и могу подтвердить, что со временем он дает весьма впечатляющие результаты.

Заключение

В этой главе вы познакомились с некоторыми полезными метриками, а также с концепцией цикла обратной связи, который основан на метриках. Теперь вы знаете, что структурный технический долг (или архитектурный долг) может серьезно снижать производительность команд разработчиков.

Я надеюсь, вы убедились, что метрики — это мощный инструмент, который помогает выявлять пагубную динамику на достаточно раннем этапе, чтобы программный проект совершенно точно не превратился в «большой ком грязи». Разумеется, для этого потребуется использовать инструменты, и к счастью, некоторые из них доступны совершенно бесплатно. Если вы еще не готовы широко применять метрики, я очень рекомендую постараться предотвратить или, по крайней мере, ограничить условия для образования циклических зависимостей в коде. Один этот шаг сам по себе позволит остановить распространение худших побочных эффектов структурной эрозии и облегчит внедрение более строгих правил в дальнейшем.

ГЛАВА 10

Измерение неизвестного с помощью подхода «цель — вопрос — метрика»

Майкл Килинг

В программной разработке, как и в жизни, измерить самое важное зачастую сложнее всего. Каков объем технического долга в системе и куда следует направить ресурсы? Соответствует ли реализованная архитектура наиболее важным атрибутам качества и в какой степени? Как со временем меняется зрелость команды в вопросах проектирования? Когда речь заходит о таких масштабных вопросах, легко поддаться искушению и дать интуитивный ответ, однако такие оценки бывают предвзяты и ненадежны. Гораздо эффективнее опираться на данные. Когда передо мной встает задача определить метрики для подобных запутанных и сложных проблем, я использую подход «цель — вопрос — метрика» (goal-question-metric, GQM).

Подход GQM — это аналитический метод, который предложили Виктор Базили (Victor Basili) и Дэвид Вайсс (David Weiss) (https://oreil.ly/SMvwj), чтобы помочь командам разработчиков определить способы измерения и оценки сложных проблем программной разработки[1]. Этот метод просто изучить и применять как индивидуально, так и в группе. Поиск подходящих метрик для оценки сложной проблемы по-прежнему требует творческого подхода и аналитического мышления, однако GQM обеспечивает ровно тот минимум структуры, который необходим, чтобы направить усилия команды на достижение более высоких результатов.

---

[1] Basili V. R., Weiss D. M. «A Methodology for Collecting Valid Software Engineering Data», в IEEE Transactions on Software Engineering SE-10, № 6 (ноябрь 1984 г.): 728–738, doi: 10.1109/TSE.1984.5010301.

Из этой главы вы узнаете, как использовать подход GQM и как проводить коллективный воркшоп, построенный на его принципах. Вы также на практическом примере увидите, как команда разработчиков применила GQM для оценки и последующего устранения недостатков в архитектуре своего продукта. Прочитав эту главу до конца, вы будете обладать всеми необходимыми знаниями, чтобы применить подход GQM в своей команде.

Подход «цель — вопрос — метрика»

Основной принцип GQM-подхода прост: чтобы проводить измерение качественно, необходимо понимать, зачем это делается. Понимание этого «*зачем*» — то есть целей, которых требуется достичь, и вопросов, на которые нужно получить ответы для оценки динамики, — позволяет определить и выбрать наиболее подходящие для решения задачи метрики. Команды, которые понимают, *зачем* им та или иная метрика, с большей вероятностью будут использовать эту метрику и опираться на нее при принятии решений в будущем. GQM-подход помогает преобразовать цели из расплывчатых формулировок о намерениях в модели, которые поддаются количественной оценке и проверке.

Модель GQM имеет иерархическую структуру. Если представить эту модель в виде дерева, то в его корне будет находиться цель. От цели, как от корня, отходят ветви-вопросы, от которых, в свою очередь, отходят ветви-метрики, которые используются для ответа на эти вопросы. Листьями такого дерева являются данные, которые используются для расчета значений метрик. Таким образом, GQM-подход обеспечивает прослеживаемость: можно проследить путь от любого отдельного листа (то есть от собираемых данных) назад к корню (то есть к изначальной цели сбора этих данных).

Создание GQM-дерева

В рамках GQM-подхода *цель* — это краткая формулировка, которая описывает, что именно необходимо понять и оценить с помощью метрик. В идеале формулировка цели описывает ее назначение, объект для получения метрик, рассматриваемую проблему или тему, а также точку зрения, с которой эта цель рассматривается.

Вот несколько примеров целей, которые ориентированы на решение задач программной архитектуры:

- повысить доступность системы с точки зрения пользователей;
- сократить время разработки новых микросервисов с точки зрения менеджеров по продукту;
- уменьшить технический долг в архитектуре с точки зрения разработчиков;

- сократить количество ошибок, которые попадают на продакшен;
- обнаруживать больше проблем на продакшен до того, как их заметят пользователи;
- повысить точность модели машинного обучения с точки зрения пользователей;
- принимать более качественные проектные решения в архитектуре с точки зрения команды разработчиков.

Определение цели на начальном этапе — это ключевой момент всего процесса. Цель задает концептуальное направление, концентрируя внимание на определенном наборе метрик. Целевым может стать любой объект, в том числе элементы архитектуры, процессы программной разработки, технические эксперименты, артефакты дизайна и даже команды и организации. Указывая объект для оценки, мы получаем возможность согласовать, что именно нужно измерять, с тем, зачем это нужно делать.

Цель часто пересматривается в ходе анализа, по мере того как углубляется понимание объекта измерения. Даже несовершенная формулировка цели, которая отражает лишь суть того, что необходимо измерять, будет полезна.

Как только цель определена, можно задавать *вопросы* для ее исследования и описания. Вопросы должны быть ориентированы на операционную деятельность и помогать оценивать динамику достижения цели: приближаетесь вы к ней или отдаляетесь?

Хорошо сформулированные вопросы могут выявить проблемы и подсказать следующие шаги для их решения. Чтобы задавать правильные вопросы, нужен пытливый ум. Нужно быть готовым на время отбросить практичность и задать те вопросы, которые действительно важны, даже если пока неясно, как на них ответить.

Вот несколько примеров вопросов, которые могут помочь оценить, зачем повышать доступность системы с точки зрения пользователей:

- Каков текущий показатель доступности?
- Какие компоненты или сервисы имеют наилучшие показатели доступности? А наихудшие?
- Какие компоненты или сервисы выходят из строя чаще всего?
- Почему компоненты или сервисы становятся недоступными?
- Какова продолжительность типичного сбоя?
- Когда происходят сбои?

Чтобы ответить на каждый вопрос, нужно определить одну или несколько *метрик*. Метрики могут принимать разные формы, в том числе простых шкал оценок, булевых значений (да/нет, истина/ложь), статистических выводов и уравнений различной сложности. Любая метрика, которую планируется использовать, в конечном счете потребует четкого и точного определения. Каждая метрика связана как минимум с одним вопросом. Одна и та же метрика иногда может применяться для ответа на несколько вопросов. Для ответа на один вопрос может потребоваться несколько метрик.

Собрав все воедино, можно построить GQM-дерево, подобное тому, что приведено на рис. 10.1.

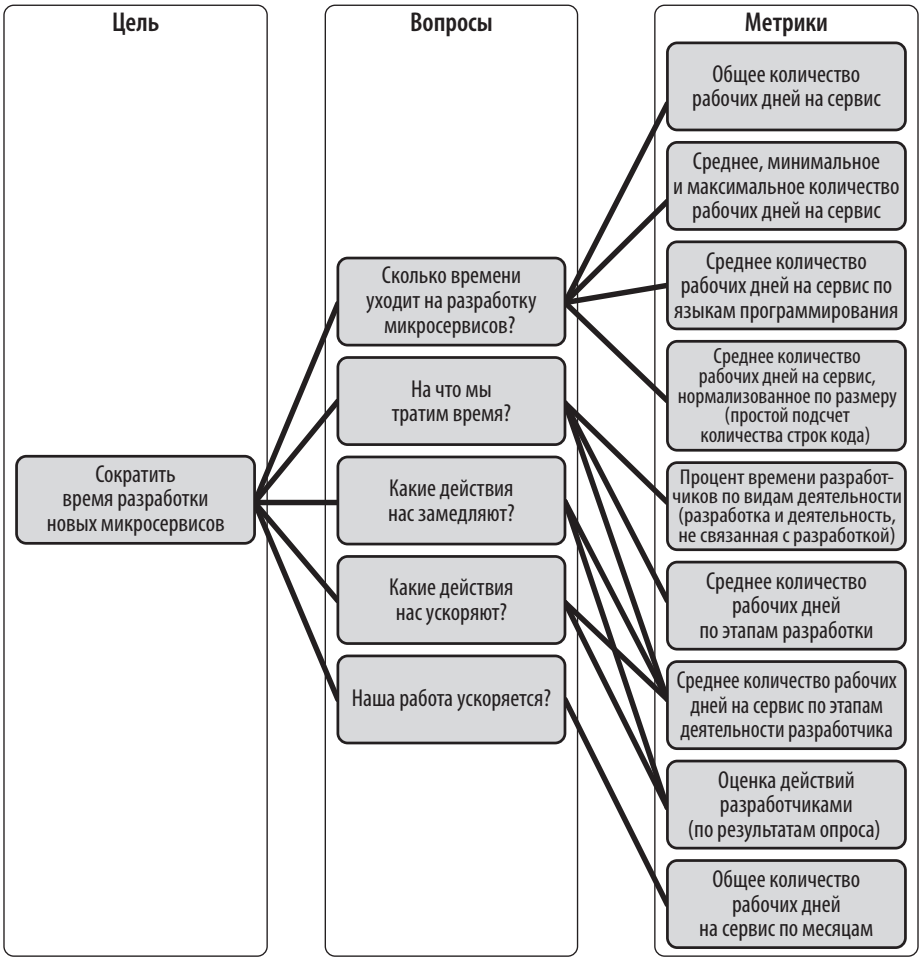


Рис. 10.1. Пример GQM-дерева

Мне нравится в этом подходе то, что его название говорит само за себя: всего три шага. Определить *цель*. Сформулировать *вопросы*, которые помогут оценить продвижение к цели. Определить *метрики*, которые помогут ответить на эти вопросы.

Хотя название говорит *почти* все. После определения метрик необходимо решить, как собирать данные для их расчета.

Расстановка приоритетов для метрик и разработка стратегии сбора данных

Недостаточно просто знать, как вы *могли бы* измерить продвижение к цели. Нужно еще и спланировать, как именно вы *будете* это делать. Если вы хорошо провели мозговой штурм, формулируя вопросы и метрики, то ваше GQM-дерево должно получиться достаточно ветвистым. На практике не все метрики дают четкий сигнал; а измерение по каким-то метрикам может оказаться нецелесообразным или дорогим. Прежде чем принимать решение о способах сбора данных, рекомендуется немного «подрезать» ветви GQM-дерева.

Лучше всего сосредоточиться на тех метриках, которые дают наиболее четкий сигнал и измерение по которым при этом дешево. В первую очередь определите ключевые метрики, дающие самые четкие сигналы. Затем выявите метрики, которые можно использовать для получения ответа сразу на несколько вопросов. Если для ответа на некоторые вопросы нужно несколько метрик, внимательно проанализируйте, действительно ли нужны они все. Помните, что для объективности полезно учитывать как позитивные, так и негативные метрики, то есть те, которые свидетельствуют об успехе, и те, которые указывают на неудачу.

После этого проанализируйте, какие данные требуются для расчета метрик. Если какую-то метрику вы еще недостаточно точно определили, самое время это сделать. Дополните GQM-дерево (как показано на рис. 10.2), чтобы наглядно представить, где будут использоваться данные. Чем больше метрик использует тот или иной фрагмент данных, тем этот фрагмент ценнее. На рис. 10.3 показан пример расширенного GQM-дерева, в котором данные увязаны с метриками. Приоритетом должны стать данные, которые необходимы для расчета наиболее ценных метрик и которые при этом дешево собирать.

В зависимости от того, что именно измеряется, данные могут поступать из множества источников. Может потребоваться собирать и записывать данные о коде. Данные о процессах и методах разработки можно получать из опросов или извлекать из баз данных управления задачами. Данные о коде можно почерпнуть из репозиториев с исходным кодом или извлечь

при помощи инструментов статического анализа. Для проведения коротких и быстрых экспериментов иногда проще и дешевле всего в течение нескольких дней собирать данные вручную. Главное — точно знать, откуда будут приходить данные, которые требуются для расчета метрик.

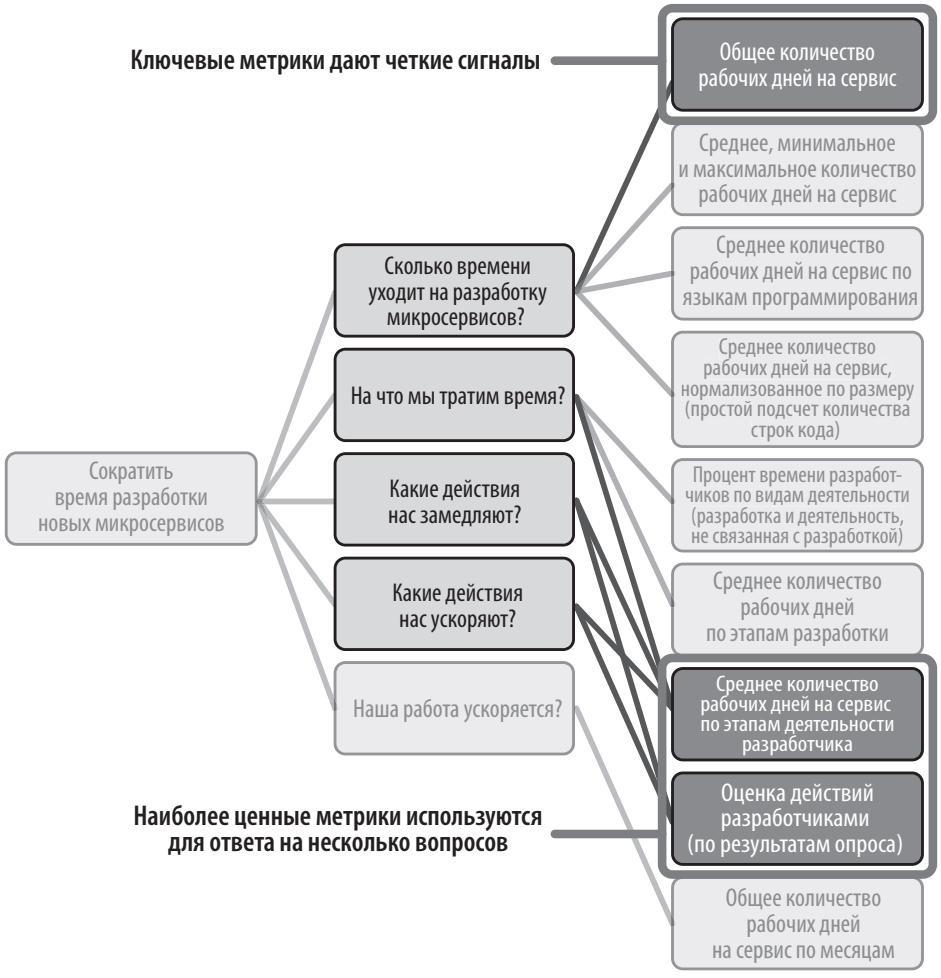


Рис. 10.2. Ключевые метрики, которые дают четкие сигналы и могут использоваться для ответа на несколько вопросов

На этом этапе информации уже достаточно, чтобы составить конкретный план сбора данных и расчета метрик. В зависимости от поставленной

цели может и не потребоваться рассчитывать абсолютно все метрики или давать ответы на все вопросы. Определите, какие именно действия предстоит выполнять, например: регистрировать и собирать данные, рассчитывать метрики и создавать дашборды. Обсудите этот план с командой и всеми остальными стейкхолдерами, а затем приступайте к его реализации.

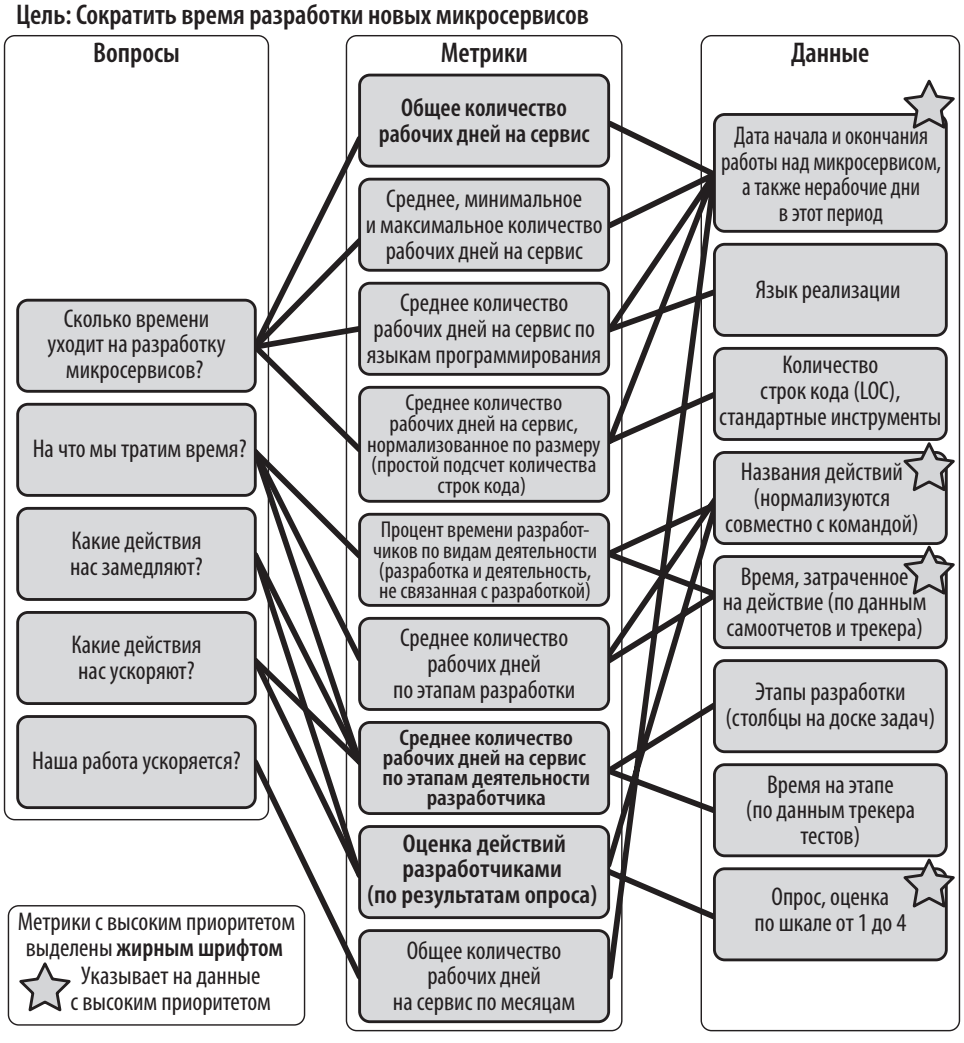


Рис. 10.3. Расстановка приоритетов для метрик и данных в GQM-дереве

Кейс: команда, которая научилась видеть будущее

Итак, мы разобрали основные принципы, которые лежат в основе GQM-подхода. Рассмотрим теперь конкретный практический пример, чтобы показать, как применять этот подход в реальной работе.

В этом примере вы узнаете о двух сбоях в работе сервиса, с которыми столкнулась одна из команд разработчиков. В ходе ретроспективного анализа (постмортема) первого инцидента команда применила GQM-подход, чтобы выявить метрики, которые позволили бы ей узнать о происшествии заблаговременно. После анализа они повысили прозрачность операций и внесли важные изменения в архитектуру. Девять месяцев спустя, когда возникла похожая проблема, эти метрики прошли настоящую проверку на практике. На этот раз, благодаря внесенным изменениям, команда узнала о проблеме раньше пользователей и с легкостью превратила потенциально серьезный отказ системы в кратковременное и незначительное неудобство.

Контекст системы

Система, рассматриваемая в этом примере, использует сторонние сервисы для выполнения некоторых операций с данными. Часть этих сервисов устанавливают ограничения по частоте запросов к API. В архитектуре применяются очереди. Они гарантируют, что данные в конечном итоге будут обработаны сторонними сервисами, а также позволяют управлять объемом запросов к этим сервисам. В рассматриваемой истории технологии не так важны, как архитектурные паттерны. На рис. 10.4 показана контекстная диаграмма соответствующих частей архитектуры.

В этой системе важный сторонний сервис, который я буду называть сервисом Foo (название вымышленное), устанавливает ограничение на количество запросов к API, которое определено в лицензионном соглашении. Когда количество запросов к сервису Foo превышает это значение, Foo отклоняет запросы и возвращает ответ «превышен лимит запросов» (rate limit exceeded). Последующие запросы будут отклоняться до тех пор, пока расчетная частота запросов не опустится ниже согласованного порога. Устанавливаются как почасовые, так и суточные ограничения. Также существуют лимиты на размер запросов и общую вычислительную нагрузку на сервис Foo.

Важной частью архитектуры являются асинхронные очереди задач. Помимо управления взаимодействием со сторонними сервисами, такими как Foo, они управляют рабочими процессами и другими задачами по анализу данных. Если очередь становится слишком большой, это может негативно

отразиться на работе внутренних пользователей. Очереди спроектированы отказоустойчивыми — например, они могут повторно отправлять запросы к сторонним сервисам в случае определенных сбоев. При временном сбое система постепенно приходит в корректное и согласованное состояние, если очереди и их обработчики продолжают работать. Самовосстановление — важное свойство системы, особенно в случае с сервисом Foo, поскольку лимиты легко превысить, но они быстро сбрасываются.

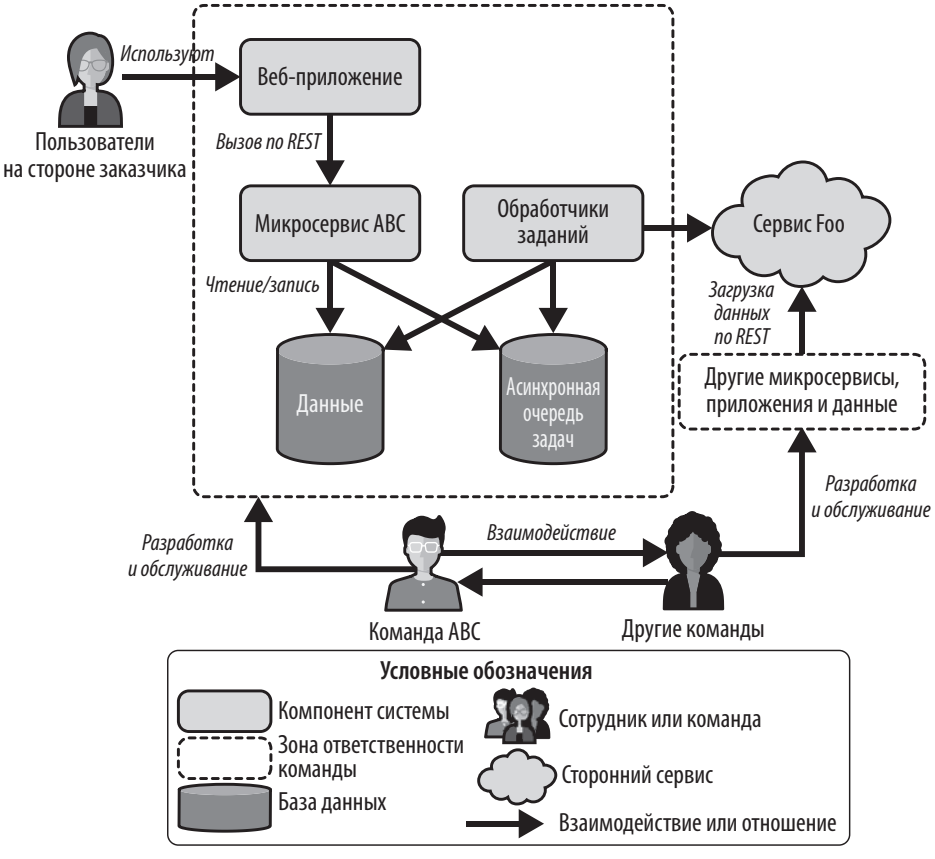


Рис. 10.4. Контекстная диаграмма системы

Инцидент № 1: слишком много запросов к сервису Foo

Ранним утром в понедельник сервис Foo начал отклонять запросы из-за превышения лимита на частоту запросов к API. Поскольку лимит запросов к сервису Foo является общим для всей программной системы, это затронуло сразу несколько компонентов. В течение нескольких часов

время нахождения каждой задачи в очереди на обработку резко возросло, и пользователи ощутили это на себе.

Команда быстро выявила проблему. Каждые выходные в Foo загружается большой пакет данных, чтобы в понедельник утром он был доступен внутренним пользователям. Сбой в работе хранилища в сочетании с ошибкой в пакетной операции привел к ситуации, когда пакетная операция начала агрессивно повторять запросы к API сервиса Foo, что в итоге и привело к превышению лимита. Остановка этой вышедшей из-под контроля пакетной операции позволила системе постепенно восстановиться. После того как команда починила хранилище, пакетная операция успешно завершилась.

Во время разбора этого инцидента команда разработчиков решила, что исправление ошибки в пакетной операции — очевидная и важная задача. Они задались вопросом: можно ли было узнать о проблеме со связью с сервисом Foo раньше? Возможно ли было заметить приближение потенциального инцидента и устранить его до того, как он затронет пользователей? Чтобы ответить на эти вопросы, команда решила прибегнуть к подходу GQM.

Как выглядит сбой?

Если бы команда узнала о проблеме раньше, она, возможно, смогла бы отреагировать быстрее и избежать негативных последствий для пользователей. Команде необходимо было придумать способ, как лучше выявлять потенциальные сценарии сбоев, чтобы быстрее реагировать на возможные отказы в обслуживании.

Цель была проста: обнаруживать проблемы, связанные с сервисом Foo, на ранней стадии, а затем смягчать их последствия или полностью устранять до того, как они начнут негативно сказываться на пользователях. В идеале пользователи должны считать, что команда инженеров — это всемогущие волшебники, которые умеют предугадывать будущее.

Держа эту цель в уме, команда провела мозговой штурм для определения вопросов и метрик, которые помогли бы до этой цели дойти, — а это всегда довольно сумбурный процесс. Вопросы и метрики менялись и развивались по мере того, как команда разбиралась, что на самом деле означает «предвидеть сбои» в работе с сервисом Foo. В течение примерно получаса члены команды набрасывали и развивали свои идеи в общем документе, куда собирали все возникающие вопросы.

В табл. 10.1 кратко изложены результаты работы команды.

Таблица 10.1. Итоги мозгового штурма по метрикам

Цель: обнаруживать проблемы при использовании сервиса Foo, чтобы команда инженеров могла смягчить их последствия или устранить их до того, как они начнут отражаться на пользователях.	
Вопросы	**Метрики**
Сколько сейчас совершается вызовов API сервиса Foo?	Количество вызовов API по данным сервиса Foo
Насколько мы близки к превышению лимита по частоте запросов к сервису Foo?	Оставшееся количество вызовов API (общая доступная квота минус зарегистрированное количество вызовов)
	Процент оставшейся квоты API для всех компонентов (зарегистрированное количество вызовов/общая квота)
	Процент оставшейся квоты API для каждого компонента (наблюдаемое количество вызовов по компоненту/квота, выделенная компоненту)
При подключении к сервису Foo проблема на его стороне или на нашей?	Успешный сеанс проверки связи (логическая метрика с допуском ошибок для устранения кратковременных сбоев)
	Синтетический трафик синхронизируется с сервисом Foo, как и ожидалось (логическая метрика)
	Процент запросов, завершившихся с тайм-аутом, за 15-минутный интервал
	Процент запросов с ошибкой аутентификации за 15-минутный интервал
Корректно ли выполняются задания?	Общее количество запросов
	Количество штатных ответов, ответов с ошибкой и с тайм-аутом
	Процент ответов с ошибкой за 15-минутный интервал
Справляемся ли мы с текущей нагрузкой по запросам?	Глубина очереди заданий (количество ожидающих и выполняемых заданий)
	Средняя глубина очереди в динамике
	Среднее время, p99 и p95 обработки задания
	Средняя пропускная способность обработки заданий (количество заданий в единицу времени)

Теперь, когда команда разработчиков поняла, какие именно метрики необходимы для выявления проблем в работе сервиса Foo, она занялась организацией сбора данных. На этом этапе в архитектуре обнаружилось несколько недостатков. Во-первых, сбор данных осуществлялся только в том случае, когда система находилась под нагрузкой. Во-вторых, в архитектуре не было четко определено, какой компонент отвечает за обработку сбоев и повторные попытки. В-третьих, члены команды не имели четкого понимания, как реагировать на возникающие проблемы.

Прозрачность операций и улучшения архитектуры

Первая проблема была связана непосредственно с тем, как в системе был реализован сбор данных. При отсутствии трафика запросы к сервису Foo не поступали. А без запросов было невозможно определить, работает ли Foo штатно. Хотя отказ Foo в момент, когда сервис никем не использовался, не приводил к ущербу, разработчики хотели заблаговременно информировать пользователей о потенциальных проблемах. Очевидно, что команда не могла исправить сам сервис Foo, но она могла использовать полученную информацию для прогнозирования других потенциальных сбоев в подконтрольной ей системе.

Чтобы устранить этот пробел в сборе данных, команда добавила в архитектуру новый компонент, который отслеживал доступность сервиса Foo. К счастью, Foo предоставляет специальный API измерения, с помощью которого клиенты могут проверять текущее использование API и подтверждать доступность сервиса. Дополнительная информация, которую предоставлял этот API, упростила управление общим бюджетом вызовов API.

Далее команда уточнила архитектурное решение, определив, что за обработку сбоев должна отвечать очередь задач (work queque), а не сами задачи (jobs). В предыдущем варианте архитектуры этот вопрос оставался открытым. В результате в некоторых задачах неуспешные запросы к сервису Foo отправлялись повторно, что еще больше усугубляло последствия инцидента.

Во время простоя задачи, в которых совершались попытки восстановиться после сбоя, выполнялись дольше. Они неизбежно завершались ошибкой и снова попадали в очередь для повторного выполнения, что лишь увеличивало перегрузку. В результате количество запросов к сервису Foo со временем многократно возросло. В худших случаях некоторые задачи делали до 5 неуспешных попыток обращения к Foo и до 10 перезапусков, прежде чем окончательно завершиться ошибкой, что в сумме давало 50 за-

просов к API! Команда по итогу обсуждения зафиксировала в *записи об архитектурном решении* (ADR, architecture decision record), что в задачах необходимо предусмотреть быстрый отказ.

С метриками команда получила все необходимое для формирования четкого плана действий. Члены команды добавили оповещения для автоматического мониторинга определенных метрик на продакшен. Для каждой метрики были разработаны сценарии реагирования (ранбуки), чтобы все сотрудники знали, как действовать при возникновении потенциальной проблемы. Каждый ранбук содержал ссылки на метрики, что упростило диагностику проблем и исключило ложные срабатывания. Также были созданы инструменты и добавлены диагностические API, помогающие восстановить сервис.

Инцидент № 2: заглядывая в будущее

Некоторые члены команды сомневались в необходимости всей этой работы — ведь, в конце концов, первопричина была устранена. Примерно девять месяцев спустя они поняли, насколько полезными оказались найденные метрики и внесенные архитектурные изменения.

Ранним утром в пятницу разработчик сервиса Foo внес в конфигурацию изменение, которое вызвало полный отказ системы. В течение следующих 14 часов сервис Foo был полностью недоступен.

Но на этот раз команда была готова. В течение 10 минут после того, как Foo стал недоступен во всем мире, она получила оповещение по одной из ранее определенных метрик. Благодаря новым диагностическим API команда быстро убедилась, что проблема возникла на стороне сервиса Foo и находилась вне зоны ее контроля. Разработчики отключили несколько оповещений и продолжили отслеживать метрики, чтобы убедиться, что повторные попытки выполнения неуспешных задач происходят с экспоненциальной задержкой, согласно ADR. Все шло по плану.

Вскоре после начала рабочего дня, еще до того, как хотя бы один пользователь заметил проблему, команда сделала электронную информационную рассылку по внутренним пользователям. То, что девятью месяцами назад стало бы критической ситуацией с наивысшим приоритетом, теперь превратилось в едва заметное событие (по крайней мере, для этой команды!). Когда сервис Foo возобновил работу, система самостоятельно восстановилась в соответствии с дизайном, а команда продолжила отслеживать метрики, пока в течение следующих нескольких часов все не пришло в норму.

Выводы

На практическом примере вы увидели, как команда применила подход «цель — вопрос — метрика» (GQM), чтобы обосновать изменения в архитектуре системы, которые помогли ей эффективнее реагировать на серьезные сбои. Метрики, которые команда определила в ходе этого процесса, легли в основу ее стратегии обеспечения прозрачности операций и реагирования на инциденты. Кроме того, пристальное изучение метрик и данных, которые требуются для их расчета, выявило пробелы в архитектуре. Команда добавила новый компонент, который собирает необходимые данные, и уточнила, какой элемент архитектуры должен следить за повторными попытками выполнения операций.

Часто в ходе разборов инцидентов выявляются слабые места в стратегиях обеспечения прозрачности операций и мерах реагирования. Как вы уже убедились, GQM можно использовать не только для того, чтобы выявить эти слабые места, но и для того, чтобы наметить путь создания более совершенной программной системы.

Проведение воркшопа по GQM

GQM — это превосходный инструмент, который позволяет анализировать сложные проблемы как в одиночку, так и в небольших группах. К нему также можно прибегать в формате структурированного группового воркшопа, в котором могут участвовать стейкхолдеры с разным опытом и знаниями[1]. В этом разделе вы научитесь проводить такие воркшопы.

Краткое описание воркшопа

Цель воркшопа — прийти к единому мнению и обеспечить совместное владение в отношении метрик и данных, которые необходимо вычислять и собирать для конкретной цели. К концу воркшопа все участники должны прийти к пониманию, почему необходимы те или иные метрики и как их рассчитывать.

Преимущества

Воркшоп повышает доверие к планам сбора метрик и данных, поскольку он подчеркивает, что в основе всего процесса получения метрик лежат

[1] См. мою книгу «Design It! From Programmer to Software Architect» (Pragmatic Bookshelf, 2017).

цели стейкхолдеров. Участие стейкхолдеров обеспечивает более широкую поддержку итогового плана сбора метрик и данных и способствует более вдумчивому их анализу.

Сам воркшоп дает возможность показать всей группе, как проводится структурированный анализ. Участники, которые выносят с воркшопа положительный опыт, зачастую стараются применять GQM-подход и в других ситуациях.

Участники

В воркшопе могут участвовать как технические, так и нетехнические специалисты. В зависимости от цели участие нетехнических специалистов может даже быть *обязательным*. Например, если на воркшопе предстоит исследовать цель, которая связана с конкретным бизнес-процессом, на нем должен присутствовать профильный эксперт. Аналогично, если цель касается запуска продукта, то в воркшопе должны участвовать представители отделов управления продуктом, маркетинга, дизайна и продаж. Среди участников всегда должен быть хотя бы один программный разработчик.

Подобный воркшоп наиболее эффективен в небольших группах от двух до пяти человек, но участников может быть и больше, тогда работу следует организовать в подгруппах. Не стоит забывать, что GQM также можно успешно применять самостоятельно!

Подготовка и материалы

Перед воркшопом подготовьте черновую формулировку цели, с которой начнется работа. Понимание цели, которую предстоит исследовать, также помогает определить, все ли необходимые участники приглашены.

Если воркшоп проводится очно, то все, что понадобится, — это большая маркерная доска и маркеры. Клейкие листочки (стикеры) необязательны, но их можно использовать для мозгового штурма при формулировании вопросов и поиске метрик.

Для удаленного воркшопа с распределенной командой предпочтительна виртуальная доска (например, Miro), но подойдет и любой документ для совместной работы, который все участники могут редактировать (например, Google Docs или Dropbox Paper). В крайнем случае можно обойтись и простой демонстрацией экрана, но это усложнит взаимодействие участников.

Результаты

По итогам воркшопа должны быть получены следующие результаты:

- Формулировка цели, с которой согласны все участники.
- Список вопросов, которые раскрывают суть цели.
- Приоритизированный список метрик с указанием вопросов, на которые они помогают ответить.
- Определения метрик (формальные или неформальные).
- Список данных, которые необходимы для расчета метрик.

Этапы воркшопа

Вот общие этапы проведения воркшопа:

1. Вначале произнесите вступительное слово и ознакомьте всех с основными правилами. Например: «Сегодня мы вместе поработаем над определением метрик, которые необходимы для оценки предстоящего запуска продукта. Пожалуйста, не забывайте на протяжении всего воркшопа относиться друг к другу вежливо, внимательно и уважительно».
2. Запишите формулировку цели и разместите ее так, чтобы она была видна каждому участнику. При работе с маркерной доской оставьте достаточно свободного места для добавления вопросов и метрик.
3. Предложите участникам сформулировать вопросы так: «На какие вопросы нам нужно ответить, чтобы понять, достигли ли мы этой цели?» Продолжайте собирать вопросы, пока не истечет отведенное время или у группы не кончатся идеи.
4. Выберите один из вопросов и предложите группе в режиме мозгового штурма подобрать метрики, которые помогут на него ответить. Фиксируйте предложенные метрики на видном для всех месте. Соедините линией метрику со всеми вопросами, на которые она помогает ответить. Помните, что это мозговой штурм. Приглашайте участников думать креативно и пока не переключать внимание на то, как рассчитать метрику или собрать данные. Продолжайте сбор идей по метрикам, пока не истечет отведенное время или у группы не иссякнут идеи.
5. Когда для каждого вопроса будет найдена хотя бы одна метрика, вернитесь к цели и выполните проверку на здравый смысл. Помогают ли эти метрики в оценке цели? Не нужно ли переформулировать цель? Есть ли новые цели, которые также следует рассмотреть? При необходимости скорректируйте формулировку цели.

6. Определите, какие данные нужны для расчета каждой метрики. Возможно, также потребуется скорректировать определения метрик.
7. Расставьте приоритеты для метрик. Это можно сделать несколькими приведенными ниже способами (но достаточно одного). Среди наиболее распространенных можно выделить выявление обязательных метрик; метрик, которые при максимальной отдаче и минимальных затратах дают ответы на несколько вопросов, а также точечное голосование за метрики и их сортировку по соотношению ценности и трудозатрат.
8. В завершение предоставьте участникам возможность поделиться своими мыслями и наблюдениями. Были ли какие-нибудь неожиданности? Достигнут ли консенсус в отношении наиболее важных метрик? Есть ли метрики, реализация которых кажется проблематичной или дорогой?
9. По окончании воркшопа зафиксируйте его итоги и разошлите их всем участникам. Если требуется, подготовьте в качестве домашнего задания отчет с описанием выводов, к которым пришла группа.

Рекомендации и советы по организации воркшопа

Вот несколько рекомендаций и советов по организации воркшопа:

- В процессе разработки вопросов и метрик участники могут использовать стикеры для мозгового штурма, размещая по одному вопросу на каждом стикере. Когда мозговой штурм подойдет к концу, попросите одного из участников зачитать вслух записи на стикерах. Прежде чем двигаться дальше, сгруппируйте их и удалите дубликаты.
- Разработка метрик может оказаться непростой задачей! Если группа зайдет в тупик, поддержите нестандартное мышление, предложив участникам примерно следующее: «Давайте пока не думать, как мы будем получать данные. Когда мы поймем, какие метрики нам нужны, мы придумаем, как их вычислять».
- Ищите возможности повторно использовать метрики или данные. Одни и те же метрики могут дать ответ на несколько вопросов, а одни и те же данные могут применяться для вычисления нескольких метрик.
- Когда речь идет о вопросах, связанных с системой, ее архитектура, скорее всего, будет влиять на возможность собирать данные. Чтобы оценить затраты на сбор данных, в воркшопе должен участвовать специалист, который хорошо разбирается в архитектуре.
- Не забудьте сфотографировать GQM-дерево! Это быстрый и простой способ донести суть GQM-анализа.

Пример

В этой главе вы уже видели несколько примеров GQM-деревьев, а также целей, вопросов и метрик. На рис. 10.5 показано GQM-дерево, которое было создано во время очного GQM-воркшопа. Цель этого воркшопа заключалась в том, чтобы определить, какие аналитические инструменты можно было бы использовать для пометки записей в целях расследования мошенничества. Обратите внимание, что на этом этапе GQM-дерево получилось довольно запутанным!

После воркшопа его ведущий подготовил письменный обзор, в котором были конкретизированы рассмотренные метрики. На последующей встрече участники расставили приоритеты для этих метрик. В данном случае механизмы сбора данных и вычисления метрик были ключевыми системными требованиями, и они напрямую повлияли на дизайн программной архитектуры и область применения проекта.

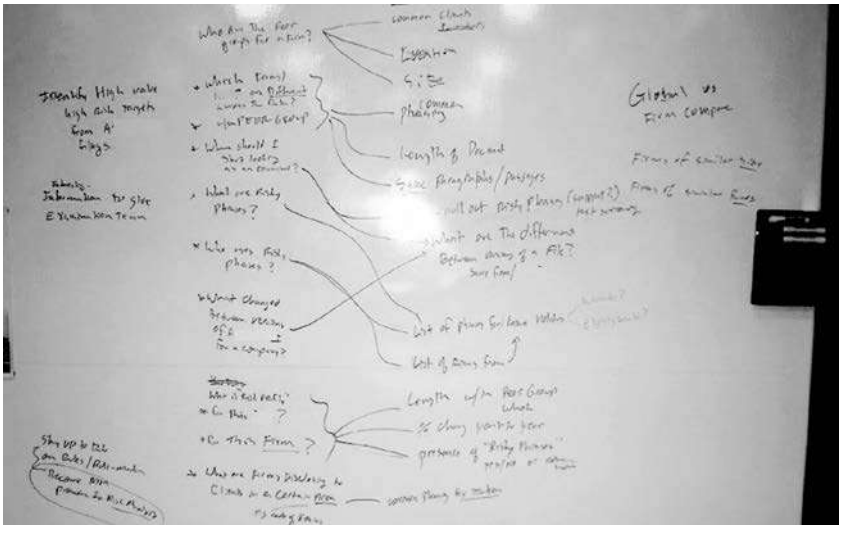


Рис. 10.5. Пример GQM-дерева на доске во время GQM-воркшопа

Заключение

На своей первой работе после университета я попал в талантливую команду, которая занималась оценкой сложной, распределенной, критически важной для безопасности системы, работавшей в реальном времени. В ходе этой оценки мы собрали и использовали огромное количество метрик. Наш тимлид любил повторять: «Метрика только указывает на проблему, но не

решает ее». GQM-подход предоставляет необходимый контекст, чтобы правильно интерпретировать метрики на практике.

Как программному архитектору, мне бывает совсем не сложно помочь командам разработчиков получить метрики, которые для них наиболее важны, особенно когда у них нет четкого понимания, что именно нужно измерять. В разное время я могу заниматься стратегическим планированием, формулировать цели и ключевые результаты, готовиться к запуску продукта, оценивать качество программного компонента, решать, что делать с техническим долгом, или помогать командам проектировать архитектуру системы и подбирать для нее метрики, чтобы достичь определенных атрибутов качества.

GQM — один из основных методов в моем арсенале. Он полезен и как самостоятельный инструмент для анализа, и как инструмент для обучения команд, и как средство для проведения воркшопов, чтобы достичь согласия между всеми стейкхолдерами. GQM бывает полезен вне зависимости от того, нужно ли документировать метрики или просто начать обсуждение того, как их реализовать и как собрать данные. Начните с цели. Задайте вопросы, которые позволят вам оценить эту цель. Проведите мозговой штурм, чтобы найти метрики, которые помогут ответить на эти вопросы.

Для того, что вам хорошо известно, найти метрики легко, зато самые полезные метрики помогут вам оценить то, что вы понимаете еще не до конца. Конечно, то, чего мы не понимаем, измерить сложнее всего — и техника GQM поможет вам найти свой фарватер в этих мутных водах.

Об авторах

Кристиан Чичери (Christian Ciceri) — программный архитектор и соосновательь компании Apiumhub (https://apiumhub.com), которая занимается программной разработкой и известна высоким качеством своих архитектурных решений. Он также является руководителем направления программной архитектуры в компании VYou (https://www.vyou-app.com/en), которая разрабатывает приложение для управления идентификацией и доступом клиентов, и главой группы модераторов саммита Global Software Architecture Summit (https://gsas.io). Его профессиональная деятельность началась с особого интереса к объектно-ориентированному проектированию, а также с глубокого изучения паттернов и техник проектирования как на уровне кода, так и на уровне архитектуры. В прошлом он активно применял гибкие методологии, в частности экстремальное программирование, и имеет опыт работы с такими практиками, как разработка через тестирование (TDD), непрерывная интеграция, пайплайны сборок и эволюционное проектирование.

Кристиан всегда стремился к обширным техническим знаниям, поэтому он исследовал широкий спектр технологий и архитектурных стилей, включая Java, .NET, динамические и чисто скриптовые языки, разработку нативных приложений на C++, классическую многоуровневую и доменно-ориентированную архитектуры, классическую сервис-ориентированную архитектуру (SOA) и корпоративные сервисные шины (ESB). По его собственным словам, «программный архитектор должен создавать рабочую экосистему, которая позволяет командам выпускать масштабируемые, предсказуемые и малозатратные программные продукты».

Дэвид Фарли (Dave Farley) — один из первопроходцев в области непрерывной доставки и идейный лидер, а также эксперт-практик в вопросах непрерывной доставки, DevOps, разработки через тестирование и программной разработки вообще.

На протяжении многих лет, начиная с ранних этапов развития компьютерных вычислений, Дэйв был программистом, разработчиком, системным архитектором, а также успешно руководил несколькими командами. Опираясь на понимание фундаментальных принципов работы компьютеров и приложений, он сформировал новаторские подходы, которые изменили

современную программную разработку. Дэйв бросал вызов традиционному мышлению и возглавлял команды при создании программных продуктов мирового уровня.

Он является соавтором книги «Continuous Delivery», удостоенной премии Jolt, часто выступает на конференциях и ведет успешный, активно развивающийся YouTube-канал, посвященный программной инженерии. Дэйв создал одну из самых быстрых финансовых бирж в мире, стоял у истоков концепции разработки через поведение (Behavior-Driven Development, BDD), написал новый бестселлер «Modern Software Engineering» (Addison-Wesley, 2021)[1], стал соавтором «The Reactive Manifesto» и лауреатом премии Duke's Choice Award за разработку продукта с открытым исходным кодом LMAX Disruptor.

Дэйв стремится помогать командам разработчиков по всему миру повышать надежность и качество проектирования их продуктов, делясь своим опытом через консалтинг, свой YouTube-канал и обучающие курсы.

Нил Форд (Neal Ford) — директор, программный архитектор и, как он сам себя называет — meme wrangler («укротитель мемов») в Thoughtworks, международной ИТ-консалтинговой компании, которая специализируется исключительно на комплексной разработке и доставке ПО. До прихода в Thoughtworks Нил занимал должность технического директора в DSW Group Ltd., широко известной компании в сфере обучения и повышения квалификации сотрудников.

Нил получил степень в области computer science в Университете штата Джорджия по специальности «Языки программирования и компиляторы», а также дополнительную специализацию в области математики с уклоном в статистический анализ. Он также проектирует и разрабатывает приложения, учебные материалы, пишет статьи для журналов и создает видеопрезентации. Кроме того, является автором девяти книг, не считая совместных работ, подобных этой. В консалтинге он в основном фокусируется на проектировании и создании крупномасштабных корпоративных приложений. Нил — всемирно признанный спикер, который за 15 с лишним лет выступил более чем на 1000 конференций для разработчиков по всему миру. Если вы хотите узнать о Ниле больше, зайдите к нему на сайт nealford.com — он будет рад вашим отзывам по адресу nford@thoughtworks.com.

Эндрю Хармел-Лоу (Andrew Harmel-Law) — активный и ответственный ведущий технический специалист в Thoughtworks. Эндрю специализи-

[1] Фарли Д. «Современная программная инженерия. ПО в эпоху эджайла и непрерывного развертывания». СПб., издательство «Питер».

руется на технологиях Java/JVM, Agile-доставке, инструментах сборки и автоматизации, а также на предметно-ориентированном проектировании. У него есть опыт работы на всех этапах жизненного цикла программной разработки во многих секторах, включая государственный сектор, банковское дело и электронную коммерцию. Ему особенно нравится работать над созданием крупномасштабных программных решений, отвечающих сложным требованиям клиентов. Он понимает, что в таких проектах люди, инструменты, архитектура и процессы играют ключевую роль. Эндрю стремится как можно активнее делиться своим опытом, и это стремление проявляется не только в его работе как консультанта, но и неофициально, в формате наставничества, статей в блоге, выступлений и организаций конференций, а также публикаций своего кода в открытом доступе. С ним можно связаться через Twitter (@al94781), где он с удовольствием общается на любые темы, связанные с ПО и комиксами.

Майкл Килинг (Michael Keeling) — программный разработчик в Kiavi и автор книги «Design It! From Programmer to Software Architect». До прихода в Kiavi он работал в IBM над сервисом Watson Discovery Service и имеет опыт работы с различными программными системами, включая сервис-ориентированные архитектуры, корпоративные поисковые системы и даже боевые системы. Майкл — признанный спикер, обладатель профессиональных наград, который регулярно участвует в жизни сообществ архитекторов и сторонников Agile. Имеет степень магистра в области программной инженерии Университета Карнеги — Меллона и степень бакалавра в области computer science Колледжа Вильгельма и Марии. Его текущие исследовательские интересы включают методы и паттерны проектирования и человеческий фактор в программной инженерии. С ним можно связаться через Twitter (@michaelkeeling) или его веб-сайт https://www.neverletdown.net.

Карола Лилиенталь (Carola Lilienthal) — старший программный архитектор и управляющий директор компании WPS (Workplace Solutions). Она посвятила свою деятельность проектированию хорошо структурированных и долгосрочных программных систем. С 2003 года она и ее команды используют для этой цели предметно-ориентированное проектирование (DDD). DDD и устойчивая архитектура ПО стали темами ее многочисленных выступлений на конференциях, включая O'Reilly Software Architecture Conference. Свой опыт она обобщила в книге «Sustainable Software Architecture» (изд-во Rocky Nook, 2019), а также в книге Вона Вернона (Vaughn Vernon) «Domain-Driven Design Distilled» (Addison-Wesley, 2016), которую перевела на немецкий язык.

Жоао Роза (João Rosa) — независимый консультант (https://joaorosa.consulting), специализирующийся на развитии организаций — он помогает им

преодолеть разрыв между стратегией и ее реализацией. В рамках своей консалтинговой практики выступает в роли стратегического советника руководителей высшего звена и временно занимает такие позиции, как технический директор (CTO) или директор по продукту и технологиям (CPTO). Жоао любит делиться знаниями и ведет подкаст Software Crafts Podcast (https://oreil.ly/3BPYl), где берет интервью у представителей индустрии программной разработки, начиная беседу с обсуждения определенной эвристики или паттерна. Вместе со своим хорошим другом Кенни Баасом-Швеглером (Kenny Baas-Schwegler, @kenny_baas) он подготовил книгу «Visual Collaboration Tools», в которой собраны практические истории специалистов со всего мира. Доходы от продаж этой книги направляются на развитие технологических инициатив (https://oreil.ly/PkqWZ), поддерживающих инклюзивность и разнообразие. Жоао также является публичным спикером и тренером; больше информации можно найти на его веб-сайте (https://joaorosa.io).

Александр фон Цитцевиц (Alexander von Zitzewitz) — сооснователь и директор компании hello2morrow, которая разработала Sonargraph (https://oreil.ly/apldy), мощный статический анализатор кода. Он также известен как спикер на конференциях, тренер и консультант, специализирующийся на программной архитектуре и метриках. Александр занимается программной разработкой и управлением командами разработчиков с 1980-х годов. Он убежден, что соблюдение всего нескольких правил в отношении архитектуры и метрик может кардинально улучшить результат любого программного проекта. Недавно разработал новые метрики (https://oreil.ly/QACxH), выявляющие структурные разрушения в программных проектах на ранних стадиях. В 2008 году он переехал из Германии в США. Среди его хобби — пешие прогулки, игры-стратегии и джаз. Имеет ученую степень в области computer science, полученную в Мюнхенском техническом университете (Technische Universität) в Германии.

Рене Вайсс (Rene Weiss) — технический директор (CTO) в компании Finabro. Более 13 лет развивает и поддерживает инициативы по гибкой программной разработке в различных условиях и отраслях, выступая в таких ролях, как программный разработчик, архитектор, менеджер проектов, Scrum-мастер, владелец продукта и руководитель отдела программной разработки. Он также проводит семинары по программной архитектуре и консультирует команды по вопросам в этой области. В свободное от работы с клиентами время Рене пишет доклады и рассказывает об (эволюционной) программной архитектуре, а также о том, как развивать архитектуру с помощью фитнес-функций. Является спикером на многих международных конференциях, включая конференцию O'Reilly Software Architecture Conference.

Эойн Вудс (Eoin Woods) занимает должность технического директора (CTO) в компании Endava, где определяет техническую стратегию, курирует развитие компетенций и руководит инвестициями в новые технологии. Эойн является автором множества публикаций как в исследовательских, так и в отраслевых сообществах, а также соавтором книг «Software Systems Architecture» (Addison-Wesley, 2011) и «Continuous Architecture in Practice» (Addison-Wesley, 2021). В 2018 году Институт программной инженерии при Университете Карнеги — Меллона удостоил его премии Linda Northrup Award за вклад в программную архитектуру. Он регулярно выступает на конференциях и является активным участником лондонского сообщества разработчиков. Его основные технические интересы включают программную архитектуру, DevOps, информационную безопасность и энергоэффективность программных продуктов.

Иллюстрация на обложке

На обложке этой книги изображены горные тупайи (*Tupaia montana*). Эти животные в изобилии встречаются только в родных для них горных лесах острова Борнео и нигде более.

Горные тупайи живут социальными группами, в которых, возможно, доминируют два самца. Они питаются членистоногими, такими как пауки и ракообразные, а также фруктами. В 2016 году Международный союз охраны природы признал популяцию горных тупай стабильной и отнес этот вид к категории вызывающих наименьшие опасения.

Многие животные на обложках изданий O'Reilly находятся под угрозой исчезновения; все они важны для мира.

Иллюстрация на обложке выполнена Карен Монтгомери (Karen Montgomery) на основе старинной гравюры из ботанического сада *Le Jardin Des Plantes*.

Нил Форд, Дэвид Фарли, Кристиан Чичери, Эндрю Хармел-Лоу,
Майкл Килинг, Карола Лилиенталь, Жоао Роза,
Александр фон Цитцевиц, Эойн Вудс, Рене Вайсс

Метрики программной архитектуры. Кейсы, повышающие качество ПО

Перевел с английского Е. Бартов

Изготовитель: ТОО «Спринт Бук». Место нахождения и фактический адрес:
010000, Казахстан, город Астана, район Алматы, проспект Ракымжан Кошкарбаев, д. 10/1, н. п. 18.

Дата изготовления: 09.2026. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 20.07.26. Формат 70×100/16. Бумага офсетная. Усл. п. л. 19,350. Тираж 500. Заказ 0000.

Марк Ричардс, Нил Форд

ФУНДАМЕНТАЛЬНЫЙ ПОДХОД К ПРОГРАММНОЙ АРХИТЕКТУРЕ

2-е международное издание



Хотите выйти за пределы простого кодинга и сделать мощный карьерный рывок?

Многие сильные разработчики мечтают о роли архитектора ПО — одной из самых высокооплачиваемых и востребованных позиций в индустрии. Но раньше путь к ней был полон сомнений: «А вдруг я не потяну? Где взять реальное руководство?»

Теперь такое руководство существует!

В этом обновленном издании вы найдете всесторонний обзор многочисленных аспектов программной архитектуры, в том числе несколько новых глав, посвященных последним достижениям в этой области. Как действующие, так и будущие архитекторы смогут изучить паттерны архитектуры, определение компонентов, визуализацию архитектуры, работу с данными, топологии команд, облачные решения, работу с генеративным ИИ и многие другие темы.

Марк Ричардс и Нил Форд обладают бесценным практическим опытом; в течение многих лет они профессионально проводили курсы по архитектуре ПО, уделяя особое внимание принципам построения архитектуры, применимым ко всем технологическим стекам. Они предлагают современный взгляд на архитектуру ПО с учетом всех нововведений последнего десятилетия.

SPRINT book



Нил Форд, Ребекка Парсонс,
Патрик Куа, Прамод Садаладж

ЭВОЛЮЦИОННАЯ АРХИТЕКТУРА. АВТОМАТИЗИРОВАННОЕ УПРАВЛЕНИЕ ПРОГРАММНЫМ ОБЕСПЕЧЕНИЕМ

2-е международное издание

Новые инструменты, фреймворки методики и парадигмы вновь и вновь меняют экосистему разработки программного обеспечения. Непрерывный прогресс основных практик разработки на протяжении последних пяти лет заставил искать новые пути и подходы к архитектуре, чтобы соответствовать постоянно меняющимся требованиям пользователей. В обновленном издании авторы Нил Форд, Ребекка Парсонс, Патрик Куа и Прамод Садаладж приводят реальные примеры, соответствующие потребностям современной разработки ПО.

SPRINT book